

Scheduling independent tasks under budget and time constraints

Yiqin Gao

▶ **To cite this version:**

Yiqin Gao. Scheduling independent tasks under budget and time constraints. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Lyon, 2021. English. NNT : 2021LYSEN051 . tel-03412631v2

HAL Id: tel-03412631
https://theses.hal.science/tel-03412631v2

Submitted on 3 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.




Numéro National de Thèse : 2021LYSEN051

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N°512
École Doctorale en Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 29/09/2021, par :

Yiqin GAO

Scheduling independent tasks under budget and time constraints

Ordonnancement de tâches indépendantes sous contraintes budgédaire et temporelle

Devant le jury composé de :

Olivier	BEAUMONT	Directeur de Recherche, INRIA	*Rapporteur*
Arnaud	LEGRAND	Directeur de Recherche, CNRS	*Rapporteur*
Florina	CIORBA	Associate Professor, University of Basel	*Examinatrice*
Georges	DA COSTA	Maître de conférences, Université Paul Sabatier	*Examinateur*
Veronika	REHN-SONIGO	Maîtresse de conférences, Université de Bourgogne Franche-Comté	*Examinatrice*
Yves	ROBERT	Professeur, ENS de Lyon	*Co-encadrant de thèse*
Frédéric	VIVIEN	Directeur de recherche, INRIA	*Directeur de thèse*

Contents

Introduction vii

French summary xi

1 Related work 1
- 1.1 Scheduling stochastic independent tasks on homogeneous platform . . . 1
  - 1.1.1 Discrete distributions . . . 2
  - 1.1.2 Continuous distributions . . . 2
- 1.2 Resource provisioning on cloud platform . . . 3
- 1.3 Bags of tasks . . . 3
- 1.4 Imprecise computations and anytime tasks . . . 4
- 1.5 Non-clairvoyant task scheduling . . . 4
- 1.6 Firm real-time tasks . . . 5
- 1.7 Job queues . . . 5
- 1.8 Task pruning in heterogeneous computing systems . . . 5
- 1.9 Multi-criteria real-time tasks scheduling . . . 6

2 Scheduling stochastic tasks on heterogeneous cloud platforms 9
- 2.1 Introduction . . . 9
- 2.2 Problem definition . . . 10
  - 2.2.1 Platform and tasks . . . 10
  - 2.2.2 Constraints and optimization objective . . . 11
- 2.3 Complexity results . . . 12
  - 2.3.1 Problem instance with $b = Kd$ . . . 12
  - 2.3.2 NP-completeness . . . 13
  - 2.3.3 Greedy heuristic . . . 13
- 2.4 Experiments . . . 16
  - 2.4.1 Cutting threshold heuristics . . . 16
  - 2.4.2 Processor selection heuristics . . . 16
  - 2.4.3 Parameters . . . 17
  - 2.4.4 Result synthesis for all distributions . . . 18
  - 2.4.5 Lognormal distribution . . . 19
  - 2.4.6 Summary . . . 21
- 2.5 Conclusion . . . 21

3 Scheduling stochastic tasks with unknown probability distribution 29
3.1 Introduction . . . 29
3.2 Problem definition . . . 30
3.3 Threshold estimation for unknown distributions . . . 31
3.3.1 The empirical distribution function . . . 32
3.3.2 Survival analysis and the Kaplan-Meier estimator . . . 32
3.4 Taking decisions . . . 34
3.4.1 *One-size-fits-all* policies . . . 34
3.4.2 Automatic inference . . . 35
3.5 Experiments . . . 36
3.5.1 Experimental methodology . . . 36
3.5.2 Results . . . 36
3.6 Conclusion . . . 50

4 Efficient task-dropping strategies for firm real-time systems 51
4.1 Introduction . . . 51
4.2 Problem framework . . . 52
4.3 Heuristics . . . 53
4.4 Resolution of Markov chain . . . 55
4.4.1 RoundRobin strategy . . . 55
4.4.2 EarliestStartTime strategy . . . 65
4.4.3 Limit behavior of the Markov chain . . . 67
4.5 Performance evaluation . . . 68
4.5.1 Experimental methodology . . . 68
4.5.2 Results . . . 69
4.6 Conclusion . . . 83

5 Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms 85
5.1 Introduction . . . 85
5.2 Model . . . 87
5.2.1 Platform and tasks . . . 87
5.2.2 Reliability . . . 88
5.2.3 Power and energy . . . 88
5.2.4 Optimization Objective . . . 90
5.2.5 Complexity . . . 90
5.3 Mapping . . . 92
5.4 Scheduling . . . 93
5.5 Lower bound . . . 97
5.6 Performance evaluation . . . 98
5.6.1 Experimental methodology . . . 98
5.6.2 Results . . . 99
5.7 Conclusion . . . 112

Conclusion 113

Bibliography 117

Publications **127**

Introduction

After entering the information age, the speed of acquiring information increases constantly, and computational science has become a critical tool in various scientific research fields, such as weather forecasting, aerospace, biomedical, etc., in order to complete computationally intensive tasks. Computers play also an important role in our daily life. They are frequently used in different places like schools, banks, governments, etc.

In order to meet the ever-increasing requirements in terms of information processing (e.g., storage, computing performance), building high-performance computers (i.e., supercomputers) has turned out indispensable. The computational works (i.e., tasks) in different fields have different characteristics and requirements. Many of them have time constraints: Tasks completed after their predefined deadline have little or no value. Furthermore, the available budget is usually also limited. Budget here can refer to monetary or energy resources.

The main goal of this thesis is to design scheduling heuristics for independent tasks under budget and time constraints, in order to satisfy different criteria (i.e., system performance, energy consumption or reliability). More precisely, given a set of independent tasks with their deadline, and a platform composed of identical or different processors, we need to decide how to allocate tasks on the platform, in which order to execute them on each processor, in order to meet their requirements. We deal with diverse requirements in our work: meet a given reliability threshold, or maximize the number of tasks successfully executed. As for the budget, we may have limited monetary resource available, or we need to minimize the energy consumption. The problem becomes more complicated with multiple constraints and objectives. But inevitably, such complicated problems appear in applications and systems of more and more research domains. Therefore, we urgently need to study the solution to these problems.

There exists diverse task models in different computational problems. *Real-time tasks* is a popular one used in many applications. Real-time systems (or tasks) are systems where tasks are input periodically and must complete successfully before a fixed time-interval called the deadline. In the literature, real-time systems (or tasks) are classified as *hard* or *soft* systems [5]. For hard real-time tasks, no deadline should be missed: it is mandatory that each task completes before its deadline. This is related to our work in Chapter 5, in which we have a set of periodic tasks and the deadline of each task instance should be matched. On the contrary, for soft real-time tasks, some deadlines can be missed. There are two scenarios then: either a task that missed its deadline must still be completed, or it can be ignored. Our work in Chapter 4 refers to the latter case: we have periodic tasks which are allowed to miss their deadline, and there is no value to complete them after their deadline. This is the case of firm real-time tasks. Examples of applications for firm real-time tasks include multimedia applications, satellite-based tracking, financial forecast systems, and robotic assembly lines [53, 56, 59]. In our work of Chapter 4, we assume that the tasks are skippable. While the processors are occupied, all submitted jobs are waiting for their turn (or waiting to be dropped) in a job queue. We consider a platform

composed of identical processors, and the objective is to maximize the expected number of tasks successfully completed before their own deadline, especially in the case that the system is overloaded.

In Chapters 2 and 3 of this thesis, we consider another task model. We have a set of independent tasks for which the execution times follow the same distribution. We name that "stochastic tasks" or "bag of tasks". Furthermore, in our task model, it is not necessary for all tasks to be completely processed to obtain a meaningful result. They are all available in the beginning and have common budget and deadline constraints. The scheduler can decide to interrupt a (long) running task at any time and then launch a new one, hoping that the latter can take less time to complete, but the budget already spent for the interrupted task is lost. The objective is to successfully complete as many tasks as possible, but it does not matter which ones succeed. This problem is closely related to imprecise computations. Most often, tasks in imprecise computations are divided into a mandatory and an optional part: while the execution of all mandatory parts is necessary, the execution of optional parts is decided by the user. Often, the user has not the time or the budget to execute all optional parts, and she must select which ones to execute. Our work in Chapters 2 and 3 corresponds to optimizing the processing of the optional parts. Among domains where tasks may have optional parts (or where some tasks may be entirely optional), one can cite recognition and mining applications [69], robotic systems [48], speech processing [27]; and [55] also cites multimedia processing, planning, artificial intelligence, and database systems. In these applications, the processing times of the optional parts are heavily data-dependent, hence the need to estimate them via a probability distribution.

In addition to the task model, as already mentioned above, there is also a variety of requirements in computational problems. It is shown that transient faults are much more frequent than permanent faults and represent the major fraction of detected errors. Transient faults are induced mainly by outside disturbances, such as neutron and alpha particle strikes [85, 93]. Nowadays, the scale of computational problems and digital systems is getting larger and larger. Although fault rates of a single component in the platform may not be high, with the increase of the scale of problems to be solved, the scale of the supercomputer increases as well and is more likely to encounter a fault. Hence, it is important to ensure correct scientific computations in face of transient faults. In the work in Chapter 5, we have a set of periodic tasks, as discussed above, and a heterogeneous platform. The worst case execution time (WCET) of each task on each processor is known in advance. We use the classical replication strategy to guarantee the reliability target of each task face to transient fault.

In Chapters 2 and 3, we consider a budget constraint. In these two chapters, the global cost of a processor is the product of its usage time and its unit cost per second, and the budget can refer to a monetary resource constraint. But in many cases, it is necessary to establish power management techniques when providing scheduling strategies, not only due to monetary constraints, but also for protecting the environment. Presently, many domains are pursuing sustainable development. It is also the case in supercomputing. The list of Green500 [60], which is announced twice a year with TOP500, ranks the supercomputers in the world by energy efficiency. It means that performance is no longer the only indicator for evaluating supercomputers. Energy-efficiency becomes one of the critical objectives in a variety of problems. In Chapter 5, we aim at minimizing the global energy consumption while meeting the reliability target and the deadline constraint mentioned above. However, minimizing energy consumption is antagonisitc to the requirement of reliability, because we need extra resources to avoid faults

(e.g., replication method in our work). Hence, we should deal with the trade-off between the reliability and the energy.

The rest of the thesis is organized as following: In Chapter 1, we review the related work of this thesis, and we present the previous work of our team, in which authors designed scheduling strategies for stochastic independent tasks under budget and deadline constraints on homogeneous platforms, in order to maximize the number of tasks successfully executed. The first three works extends the state-of-the-art in three different directions: In Chapter 2, we consider the same task model but applied to an heterogeneous platform. In Chapter 3, we treat the problem under a non-clairvoyant case: we do not know in advance the task execution times, and we must learn them during the execution. After that, in Chapter 4, instead of having all tasks available in the beginning, we consider a real-time system: tasks arrive periodically and have their own deadline. Finally, we study a more complicated problem in Chapter 5: we have periodic tasks and a heterogeneous platform. We need to find a heuristic which minimizes the expected energy consumption, while matching deadline and reliability constraints of all tasks. The main contributions of each chapter are summarized below.

Chapter 1: Related work

This is a preliminary chapter in which we introduce related work, as well as previous which this thesis is building upon.

Chapter 2: Scheduling stochastic tasks on heterogeneous cloud platforms [C1, R1]

In this chapter, we extend the previous work to the case of an heterogeneous platform. In the previous work, the target platform was made of identical processors, and the main questions are how many processors to enroll and whether and when to interrupt tasks if they have been executing for a long time. The authors presented an asymptotically optimal strategy in which they stop all unsuccessful tasks after executing for a duration l^{opt}. This duration was named the optimal cutting threshold, and this threshold can be calculated numerically according to the distribution $\mathcal{D}$ of the task execution times. As for this work, the cloud platform is composed of several types of processors, where each type has a unit execution cost that depends upon its characteristics. The task execution times follow a variety of standard probability distributions (exponential, uniform, half-normal, lognormal, gamma, inverse-gamma and Weibull) whose mean and standard deviation both depend upon the processor type. In this work, the main questions are which processors to enroll, and whether the optimal cutting threshold provided in the previous work is efficient in the case of an heterogeneous platform. We assess the complexity of the problem by showing its NP-completeness and providing a 2-approximation for the asymptotic case where budget and deadline both tend to infinity. Then we introduce several heuristics and compare their performance by running an extensive set of simulations.

Chapter 3: Scheduling stochastic tasks with unknown probability distribution [R3]

Based on the results obtained in the previous work and in Chapter 2, in this chapter, we go back to the homogeneous platform, and we study the scheduling strategy when task execution times are not known before execution; instead, the only information available to the scheduler is that they obey the same (unknown) probability distribution. The scheduler needs to acquire some information before deciding for a cutting threshold. In addition, the cutting threshold may be re-evaluated as new information is acquired when the execution progresses further. This

work presents several strategies to determine a good cutting threshold, and to decide when to re-evaluate it. In particular, we use the Kaplan-Meier estimator to account for tasks that are still running when making a decision. The efficiency of our strategies is assessed through an extensive set of simulations with various budget and deadline values, and ranging over a variety of probability distributions.

Chapter 4: Efficient task-dropping strategies for firm real-time systems

As stated above, Chapters 2 and 3 deal with the problem in which all tasks are available at time 0 and have the same deadline. In this chapter, we are no longer in an offline case. In contrast, task instances arrive periodically and have their own deadline. The real-time version of the problem considered in this chapter is dramatically more complicated, because the pressure on the system at each time-step plays an important role, and dynamic scheduling decisions must be taken. In this chapter, we design several heuristics which decide whether and when to interrupt long-lasting tasks, and when a processor is idle, to launch which task in the waiting queue. We discretize the time and construct a Markov Chain. We prove that this chain is aperiodic and irreducible, and we compute with this theoretical model the expected performance of our heuristics. On the practical side, a comprehensive set of simulations with variety of parameters is launched, in order to evaluate our heuristics.

Chapter 5: Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms [R2, C2, R4]

Low energy consumption and high reliability are widely identified as increasingly relevant issues in real-time systems on heterogeneous platforms. Thus, after focusing on success rate, in this chapter, we propose a multi-criteria optimization strategy to minimize the expected energy consumption while enforcing the reliability threshold and meeting all task deadlines. The platform is composed of processors with different (and possibly unrelated) characteristics, including speed profile, energy cost and failure rate. The tasks arrive periodically, as in Chapter 4. But instead of following a certain probability distribution, the information known about task execution times in this work is their WCETs on each processor. Each instance of a task is replicated to ensure a prescribed reliability threshold. We provide several mapping and scheduling heuristics to solve this challenging optimization problem. Specifically, a novel approach is designed to control (i) how many replicas to use for each task, (ii) on which processor to map each replica and (iii) when to schedule each replica for each task instance on its assigned processor. Different mappings achieve different levels of reliability and consume different amounts of energy. Scheduling matters because once a task replica is successful, the other replicas of that task instance are canceled, which calls for minimizing the amount of temporal overlap between any replica pair. The experiments are conducted for a comprehensive set of execution scenarios, with a wide range of processor speed profiles and failure rates.

French summary

Après l'entrée dans l'ère de l'information, la vitesse d'acquisition de l'information augmente et la science informatique devient un outil essentiel dans divers domaines de la recherche scientifique, tels que les prévisions météorologiques, l'aérospatiale, le biomédical, etc., afin d'effectuer des tâches à forte intensité de calcul. Les ordinateurs jouent également un rôle important dans notre vie quotidienne. Ils sont fréquemment utilisés dans différents endroits comme les écoles, les banques, les gouvernements, etc.

Afin de répondre aux exigences croissantes en termes de traitement de l'information (par exemple, stockage, performances de calcul), la construction d'ordinateurs de hautes performances (i.e., superordinateurs) devient indispensable. Les travaux de calcul (i.e., tâches) de différents domaines ont des conditions et des exigences différentes. Beaucoup d'entre eux ont des contraintes de temps: les tâches terminées après leur échéance prédéfinie ont peu ou pas de valeur. D'un autre côté, le budget dont nous disposons est généralement également limité. Le budget peut ici faire référence à des ressources monétaires ou énergétiques.

Par conséquent, l'objectif principal de cette thèse est de concevoir des heuristiques d'ordonnancement pour des tâches indépendantes sous contraintes de budget et de temps, afin de satisfaire différents critères (i.e., performances du système, consommation d'énergie ou fiabilité). Plus précisément, étant donné un ensemble de tâches indépendantes avec leur échéance, et une plate-forme composée de processeurs identiques ou différents, nous devons décider comment allouer les tâches sur la plate-forme, dans quel ordre les exécuter sur chaque processeur, afin de répondre aux contraintes. Nous traitons de diverses contraintes dans notre travail: atteindre un seuil de fiabilité prédéfinie, ou maximiser le nombre de tâches exécutées avec succès. En ce qui concerne le budget, nous pouvons disposer d'une limite de ressources monétaires ou nous devons minimiser la consommation d'énergie. Le problème devient plus compliqué avec de multiples contraintes et objectifs. Mais inévitablement, un tel problème apparaît dans les applications et les systèmes de plus en plus de domaines de recherche. Il est donc urgent d'étudier la solution à ces problèmes.

Il existe divers modèles de tâches dans différents problèmes de calcul. Le modèle des tâche temps réel est un modèle populaire utilisé dans de nombreuses applications. Les systèmes (ou tâches) en temps réel sont des systèmes dans lesquels les tâches sont entrées périodiquement et doivent se terminer avec succès avant un intervalle de temps fixe appelé échéance. Dans la littérature, les systèmes (ou tâches) en temps réel sont classés en systèmes *dur* ou *souple* [5]. Pour les tâches en temps réel dures, aucune échéance ne doit être manquée : il est obligatoire que chaque tâche se termine avant son échéance. Ceci est lié à notre travail dans le chapitre 5, dans lequel nous avons un ensemble de tâches périodiques et la date limite de chaque instance de tâche doit être respectée. Au contraire, pour les tâches en temps réel souples, certaines échéances peuvent être dépassées. Il y a alors deux scénarios : soit une tâche qui a dépassé son échéance doit encore être terminée, soit elle peut être ignorée. Notre travail dans le chapitre 4

se réfère à ce dernier cas : nous avons des tâches périodiques qui sont possible de manquer leur échéance, et il n'y a aucune sens de les terminer après leur date limite. C'est le cas des tâches temps réel fermes. Des exemples d'applications pour des tâches en temps réel fermes incluent les applications de multimédias, le suivi de trajet basé sur des satellites, les systèmes de prévision financière et les chaînes de montage robotiques [53, 56, 59]. Dans notre travail du chapitre 4, nous supposons que les tâches sont interruptibles. Pendant que les processeurs sont occupés, tous les travaux soumis attendent leur tour (ou attendent d'être abandonnés) dans une file d'attente de travaux. Nous considérons une plate-forme composée de processeurs identiques, et l'objectif est de maximiser l'espérance du nombre de tâches accomplies avec succès avant leur propre échéance, surtout dans le cas où le système est surchargé.

D'autre part, dans les chapitres 2 et 3 de cette thèse, nous considérons un autre modèle de tâche. Nous avons un ensemble de tâches indépendantes pour lesquelles les temps d'exécution suivent la même distribution. Nous appelons cela « tâches stochastiques » ou « sac de tâches ». De plus, dans notre modèle de tâches, il n'est pas nécessaire que toutes les tâches soient complètement traitées pour obtenir un résultat significatif. Ils sont tous disponibles au départ et ont des contraintes de budget et de temps communes. L'ordonnanceur peut décider d'interrompre à tout moment une tâche (longue) en cours d'exécution puis en lancer une nouvelle, en espérant que cette dernière puisse prendre moins de temps à se terminer, mais le budget déjà dépensé pour la tâche interrompue est perdu. L'objectif est d'accomplir avec succès autant de tâches que possible, mais nous ne nous soucions pas savoir lesquelles sont réussies. Cette définition est étroitement liée aux calculs imprécis (*imprecise computations*). Le plus souvent, les tâches aux calculs imprécis sont divisées en une partie obligatoire et une partie facultative : alors que l'exécution de toutes les parties obligatoires est nécessaire, l'exécution des parties facultatives est décidée par l'utilisateur. Souvent, l'utilisateur n'a pas le temps ou le budget pour exécuter toute la partie facultative, et elle doit sélectionner ce qu'elle va exécuter. Notre travail dans les chapitres 2 et 3 correspond à l'optimisation du traitement de cette partie optionnelle. Parmi les domaines où les tâches peuvent avoir une partie facultative (ou certaines tâches peuvent être entièrement facultatives), on peut citer les applications de reconnaissance et de minage [69], les systèmes robotiques [48], le traitement de la parole [27] ; et [55] cite également le traitement multimédia, la planification, l'intelligence artificielle et les systèmes de bases de données. Dans ces applications, le temps de traitement de la partie optionnelle est fortement dépendant des données, d'où la nécessité de les estimer via une distribution de probabilité.

En plus du modèle de tâche, comme déjà mentionné ci-dessus, il existe également une variété d'exigences dans les problèmes de calcul. Il est montré que, parmi toutes les erreurs détectées, les défauts permanents ne causent qu'une petite fraction par rapport aux défauts transitoires. Les erreurs transitoires sont principalement induites par des perturbations extérieures, telles que les impacts de neutrons et de particules alpha [85, 93]. De nos jours, l'échelle des problèmes de calcul et des systèmes numériques est de plus en plus grande. Bien que les taux d'erreur d'un seul composant de la plate-forme ne soient pas élevés, avec l'augmentation de l'échelle des problèmes à résoudre, la taille du superordinateur augmente également et est plus susceptible de rencontrer une panne. Par conséquent, il est important d'assurer un calcul scientifique correct face aux erreurs transitoires. Dans le travail du chapitre 5, nous avons un ensemble de tâches périodiques, comme déjà mentionné ci-dessus, et une plate-forme hétérogène. Le temps d'exécution dans le pire des cas (WCET) de chaque tâche sur chaque processeur est connu à l'avance. Nous utilisons la réplication, une stratégie classique, pour garantir une fiabilité prédéfinie de chaque tâche face à une faute transitoire.

Comme mentionné ci-dessus, dans les chapitres 2 et 3, nous considérons une contrainte budgétaire. Dans ces deux chapitres, le coût global d'un processeur est le produit de son temps d'utilisation et de son coût unitaire de temps, et le budget peut faire référence à une contrainte de ressources monétaires. Mais il est nécessaire d'établir une gestion de l'énergie tout en proposant des stratégies d'ordonnancement dans de nombreux cas, non seulement en raison de contraintes monétaires, mais aussi pour la protection de l'environnement. À l'heure actuelle, de nombreux domaines poursuivent le développement durable. C'est aussi le cas du supercalcul. La liste des Green500 [60], qui est annoncée deux fois par an avec TOP500, classe les superordinateurs dans le monde par efficacité énergétique. Cela signifie que la performance n'est plus le seul indicateur pour évaluer les superordinateurs. L'efficacité énergétique devient l'un des objectifs critiques dans une variété de problèmes. C'est ce à quoi nous sommes confrontés dans le chapitre 5, dans lequel nous devons minimiser la consommation globale d'énergie tout en respectant l'objectif de fiabilité et la contrainte de temps mentionnés ci-dessus. Cependant, minimiser la consommation d'énergie est contraire à l'exigence de fiabilité, car nous avons besoin de ressources supplémentaires pour éviter les pannes (par exemple, la méthode de réplication dans notre travail). Par conséquent, nous devrions traiter le compromis entre la fiabilité et l'énergie.

Le reste de la thèse est organisé comme ci-dessous: Dans le chapitre 1, nous revoyons l'état-de-l'art de cette thèse, et nous présentons le travail précédent de notre équipe, dans lequel les auteurs ont conçu une stratégie d'ordonnancement pour des tâches indépendantes stochastiques sous contraintes de budget et de temps sur une plate-forme homogène , afin de maximiser le nombre de tâches exécutées avec succès. Les trois premiers travaux étendent l'état de l'art dans trois directions différentes: Dans le chapitre 2, nous considérons le même modèle de tâche dans le cas d'une plate-forme hétérogène. Au chapitre 3, nous traitons le problème sous un cas non-clairvoyant: nous ne connaissons pas à l'avance la distribution du temps d'exécution des tâches, et nous devons l'apprendre pendant l'exécution. Après cela, au chapitre 4, au lieu d'avoir toutes les tâches disponibles au début, nous considérons un système en temps réel: les tâches arrivent périodiquement et ont leur propre échéance. Enfin, nous étudions un problème plus compliqué au chapitre 5: nous avons des tâches périodiques et une plate-forme hétérogène. Nous devons trouver une heuristique qui minimise l'espérance de la consommation d'énergie, tout en respectant les contraintes de temps et de fiabilité de toutes les tâches. Les principales contributions de chaque chapitre sont résumées ci-dessous:

Chapitre 1: Etat de l'art

Il s'agit d'un chapitre préliminaire dans lequel sont introduits les états-de-l'art et le travail précédent à partir duquel cette thèse est construite.

Chapitre 2: Ordonnancement de tâches stochastiques sur des cloud plate-formes hétérogènes [C1, R1]

Dans ce chapitre, nous étendons le travail précédent au cas des plate-formes hétérogènes. Dans le travail précédent, nous sommes dans un cas homogène, et les principales questions sont le nombre de processeurs à utiliser et, si et quand interrompre les tâches si elles s'exécutent depuis longtemps. Les auteurs ont présenté une stratégie asymptotiquement optimale dans laquelle ils arrêtent toutes les tâches non-accomplies après avoir exécuté pendant le temps l^{opt}. Les auteurs l'ont nommé le seuil à couper optimal, et ce seuil peut être calculé numériquement en fonction de la distribution des temps d'exécution des tâches $\mathcal{D}$. Quant à ce travail, la

plate-forme de type *cloud* est composée de plusieurs types de processeurs, où chaque type a un coût d'exécution unitaire qui dépend de ses caractéristiques. Les temps d'exécution des tâches suivent une variété de distributions de probabilité standard (exponentielle, uniforme, semi-normale, lognormale, gamma, gamma inverse et Weibull) dont la moyenne et l'écart-type dépendent tous deux du type de processeur. Dans ce travail, les principales questions sont de savoir quels processeurs à utiliser et si le seuil à couper optimal fourni dans le travail précédent est efficace dans le cas de plate-forme hétérogène. Nous évaluons la complexité du problème en montrant sa NP-complétude et en fournissant une 2-approximation pour le cas asymptotique où le budget et le temps tendent tous deux vers l'infini. Ensuite, nous introduisons plusieurs heuristiques et comparons leurs performances en exécutant un ensemble extensif de simulations.

Chapitre 3: Ordonnancement de tâches stochastiques avec une distribution de probabilité inconnue [R3]

Basé sur des résultats obtenus dans le travail précédent et dans Chapitre 2, dans ce chapitre, nous revenons à une plate-forme homogène, et nous étudions la stratégie d'ordonnancement lorsque les temps d'exécution des tâches ne sont pas connus avant l'exécution; en revanche, la seule information disponible pour l'ordonnanceur est qu'ils obéissent à la même distribution de probabilité (inconnue). Le ordonnanceur doit acquérir certaines informations avant de décider d'un seuil à couper. De plus, le seuil à couper peut être réévalué au fur et à mesure que de nouvelles informations sont acquises lorsque l'exécution progresse. Ce travail présente plusieurs stratégies pour déterminer un bon seuil à couper, et pour décider quand le réévaluer. En particulier, nous utilisons l'estimateur de Kaplan-Meier pour tenir compte des tâches en cours d'exécution lors de la prise de décision. L'efficacité de nos stratégies est évaluée à travers un vaste ensemble de simulations avec diverses valeurs de budget et d'échéance, portant sur une variété de distributions de probabilité.

Chapitre 4: Stratégies efficaces de suppression de tâches pour les systèmes en temps réel fermes

Comme indiqué ci-dessus, les chapitres 2 et 3 traitent du problème dans lequel toutes les tâches sont disponibles au temps 0 et ont la même échéance. Dans ce chapitre, nous ne sommes plus dans un cas offline. En revanche, les instances de tâche arrivent périodiquement et ont leur propre échéance. La version temps réel du problème considéré dans ce chapitre est plus compliqué, car la pression sur le système à chaque pas de temps joue un rôle important, et des décisions d'ordonnancement dynamiques doivent être prises. Dans ce travail, nous concevons plusieurs heuristiques qui décident si et quand interrompre les tâches de longue durée, et quand un processeur est disponible, de lancer quelle tâche dans la file d'attente. Nous discrétisons le temps et construisons une chaîne de Markov. Nous montrons que cette chaîne est apériodique et irréductible, et nous calculons avec ce modèle théorique l'espérance de performance de notre heuristique. Du côté pratique, un ensemble de simulations avec une variété de paramètres sont lancés, afin d'évaluer nos heuristiques.

Chapitre 5: Stratégies écoénergétiques pour l'allocation de tâches en temps réel axée sur la fiabilité sur des plate-formes hétérogènes [R2, C2, R4]

La faible consommation d'énergie et la fiabilité élevée sont largement identifiées comme des problèmes de plus en plus pertinents dans les systèmes en temps réel sur des plate-formes

hétérogènes. Ainsi, après s'être focalisé sur le taux de réussite, dans ce chapitre, nous proposons une stratégie d'optimisation multicritères pour minimiser l'espérance de consommation d'énergie tout en respectant le seuil de fiabilité et en respectant toutes les échéances des tâches. La plate-forme est composée de processeurs avec des caractéristiques différentes (et éventuellement sans corrélation), y compris la vitesse, le coût énergétique et le taux d'échec. Les tâches arrivent périodiquement, comme dans le chapitre 4. Mais au lieu de suivre une certaine distribution de probabilité, les informations connues sur les temps d'exécution des tâches dans ce travail sont leurs WCET sur chaque processeur. Chaque instance d'une tâche est répliquée pour garantir un seuil de fiabilité prescrit. Nous proposons plusieurs heuristiques d'attribution et d'ordonnancement pour résoudre ce problème d'optimisation difficile. Plus précisément, une nouvelle approche est conçue pour contrôler (i) le nombre de répliques à utiliser pour chaque tâche, (ii) sur quel processeur attribuer chaque réplique et (iii) quand ordonnancer chaque réplique pour chaque instance de tâche sur son processeur attribué. Différentes attributions atteignent différents niveaux de fiabilité et consomment différentes quantités d'énergie. L'ordonnancement est important car une fois qu'une copie de tâche est réussi, les autres copies de cette instance de tâche sont annulés, ce qui nécessite de minimiser la superposition temporel entre des paires de copies. Les expériences sont exécutées pour un grand ensemble de scénarios, avec une large gamme de vitesse et de taux d'échec pour les processeurs.

Chapter 1

Related work

We overview in this section the related work and the previous work of this thesis. At first, we present in Section 1.1 the previous work based on which we establish our proper research. Secondary, the work in this thesis falls under the scope of cloud computing since it targets to schedule independent tasks on a cloud platform under deadline and budget constraints. We overview the resource provisioning on cloud platform in Section 1.2. After that, in Chapters 2 and 3, we consider bags of tasks and imprecise computations which will be presented respectively in Section 1.3 and 1.4. Furthermore, in Chapter 3, task execution times obey a probability distribution which is unknown before execution, which is closely related to non-clairvoyant scheduling, which we survey in Section 1.5. As for the work in Chapter 4, we consider firm real-time tasks in Section 1.6 and job queue in Section 1.7, which are related with the model in this work. On the other hand, we present in Section 1.8 a set of works which consider similar problem as that in Chapter 4. Finally, in Section 1.9, we present existing works in the field of multi-criteria real-time tasks scheduling, in either homogeneous and heterogeneous platforms, which is very closely related to our work in Chapter 5.

1.1 Scheduling stochastic independent tasks on homogeneous platform

In this section, we recall previous results in [12, 13], which provide scheduling strategies for stochastic independent tasks on homogeneous platform under common budget and deadline constraints. As mentioned in the introduction, we have a bag of tasks for which the execution times follow a common probability distribution $\mathcal{D}$ which is known in advance. And we have a set of identical processors. The budget spent by a processor is the product of its execution time by the unit execution cost of that processor. Authors deal with the following two questions: Firstly, how many processors should be enrolled? Secondly, should all tasks be allowed to run until completion, or should some tasks be interrupted and, in the latter case, which tasks and when?

For the first question, authors find that, with a budget of b and a deadline of d, we should enroll $\left\lceil \frac{b}{d} \right\rceil$ processors.

Then comes the key problem in this work: A scheduling policy has to decide whether and when should unsuccessful tasks be interrupted. Of course, interrupting a running task is a risky decision, because: (i) the time and budget already spent to execute the current task will be lost

if it gets interrupted; and (ii) there is no guarantee that the new task will complete faster than the interrupted one.

Consider a given processor and a given distribution of task execution times with expected value μ and standard deviation σ. Authors propose a fixed-threshold strategy. A fixed-threshold strategy interrupts every not-yet-completed task at a predefined threshold l (i.e., when the task has been executing for a time l without completing). They introduce the OptRatio heuristic which is proved to be an asymptotically optimal policy for discrete distributions and extend it to continuous distributions. Known a distribution of task execution times, OptRatio finds an optimal cutting threshold l^{opt}. It means that, all tasks will be interrupted after executing for time l^{opt} without successfully completing. The idea behind OptRatio is that, cutting unsuccessful tasks at l^{opt} maximizes (asymptotically) the ratio $\mathcal{E}(l)$ of the probability of success to the expected execution time spent for a single task, when each task is interrupted at time l. As for the simulation, OptRatio has been shown to perform very well for a wide range of budget and deadline values. The calculation in discrete and continuous cases is presented in the following sections:

1.1.1 Discrete distributions

In this section, authors consider a discrete distribution $\mathcal{D}$ under which there are k possible task execution times, $w_1 < w_2 < ... < w_k$. A task has an execution time w_i with probability p_i, with $0 \leq p_i \leq 1$ and $\sum_{j=1}^{k} p_j = 1$. The success rate of the strategy with threshold l is computed as follows:

$$\mathcal{E}(l) = \begin{cases} 0 \quad \text{if } l < w_1 \\ \dfrac{\sum_{j=1}^{\mathbb{I}(l)} p_j}{\sum_{j=1}^{\mathbb{I}(l)} p_j w_j + \left(1 - \sum_{j=1}^{\mathbb{I}(l)} p_j\right) l} \quad \text{otherwise} \end{cases} \tag{1.1}$$

where $\mathbb{I}(l)$ is the index of the largest task execution time smaller than or equal to l: $\mathbb{I}(l) = k$ if $l \geq w_k$, and $w_{\mathbb{I}(l)} \leq l < w_{\mathbb{I}(l)+1}$ otherwise. This complicated formula has an intuitive explanation: the probability of success with cutting threshold l is $\sum_{j=1}^{\mathbb{I}(l)} p_j$, and the execution time is averaged as follows: some tasks have (successfully) executed in w_j seconds, with probability p_j, for each $j \leq \mathbb{I}(l)$, and the remaining tasks have been interrupted after l seconds (with the remaining probability $\left(1 - \sum_{j=1}^{\mathbb{I}(l)} p_j\right)$). Thus, the optimal threshold is defined as following:

$$l^{opt} = \underset{l \in \{w_1, ..., w_k\}}{\arg\max} \; \mathcal{E}(l).$$

1.1.2 Continuous distributions

In this case, $\mathcal{D}$ is a continuous distribution whose cumulative distribution function is $F(x)$ and its probability density function $f(x)$. The execution time of a task is defined by a random variable X which follows $\mathcal{D}$. With these notations, the probability that the execution is no longer than a duration t is: $P(X \leq t) = F(t)$. Then, the equation of the yield of the fixed-threshold strategy of threshold l is easily extrapolated from that for discrete distributions (Equation 1.1):

$$\mathcal{E}(l) = \frac{F(l)}{\int_0^l x f(x) dx + l(1 - F(l))} \tag{1.2}$$

The optimal threshold is then, like previously:

$$l^{opt} = \arg\max_{l} \mathcal{E}(l).$$

For most distributions, l^{opt} cannot be computed analytically, but authors provide a program [14] to compute it numerically.

Thus, the final conclusion of this work is following: With several identical processors available and a distribution of task execution times known in advance, we enroll $\left\lceil \frac{b}{d} \right\rceil$ processors and execute on each of them the fixed-threshold strategy of threshold l^{opt}.

1.2 Resource provisioning on cloud platform

Resource provisioning and scheduling are key steps to the efficient execution of workflows on cloud platforms. Singh and Chana published a survey devoted solely to cloud resource provisioning [84], that is, the decision of which resources should be enrolled to perform the computations. Resource scheduling decides which computations should be processed by each of the enrolled resources and in which order they should be performed.

In Chapters 2, 3 and 4, we refine the classical deterministic model by adding stochasticity to task execution times. We observe that the stochastic context has not received much attention. Indeed, most of the studies assume a clairvoyant setting: the resource provisioning and task scheduling mechanisms know in advance, and accurately, the execution time of all tasks. A handful of additional studies also consider that tasks may fail [63, 75]. Among these articles, Poola et al. [75] differ as they assume that tasks have uncertain execution times. However, they assume they know these execution times with a rather good accuracy (the standard deviation of the uncertainty is 10% of the expected execution time). They are thus dealing with uncertainties rather than a true non-clairvoyant setting. The work in [10] targets stochastic tasks but is limited to taking static decisions (no task interruption).

On the other hand, in Chapter 5, we are facing a multi-criteria scheduling problem. We can find extensive studies which consist in meeting deadlines and either respecting a budget or minimizing the energy consumption for deterministic workflows [2, 4, 7, 9, 24, 30, 65, 66, 94]. [95] maximizes the reliability of an energy-constrained DAG executed on a heterogeneous platform while using DVFS. Conversely, [100] minimizes the energy consumption of a reliability-constrained DAG executed on a heterogeneous platform while using, or not, DVFS. A group of authors published a book [98] and several articles on the problem of DAG scheduling on heterogeneous platforms. In Chapter 2 of book [98] and in [96] these authors consider the energy minimization when scheduling a DAG with or without DVFS. However, these two references do not consider reliability. In [97] they considered the same problem while satisfying some reliability goal. Some works are limited to a particular type of application like MapReduce [47, 50, 88]. For instance, Tian and Chen [88] consider MapReduce programs and can either minimize the financial cost while matching a deadline or minimize the execution time while enforcing a given budget.

1.3 Bags of tasks

A bag of tasks is an application composed of a set of independent tasks sharing some common characteristics: either all tasks have the same execution time or they are instances sampled

from the same probability distribution. This is exactly the case of the first three works in this thesis. There exists a survey about resource optimization for bag of tasks applications [87]. Several works devoted to bag-of-tasks processing explicitly target cloud computing [8, 16, 37, 73]. Most of them [8, 16, 37] consider the classical clairvoyant model, in which we know the exact execution time or its distribution, or the uncertain model, in which we know its range or its standard deviation, while [73] targets a non-clairvoyant setting. Vecchiola et al. [92] consider a single application comprising independent tasks with deadlines but without any budget constraints. In their model, tasks are supposed to have different execution times but they only consider the average execution time of tasks rather than its probability distribution (this is left for future work). Moreover, they do not report on the amount of deadline violations; their contribution is therefore hard to assess. Mao et al. [67] consider both deadline and budget constrained provisioning and assume they know the tasks execution times up to some small variation (the largest standard deviation of a task execution time is at most 20% of its expected execution time). Hence, this work is more related to scheduling under uncertainties than to stochastic tasks scheduling.

1.4 Imprecise computations and anytime tasks

In the works in Chapter 2 and 3, task model assumes that some tasks may not be executed. This model is very closely related to imprecise computations [3, 19, 62]. Furthermore, this task model also corresponds to the overload case of [6] where jobs can be skipped or aborted. Another related model, is that of anytime tasks [52] where a task can be interrupted at any time, with the assumption that the longer the running, the higher the quality of its output. Such a model requires a function relating the time spent to a notion of reward. Finally, we note that the general problem related to interrupting tasks falls into the scope of optimal stopping, the theory that consists in selecting a date to take an action, in order to optimize a reward [28].

1.5 Non-clairvoyant task scheduling

Most of the related works surveyed so far assume a fully or semi clairvoyant set of task execution times, which is not always true in a realistic scenario. In contrast, our model in Chapter 3 considers a fully non-clairvoyant case, in which we have no information in advance about the execution times of our bag of tasks. Although this topic has received less attention, we can still find several references. For instance, Sungjin et al. [51] and Pawan et al. [83] both worked on online algorithms. They assume that the size of arriving tasks is not known before completing them. In [51], a unified model is designed for several different scheduling problems, while [83] aims at minimizing flow-time and energy. In the work of Li [58], task execution times are unknown, and the objective is to minimize the makespan while using one or several multicore processors. A group of authors [72, 73, 74] has published several studies focusing on budget-constrained makespan minimization. They do not assume to know the distribution of execution times but try to learn it on the fly [72, 74]. This work differs from ours as these authors do not consider deadlines. For instance, in [73], the objective is to try to complete all tasks, possibly using replication on faster processors, and, in case the proposed solution fails to achieve this goal, to complete as many tasks as possible. The implied assumption is that all tasks can be completed within the budget. Using replication to ensure the completion of all tasks is our objective and strategy in Chapter 5. But in Chapters 2, 3 and 4, we implicitly assume the

opposite: there are too many tasks to complete all of them by the deadline, and therefore we attempt to complete as many as possible.

1.6 Firm real-time tasks

Many real-time systems enforce hard deadlines that cannot be missed. But as stated in the introduction, in Chapter 4, we consider soft real-time tasks which allow the missing of some deadlines. In particular, for *firm* real-time tasks, there is no value to complete a task after its deadline, hence the objective function is to maximize the number of tasks that are successfully executed. There are many applications that involve firm real-time tasks [53, 56, 59].

In a general firm real-time system, there are several periodic tasks that are input to the system, and the scheduler has to choose among the task instances which ones will be launched and which ones will not. Usually, it is assumed that task execution times can take stochastic values within a prescribed interval: each task has a *Worst-Case Execution Time (WCET)* that is an upper bound on the execution time [45]. Mapping decisions are taken statically, based upon the WCETs. On the contrary, scheduling decisions can be taken either statically, based upon the total utility of the parallel platform, or dynamically, reclaiming slack intervals based upon actual execution times. This is exactly the case in Chapter 5. But in the work of Chapter 4, we extend the state-of-the-art to probability distributions with unbounded support, by interrupting long-lasting tasks after a given duration threshold has been reached.

1.7 Job queues

As already mentioned, in the work of Chapter 4, we deal with the list of waiting jobs via a queue. It is however very different from what is done in queueing theory systems [46] where the typical constraint is to select in which order jobs should be executed to optimize an objective such as their response time. In this work, jobs all follow the same probability distribution. We use a simple First-Come-First-Served strategy, in order to select which jobs we execute next among jobs that have not been *dropped*.

The closer to our work is the job dropping model [20], where upon arrival of a job, the system selects, a priori, whether it should be *dropped* (i.e. should not be added to the queue). The decision to drop a job often depends on a function of queue parameters (number of jobs in the queue, average load of the queue): linear function (average queue size) [29] or other more complicated functions [26, 78]. Then, all jobs from the queue are executed (for example following a FCFS strategy). Contrarily to the job dropping model, in Chapter 4, we decide to drop jobs *a posteriori*, that is, once it is their turn to be scheduled, the algorithm decides based on information on their deadline and on their distribution of execution time whether they should be executed or dropped.

1.8 Task pruning in heterogeneous computing systems

In a series of publications [22, 23, 36, 70, 80], authors consider an oversubscribed heterogeneous computing system to which tasks are submitted at random times. Similar as in our work of Chapter 4, authors have been investigated task pruning techniques in order to maximize the proportion of tasks that are successfully executed on the system. Different from our heuristic which contains one job queue, in their work, when a task is released, it is at first stored in a

batch queue, and then can be allocated to the *machine queue* of a processor by the mapping process. At each mapping event (completion of an old task or arrival of a new task), the success probability of all tasks in the machine queue is recomputed, via a costly convolution over all possible durations of the tasks in the queue weighted by their respective probabilities. Tasks in the batch queue are also considered when there remains a processor with available positions after the previous computation. Tasks with low probability to meet their deadline are dropped from the machine queue and deferred in the batch queue. Tasks are deferred mean that their assignment to a processor is postponed. In contrast, a task dropped from the machine queue is definitely removed out of the system. Contrarily to our problem, there are several task types, several processor types, and task arrival dates are random rather than not periodic. This calls for a very costly solution where a whole convolution over a large time window must be recomputed at each mapping event. However, some tasks can be pruned (dropped) after some duration, which is a strategy that we also investigate in our work. The striking difference is that with a single task type and periodic releases, we are able to determine the optimal value of the key scheduling parameters once and for all and to apply them on the fly, thereby providing a schedule whose cost is constant and independent of the number of tasks that are released.

1.9 Multi-criteria real-time tasks scheduling

Section 1.2 introduced a set of works in the domain of resource provisioning and task scheduling. However, these studies do not consider real-time applications. The periods and deadlines which constrain real-time tasks render problems significantly harder to tackle. This is the case of our work in Chapters 4 and 5, in which we consider a set of periodic tasks. Section 1.6 presented the special case of firm real-time tasks which is a model related to Chapters 4. Instead of focusing on the number of tasks executed, Chapter 5 consider at the same time energy consumption, deadline and reliability. Thus, in this section, we will find a few works about multi-criteria real-time tasks scheduling. We will start with works on homogeneous platform, and then extend to those on heterogeneous platform.

Liu and Layland first introduced the Earliest Deadline First (EDF) and the Rate Monotonic (RM) scheduling policies for real-time systems and provided the utilization bounds for both policies in 1973 [61]. Since then, the real-time scheduling problem has been extensively studied. There exists a very significant literature on real-time scheduling for multiprocessor systems. However, most work is devoted to homogeneous processor systems, as exemplified by the survey [21] which ignores altogether heterogeneous systems, and by the more recent survey [82] where only 9 of the 78 references deal with heterogeneous platforms. [45] minimizes the energy when scheduling independent tasks with different deadlines on a homogeneous platform while satisfying some threshold on reliability. The study [43] improved the solution from [45], in particular by carefully avoiding overlaps between primary and secondary replicas. [44] considers the same problem; however, it uses checkpointing to cope with failures when all other works consider replication.

We refer the interested reader to [21, 43, 45, 82] for a comprehensive overview of the related work for homogeneous platforms. Heterogeneous platforms make the problem even harder because processors can have different speeds, energy costs, and failure rates. Therefore, the processor preferred for one task by one of the objectives and constraints —deadline satisfaction, energy minimization, reliability threshold satisfaction— may be the worst processor for another objective or constraint. The heuristics have thus to perform complicated trade-offs in these three-criteria settings.

Some related works target the scheduling of real-time applications on heterogeneous platforms, but without considering fault tolerance. For instance, [103] targets the execution of a DAG, but considering neither energy consumption nor fault-tolerance (when DAGs are scheduled, tasks are always assumed to have the same deadline). [42] targets the execution of independent tasks that access shared resources, the access to resources being exclusive. Their objective is to maximize the number of instances for which a solution is found. [71], [79] and [102] minimize energy consumption by using DVFS, [71] when scheduling independent tasks, [79] a DAG, and [102] a moldable application. [91] considers the scheduling of independent tasks and DAGs under an energy constraint, while [89] considers the scheduling of independent tasks under a thermal constraint. [101] proposes a fully polynomial-time approximation scheme (FPTAS) for minimizing the energy consumption for a set of independent tasks executed on a set of heterogeneous (unrelated) processing elements.

On the other hand, some related work considers the execution of real-time applications on heterogeneous failure-prone platforms but is limited to coping with a single failure per task or per processor. [76] maximizes the reliability of the considered DAG but does not consider energy consumption and follows the primary/backup technique and, thus, is limited to at most one failure per task of the DAG. [77] attempts to maximize resource utilization (and does not consider energy) when scheduling a set of independent tasks. It assumes that at most one processor can fail, which enables the simultaneous scheduling of several backup tasks on the very same processor, since at most one of them will need to be executed. [49] minimizes the energy consumed for the execution of a DAG while satisfying a reliability threshold. The proposed solution uses DVFS and Power Mode Management (i.e., the ability to switch off idle processors to low-power inactive state). This solution, however, cannot produce a schedule more reliable than the original one. It also supports at most one fault per processor. [39] minimizes the energy consumed for the execution of a set of independent tasks while satisfying a reliability threshold using DVFS and following a primary-backup approach.

Very few studies consider the execution of real-time applications on heterogeneous failure-prone platforms and can cope with two or more failures per task. [86] minimizes the energy consumed for the execution of a set of independent tasks while satisfying a reliability threshold. The proposed solution uses DVFS. This solution, however, is based on a primary-backup approach that is then extended. This approach, by design, cannot produce a schedule more reliable than the original one with two replicas per task, strongly relies on DVFS, and schedules several replicas of a same task on the same processor (what most other approaches forbid). [38] targets the execution of a DAG on a heterogeneous platform while satisfying a reliability threshold. However, the objective is not the minimization of energy consumption but the maximization of the *utilization of energy consumption*, which can be seen as a yield of reliability improvement with respect to increased energy consumption. As a consequence, [38] produces energy greedy schedules (see subplots (a-1), (b-1), and (c-1) of Figure 1 in [38]). In Chapter 3 of the already mentioned book [98], the authors consider cost minimization when scheduling a DAG under deadline and reliability constraints. Therefore, we consider the same problem but for a set of independent tasks rather than for a DAG. Because of the dependence between tasks and the chosen as-soon-as-possible scheduling of [98], this solution tends to schedule simultaneously the different replicas of a single task. As already pointed out in the studies [43, 45] this can lead to a significant waste of energy. Therefore, it would have been unfair to compare our solution to that of [98] applied to independent tasks.

From what precedes, we have identified only a single existing solution that enables to schedule real-time tasks on heterogeneous platforms while minimizing energy consumption and sat-

isfying some bound on the overall reliability. However, this solution being dedicated to DAGs lacks the possibility to minimize overlapping between replicas of a same task, which has been previously shown to be crucial [43] and which we have given special care to in our work (see Section 5.4).

Chapter 2

Scheduling stochastic tasks on heterogeneous cloud platforms

2.1 Introduction

In this first chapter, we deal with the following problem: given a cloud platform and a bag of stochastic tasks, how to maximize the number of successful task executions, given a budget and a deadline. The cloud platform is composed of several processor types, each with a different unit cost and computing capacity. The execution time of the tasks follows a different probability distribution on each processor type, in order to account for their different performance. For instance, the expectation of the distribution of task durations on a given processor can be inversely proportional to the raw speed of that processor, while the standard deviation can account for the interplay between task profiles and processor parameters, such as memory usage, communication pattern, etc. In this chapter, we use an extensive set of widely used distributions, namely exponential, uniform, half-normal, lognormal, gamma, inverse-gamma and Weibull distributions.

This task model assumes that some tasks may not be executed in the end. In fact, there are three cases: (i) some tasks are launched and reach completion, meaning that they are successfully executed: (ii) some tasks are launched but they are interrupted before completion, meaning that their execution has failed; and (iii) some tasks are not launched at all. The objective is to maximize the number of successful tasks, given the deadline and budget constraints. This scheduling problem naturally arises with many applications in the context of information retrieval. Informally, the goal is to extract as much information as possible, by launching analysis tasks whose execution time strongly depends upon the nature of the data sample being processed. A typical example is a set of image files, whose processing times heavily depend upon the elements that are present (or not) within each image. Not all data samples must be processed, but the larger the number of data samples successfully processed, the more accurate the analysis. Furthermore, as mentioned in the introduction, this task model is closely related to *imprecise computations* [3, 19, 62], particularly in the context of real-time computations.

As stated in 1.1, with a single processor, the problem is to decide whether, and when, to interrupt a long-lasting task, with the hope to launch a new one that would execute faster. Previous work [12, 13] showed that there exists an optimal threshold at which each running task should be interrupted. Interrupting each yet unsuccessful task when it reaches this optimal cutting threshold is shown to maximize the expected *success rate* on the processor, i.e., the

average number of tasks successfully executed per time unit. This cutting threshold depends upon the probability distribution of task execution times and is computed numerically.

With several processors of different types, the problem becomes dramatically more complicated, because we have to decide how many processors to enroll, and of which type. In addition to success rate, the unit cost of the processor plays an important role. In fact, the key parameter is the *yield*, defined as the ratio of the success rate over the unit cost: it gives the expected number of successful tasks per budget unit. Intuitively, one would like to sort available processors by non-increasing yields, and greedily enroll them in this order. With this greedy algorithm, there remains to determine how many processors to enroll. We show how to determine this number and call GREEDY the resulting greedy algorithm with the optimal number of processors. Unfortunately, GREEDY is not optimal. In fact, we show that the problem to decide which processor to enroll is NP-complete, but we also show that GREEDY is guaranteed to be a 2-approximation. These results lay the foundation for the complexity of the problem with several processors. On the practical side, we compare GREEDY with a variety of other heuristics, using an extensive set of simulations, and observe that it always achieve a close-to-optimal performance, which makes it the heuristic of choice for the target optimization problem.

The main contributions of this chapter are the following:

- We provide several theoretical results (NP-completeness, the GREEDY approximation algorithm and performance lower bound) for the problem instance with large budget and deadline. These results show the difficulty of the optimization problem under study, and lay the foundations for its analysis;
- We compare the performance of GREEDY to that of several heuristics for the general problem with arbitrary deadline and budget values, and for all the probability distributions mentioned above. Not only GREEDY is superior to the other heuristics, but its performance is very close to the lower bound on most instances. Altogether, GREEDY provides a robust approach to the problem.

The rest of the chapter is organized as follows. We detail the framework and objective in Section 2.2. We provide complexity results (NP-completeness and 2-approximation algorithm) in Section 2.3. We compare these heuristics in Section 2.4, assessing their performance for an extensive set of simulation parameters. Finally, we provide concluding remarks in Section 2.5.

2.2 Problem definition

This section details the framework and the scheduling objective. See Table I for a summary of main notations.

2.2.1 Platform and tasks

We aim at scheduling a set of independent stochastic tasks on a cloud platform. The cloud platform is composed of a set of different processors, each with their own characteristics. In the abstract formulation of the problem, there is a set $\mathcal{P} = \{m_1, m_2, \ldots, m_M\}$ of M processors. Each processor has a unit cost: c_i is the amount of budget spent per unit of time for executing a task on m_i. The execution time of a task on m_i obeys a probability distribution $\mathcal{D}_i$ which is chosen as a probability distribution whose mean and standard deviation both depend on the characteristics of m_i. The rationale for such a framework is the following. First, we assume that task execution times are data-dependent, as is the case in many applications, and therefore exhibit stochastic behaviors which can be nicely modeled by a probability distribution. Second, task execution

Table I: Summary of main notations.

	Platform
$\mathcal{P}$	platform
M	number of processors
m_i	the i-th processor
c_i	unit cost of m_i
l_i	cutting threshold for task interruption on m_i
$\mathcal{E}_i$	success rate of m_i, computed using l_i
$\mathcal{Y}_i$	yield of m_i, where $\mathcal{Y}_i = \frac{\mathcal{E}_i}{c_i}$
$\mathcal{Y}^{tot}$	total platform yield
k	number of processor categories
n_j	number of processors of type j (hence $M = \sum_{j=1}^{k} n_j$)
	Tasks
$\mathcal{D}_i$	probability distribution of execution times on m_i
μ_i, σ_i	mean, standard deviation of $\mathcal{D}_i$
	Constraints
b	budget
d	deadline
K	ratio b/d

times cannot be easily encapsulated as a mere function of the number of cores of their host processor, because many parameters such as memory usage and communication patterns must be taken into account. Therefore, it would not make sense to consider a unique probability distribution and simply scale it by a unique parameter, say the number of cores of each processor, to induce actual execution times on that processor. Instead, we use a different probability distribution for each processor, with values of mean and standard deviation accounting for the heterogeneity of sources. It makes sense to assume that the mean μ_i of $\mathcal{D}_i$, which is the expectation of execution times on m_i, is somewhat related to the number of cores $nbcores_i$ of m_i. In the experimental section (Section 2.4), we explore scenarios where the mean values μ_i are inversely proportional to the core counts $nbcores_i$, but we vary the standard deviations σ_i to account for a wide range of heterogeneity degrees. We report results for a variety of standard probability distributions (exponential, uniform, half-normal, lognormal, gamma, inverse-gamma and Weibull).

Finally, in many experimental cloud platforms, there is only a reduced set of different processor types, with several available identical processors per type. We let k be the number of types and n_j be the number of available processors for type j, where $\sum_{j=1}^{k} n_j = M$.

2.2.2 Constraints and optimization objective

The user has a limited budget b and an execution deadline d. The optimization problem is to select a subset of processors and to maximize the expected number of tasks that can be successfully completed on these processors before the deadline is reached or the totality of the budget is spent. More precisely, the optimization problem $\textsc{Opt}(\mathcal{P}, b, d)$ is the following:

- Decide which processors to launch: it can be any subset of $\mathcal{P}$;
- Each processor in $\mathcal{P}$ executes tasks continuously, as soon as it is started and until the deadline or the budget is exceeded, whichever comes first;
- At any time and on each processor, decide whether to interrupt the task that is currently executing and launch a new one: each task can be deleted by the scheduler at any time before completion;
- The execution of each task is non-preemptive. In a non-preemptive execution, interrupted tasks cannot be relaunched, and the time/budget spent computing until interruption is completely lost.

2.3 Complexity results

In this section, we present complexity results with several processors, assuming large budget and deadline values. We start by formulating the asymptotic optimization problem in Section 2.3.1. We assess its complexity in Section 2.3.2. Then we introduce a greedy polynomial heuristic in Section 2.3.3, and show that it is a 2-approximation.

2.3.1 Problem instance with $b = Kd$

Consider a given processor $m_i \in \mathcal{P}$. Given the distribution $\mathcal{D}_i$ of task execution times on m_i, we choose a cutting threshold l_i^{cut} at which to interrupt tasks, using any of the methods in Section 1.1 (for instance we take $l_i^{cut} = l_i^{opt}$, the value computed for OptRatio). We then derive the (asymptotic) success rate $\mathcal{E}_i$ (average number of successful tasks per time unit) and the yield $\mathcal{Y}_i = \frac{\mathcal{E}_i}{c_i}$ (average number of successful tasks per cost unit), where c_i is the unit cost of m_i. The asymptotic behavior of m_i is characterized by these two parameters. With several processors, if there is no deadline, the best solution is to use a single processor, namely the one with highest yield $\mathcal{Y}_i$. Introducing a deadline makes parallelism unavoidable, and raises the question of selecting which processors to enroll. In the following, we assume that budget and deadline are proportional: $b = Kd$ for some constant $K \geq 1$, and aim at deriving asymptotic results when b tends to infinity under that constraint. Intuitively, K represents the total cost per time unit available until deadline d. Hence, the potential parallelism that can be achieved.

Now assume that we enroll a subset $\mathcal{Q} = \{m_i, i \in Q\}$ of processors from $\mathcal{P}$. Here, Q simply represents the subset of $\{1, 2, \ldots, M\}$ that records the indices of enrolled processors. These processors will work continuously until the budget is exhausted or the deadline has been reached, whichever comes first. If the processors in $\mathcal{Q}$ work for a duration t, the total budget spent is $t \times \sum_{i \in Q} c_i$. Hence,

$$t = \min\left(d, \frac{b}{\sum_{i \in Q} c_i}\right) = \frac{b}{\max(K, \sum_{i \in Q} c_i)}.$$

Asymptotically, each m_i, with $i \in Q$, is successfully executing $\mathcal{E}_i$ task per time unit. Hence, the total yield of subset $\mathcal{Q}$ is

$$\mathcal{Y}^{tot} = \frac{\sum_{i \in Q} \mathcal{E}_i}{\max(K, \sum_{i \in Q} c_i)}. \tag{2.1}$$

We are ready to define the asymptotic optimization problem with several processors:

Definition 1 (OptHetero)**.** *Given the set $\mathcal{P}$ of available processors and the constraint $b = Kd$, determine the subset $\mathcal{Q}$ of $\mathcal{P}$ so that the value of $\mathcal{Y}^{tot}$ in Equation (2.1) is maximized.*

2.3.2 NP-completeness

In this section, we show that the decision problem associated to OptHetero is NP-complete. For simplicity, we use the same name for the decision and optimization problems.

Theorem 1. OptHetero *is NP-complete.*

Proof. The decision problem is the following: given the set $\mathcal{P}$ of available processors and the constraint $b = Kd$, and given a bound on the total yield $\mathcal{Z}$, can we find a subset $\mathcal{Q}$ of $\mathcal{P}$ with total yield $\mathcal{Y}^{tot} \geq \mathcal{Z}$? The problem obviously belongs to the class NP, a certificate being the subset of enrolled processors, whose yield can be computed in linear time. For the completeness, we make a reduction from SubsetSum, a well-known NP-complete problem [35]. Consider an instance $\mathcal{I}_1$ of SubsetSum: given n positive integers a_1, a_2, ..., a_n and a target T, can we find a subset J of $\{1, 2, \ldots, n\}$ such that $\sum_{i \in J} a_i = T$? We build the following instance $\mathcal{I}_2$ of OptHetero: a platform $\mathcal{P}$ with $M = n + 1$ processors, budget/deadline constraint $b = Kd$ where $K = T + 1$. Processors characteristics are the following:

- m_i, for $1 \leq i \leq n$, has success rate $\mathcal{E}_i = Ka_i$ and unit cost $c_i = a_i$
- m_{n+1} has success rate $\mathcal{E}_{n+1} = 2K$ and unit cost $c_{n+1} = 1$.

Finally, the bound on total yield is $\mathcal{Z} = K + 1$. The size of $\mathcal{I}_2$ is clearly polynomial (and even linear) in the size of $\mathcal{I}_1$. We now show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution. Assume first that $\mathcal{I}_1$ has a solution, i.e., a subset J with $\sum_{i \in J} a_i = T$. If we enroll all processors whose index is in J plus m_{n+1}, we obtain the total yield

$$\mathcal{Y}^{tot} = \frac{\sum_{i \in J} Ka_i + 2K}{\max(K, \sum_{i \in J} a_i + 1)} = \frac{KT + 2K}{\max(K, T + 1)} = K + 1.$$

Hence, a solution to $\mathcal{I}_2$.

Assume now that $\mathcal{I}_2$ has a solution, i.e., an index subset Q with total yield $\mathcal{Y}^{tot} \geq \mathcal{Z} = K+1$. If the last processor is not enrolled, i.e., if $n + 1 \notin Q$, then $\mathcal{Y}^{tot} = \frac{\sum_{i \in Q} Ka_i}{\max(K, \sum_{i \in Q} a_i)} \leq K$, a contradiction. Hence, necessarily $n + 1 \in Q$. Let $J = Q \setminus \{n + 1\}$, we are going to show that J is a solution of $\mathcal{I}_1$. We know that

$$\mathcal{Y}^{tot} = \frac{\sum_{i \in J} Ka_i + 2K}{\max(K, \sum_{i \in J} a_i + 1)} \geq K + 1.$$

Let $U = \sum_{i \in J} a_i$. If $U \geq K$ then $\mathcal{Y}^{tot} = \frac{KU+2K}{U+1} = K + \frac{K}{U+1} < K + 1$, a contradiction. If $U \leq K - 2$ then $\mathcal{Y}^{tot} = \frac{KU+2K}{K} = U + 2 < K + 1$, a contradiction. Hence, $U = K - 1 = T$, and J is a solution to $\mathcal{I}_1$. This concludes the proof. □

2.3.3 Greedy heuristic

The OptHetero problem is similar to a knapsack problem, and a natural heuristic is to enroll processors with highest yield first. Table II shows a little example with a platform $\mathcal{P}$ consisting of $M = 5$ processors. We use $K = 5$ in the example.

In Table II, processors are ordered by non-increasing yield, so the greedy heuristic selects m_1 first, then m_2, etc. The performance achieved is the following:

- Using only m_1: $\mathcal{Y}^{tot} = \frac{10}{\max(5,1)} = 2$;
- Using m_1 and m_2: $\mathcal{Y}^{tot} = \frac{10+6.2}{\max(5,1+3)} = 3.24$;
- Using m_1, m_2 and m_3: $\mathcal{Y}^{tot} = \frac{10+6.2+8}{\max(5,1+3+4)} = 3.025$;

Table II: Example of platform $\mathcal{P}$ ($M = 5$).

Processor	Success rate	Unit cost	Yield
m_1	$\mathcal{E}_1 = 10$	$c_1 = 1$	$\mathcal{Y}_1 = 10$
m_2	$\mathcal{E}_2 = 6.2$	$c_2 = 3$	$\mathcal{Y}_2 \approx 2.1$
m_3	$\mathcal{E}_3 = 8$	$c_3 = 4$	$\mathcal{Y}_3 = 2$
m_4	$\mathcal{E}_4 = 6$	$c_4 = 4$	$\mathcal{Y}_4 = 1.5$
m_5	$\mathcal{E}_5 = 4$	$c_4 = 4$	$\mathcal{Y}_5 = 1$

- Using m_1, m_2, m_3 and m_4: $\mathcal{Y}^{tot} = \frac{10+6.2+8+6}{\max(5,1+3+4+4)} \approx 2.5167$;
- Using all five processors: $\mathcal{Y}^{tot} = \frac{10+6.2+8+6+4}{\max(5,1+3+4+4+4)} = 2.1375$.

In the example, the best choice is to use only m_1 and m_2, for a total yield $\mathcal{Y}^{tot} = 3.24$. In the following, we characterize how many processors should be chosen. Finally, note that in the example, the optimal solution is to use only m_1 and m_3, for a total yield $\mathcal{Y}^{tot} = \frac{10+8}{\max(5,1+4)} = 3.6$.

Proposition 1. *Consider a platform $\mathcal{P}$ with M processors ordered by non-increasing yields and with the constraint $b = Kd$. The total yield $\mathcal{Y}^{tot}$ achieved by the greedy heuristic is maximum when enrolling either the first $i^* - 1$ processors or the first i^* processors, whichever has the higher total yield, where i^* is the smallest index such that $\sum_{i=1}^{i^*} c_i > K$.*

In other words, the greedy heuristic should enroll processors until their cumulated cost exceeds K, and then the best solution is either using all theses processors or using all of them except the last one. In the example of Table II, we have $i^* = 3$ and the best solution is with the first two processors. We let Greedy denote the greedy heuristic which enrolls the optimal number of processors. Note that when two different processors have the same yield, we rank them and use the one with lowest unit cost first, which is better for scenarios where the budget is limited.

Proof. For $1 \leq i \leq M$, we consider the first i processors and define

- the cumulated success rate $\mathcal{S}_i^{tot} = \sum_{j=1}^{i} \mathcal{E}_j$;
- the cumulated cost $\mathcal{C}_i^{tot} = \sum_{j=1}^{i} c_j$;
- the cumulated success/cost ratio $\mathcal{R}_i = \frac{\mathcal{S}_i^{tot}}{\mathcal{C}_i^{tot}}$.

Now the total yield achieved with the first i processors is $\mathcal{Y}_i^{tot} = \min\left(\mathcal{R}_i, \frac{\mathcal{S}_i^{tot}}{K}\right)$. Note that i^* is the smallest index i such that $\mathcal{C}_i^{tot} \geq K$. First we observe that the $\mathcal{R}_i$ are non-increasing. This is because processors are ordered by ratio $\frac{\mathcal{E}_i}{c_i}$. We easily check that

$$\frac{\mathcal{E}_1}{c_1} \geq \frac{\mathcal{E}_2}{c_2} \qquad \Rightarrow \qquad \frac{\mathcal{E}_1}{c_1} \geq \frac{\mathcal{E}_1 + \mathcal{E}_2}{c_1 + c_2} \geq \frac{\mathcal{E}_2}{c_2}$$

and the result follows by induction.

For $i \geq i^*$, we have $\mathcal{Y}_i^{tot} = \mathcal{R}_i \leq \mathcal{R}_{i^*} = \mathcal{Y}_{i^*}^{tot}$. For $i \leq i^* - 1$, we have $\mathcal{Y}_i^{tot} = \frac{\mathcal{S}_i^{tot}}{K} \leq \frac{\mathcal{S}_{i^*-1}^{tot}}{K} = \mathcal{Y}_{i^*-1}^{tot}$. This concludes the proof. □

In order to show that the performance of Greedy is within a factor two of the optimal, we define the FracOptHetero fractional version of the OptHetero problem. The only difference between FracOptHetero and OptHetero is that each processor enrolled at the

beginning can now be stopped at any time before the deadline or the exhaustion of the budget. For FracOptHetero, the total yield is

$$\mathcal{Y}^{tot,frac} = \frac{\sum_{j\in\mathcal{P}} \mathcal{E}_j t_j}{b} \tag{2.2}$$

where t_j is the running time of m_j. Formally:

Definition 2 (FracOptHetero)**.** *Given the set $\mathcal{P}$ of available processors and the constraint $b = Kd$, determine t_j, which is the running time of the j-th processor in $\mathcal{P}$, so that the value of $\mathcal{Y}^{tot,frac}$ in Equation (2.2) is maximized (t_j is null if we do not use the j-th processor). Each m_i in $\mathcal{P}$ obeys the* OptRatio *strategy and interrupts all tasks at time l_i^{opt}, with expected success rate $\mathcal{E}_i$.*

For this problem, the following variant of the greedy algorithm is optimal:

Proposition 2. *Consider a platform $\mathcal{P}$ with M processors ordered by non-increasing yields and with the constraint $b = Kd$. An optimal solution for* FracOptHetero *is obtained by enrolling the first $i^* - 1$ processors until the deadline and enrolling the i^*-th processor to exhaust the rest of the budget, where i^* is the smallest index such that $\sum_{i=1}^{i^*} c_i > K$.*

Proof. For the proof, we assume that i^* does exist, otherwise all processors are enrolled until the deadline, which is optimal. Let t_i^{opt} denote the running time of m_i in the optimal solution, and t_i be its running time in the greedy algorithm. If an optimal solution is not making the greedy choice, there exists an index i such that $t_i^{opt} > t_i$. Because the greedy algorithm uses the first i^*-1 processors until the deadline, we have $i \geq i^*$. Also, because the budget is exhausted by the greedy algorithm (from the existence of i^*), there must exist an index j such that $t_j^{opt} < t_j$. Since the greedy algorithm does not use processors of index $k \geq i^* + 1$, we have $j \leq i^*$, hence $j < i$. With the ordering method in the greedy algorithm, we can conclude that $\mathcal{Y}_i \leq \mathcal{Y}_j$. Then in the optimal solution, we re-distribute the amount of budget $\beta = \min\{(t_j - t_j^{opt})c_j, (t_i^{opt} - t_i)c_i\}$ from m_i to m_j. The first term of β guarantees that, after the re-distribution, m_j spends no more budget than its does in the greedy algorithm. After the re-distribution, the yield of the optimal solution is increased by a nonnegative value $\frac{\beta(\mathcal{Y}_j - \mathcal{Y}_i)}{b}$. If $\mathcal{Y}_i < \mathcal{Y}_j$, this contradicts the optimality. Otherwise, each m_k, where $j \leq k \leq i$ has same yield (because of the ordering method of the greedy algorithm); then the optimal solution and the greedy algorithm have same global yield. This concludes the proof. □

Let $\mathcal{Y}^{opt}$ be the highest yield for OptHetero, and $\mathcal{Y}^{opt\text{-}frac}$ be the highest yield for FracOptHetero problem. From Proposition 2, we know that $\mathcal{Y}^{opt\text{-}frac}$ is achieved by the greedy algorithm, which is given by

$$\mathcal{Y}^{opt\text{-}frac} = \frac{\mathcal{S}_{i^*-1}^{tot}}{K} + \left(1 - \frac{\mathcal{C}_{i^*-1}^{tot}}{K}\right)\frac{\mathcal{E}_{i^*}}{c_{i^*}}. \tag{2.3}$$

Proposition 3. Greedy *is a 2-approximation algorithm for* OptHetero.

Proof. We need to prove that: $\mathcal{Y}_{greedy}^{tot} \geq \frac{1}{2}\mathcal{Y}^{opt}$. We have

$$\mathcal{Y}_{greedy}^{tot} = \max(\mathcal{Y}_{i^*-1}^{tot}, \mathcal{Y}_{i^*}^{tot}) = \max\left(\frac{\mathcal{S}_{i^*-1}^{tot}}{K}, \frac{\mathcal{S}_{i^*}^{tot}}{\mathcal{C}_{i^*}^{tot}}\right).$$

Similarly to the proof of Proposition 1, we can easily prove by induction that $\frac{\mathcal{S}_{i^*}^{tot}}{\mathcal{C}_{i^*}^{tot}} \geq \frac{\mathcal{E}_{i^*}}{c_{i^*}}\cdot$. As $0 \leq 1 - \frac{\mathcal{C}_{i^*-1}^{tot}}{K} \leq 1$, we have $\frac{\mathcal{S}_{i^*}^{tot}}{\mathcal{C}_{i^*}^{tot}} \geq \left(1 - \frac{\mathcal{C}_{i^*-1}^{tot}}{K}\right)\frac{\mathcal{E}_{i^*}}{c_{i^*}}\cdot$ We derive:

$$\begin{aligned}
\mathcal{Y}_{greedy}^{tot} &\geq \max\left(\frac{\mathcal{S}_{i^*-1}^{tot}}{K}, \left(1 - \frac{\mathcal{C}_{i^*-1}^{tot}}{K}\right)\frac{\mathcal{E}_{i^*}}{c_{i^*}}\right) \\
&\geq \frac{1}{2}\left[\frac{\mathcal{S}_{i^*-1}^{tot}}{K} + \left(1 - \frac{\mathcal{C}_{i^*-1}^{tot}}{K}\right)\frac{\mathcal{E}_{i^*}}{c_{i^*}}\right] \\
&= \frac{1}{2}\mathcal{Y}^{opt\text{-}frac} \geq \frac{1}{2}\mathcal{Y}^{opt}.
\end{aligned}$$

□

2.4 Experiments

This section assesses the performance of several strategies to interrupt executing tasks and to choose the number and types of processors to enroll for a given budget and deadline. The algorithms are implemented in R and the related code, data and analysis are publicly available in [33].

2.4.1 Cutting threshold heuristics

We use the same cutting threshold heuristics as in the previous work [12]:

- OptRatio is the heuristic designed in the previous work that we have already introduced in Section 1.1.
- MeanVariance(x) is the family of heuristics that interrupt a task as soon as its execution time reaches $\mu_i + x\sigma_i$, where x is some constant (positive or negative).
- Quantile(x) is the family of heuristics that interrupt a task when its execution time reaches the x-quantile of the distribution $\mathcal{D}_i$ with $0 \leq x \leq 1$.
- Finally, NeverInterrupt is the baseline heuristic that never interrupts tasks; more precisely, to avoid outliers, NeverInterrupt interrupts a task when its execution reaches 100 times the mean value of the distribution.

2.4.2 Processor selection heuristics

As we have different types and numbers of processors, we aim at finding the optimal subset to be enrolled. This is especially true when we only have the budget to ue a small subset of the processors until the deadline. In order to achieve this goal, we compare several methods for choosing processors. They differ in their criteria to order the processors and then greedily select the processors in that order. Each method comes in two versions: the *limited (ltd)* version enrolls the first $i^* - 1$ processors, where i^* is the smallest index such that $\sum_{i=1}^{i^*} c_i > K$; the refined version selects the best total yield when either using $i^* - 1$ processors, as in the limited version, or using i^* processors. This choice has for objective to show the improvement of the last step on results. Here are the three orderings:

- Expect$^{\text{ltd}}$ and Expect (computation-speed based methods): processors are sorted by non-decreasing expected value of computation time.

- COST$^{\text{ltd}}$ and COST (cost-per-time-unit based methods): processors are sorted by non-decreasing unit cost.
- GREEDY$^{\text{ltd}}$ and GREEDY (yield methods): processors are sorted by non-increasing yield. GREEDY is indeed the greedy algorithm of Section 2.3.3.

In addition, we assess the absolute performance of each method by comparing with FRACTIONAL, which is the yield achieved by the solution to the fractional problem FRACOPTHETERO (see Proposition 2). Indeed, the value of *Fractional* is an upper bound to the performance, which is not always tight; we use it as a reference.

2.4.3 Parameters

In the following experiments, all platforms are composed of processors from six processor types ($k = 6$). Without special description, each processor type has 10 processors. In other words, we have $M = 60$ and $m_j = 10$ for $1 \leq j \leq 10$ in default. While studying the impact of different number of processors, we vary m_j between values in $\{1, 2, 3, 5, 7, 10\}$. Each processor type is characterized by a unit cost and a distribution that determines the execution time of each task. Type j processors have average speed $s_j = 2^{j-1}$ (i.e., $s_j \in \{1, 2, 4, 8, 16, 32\}$). These values correspond to normalized speeds in realistic platforms such as Amazon EC2[1] or Google Cloud[2] and are correlated to the number of cores in the processors. By default, the unit cost of a processor is proportional to its average speed: $c_j = s_j$. But we will also study in one paragraph the cases where unit costs are increasing faster than average speeds, in which $c_j = 1.5s_j$ or $c_j = s_j^{1.5}$.

The second processor characteristic is the distribution of the execution times, which follow standard probability distributions. The heterogeneity of a scheduling problem instance has several meanings (for instance, both tasks and processors heterogeneity are studied in [15]). In our case, we consider the heterogeneity of the expectation and the heterogeneity of the variability. For all tested distributions, the expectation of execution times is fixed as the inverse of the processor speed, which determines the first type of heterogeneity. For the second type, we control the variation of the Coefficient of Variation (CV), which is defined as the ratio of standard deviation over expectation. Similar CVs for all processors lead to a low variability heterogeneity: execution times varies similarly on all processors. On the contrary, different CVs mean that execution times are closer to their expectations on some processors than on some others. For instance, two distributions with expectations 1 and 2 and the same CV 1 have expectation heterogeneity but no variability heterogeneity. This is the opposite with distributions both with expectation 1 and with CVs 1 and 2. This second type of heterogeneity is controlled by parameter x_{CV}. Of course for exponential and half-normal distributions, which have a single parameter, the standard deviation is given when choosing the mean, so this discussion only applies to the two-parameter distributions (lognormal, uniform, gamma, inverse-gamma and Weibull). Altogether, the expected value μ_j and standard deviation σ_j on m_j are set as follows: $\mu_j = \frac{1}{s_j}$, $\sigma_j = \mu_j U$ where the parameter U is drawn randomly and uniformly in the interval $[3 - x_{\text{CV}}, 3 + x_{\text{CV}}]$. We use $0 \leq x_{\text{CV}} \leq 3$ in the experiments.

Finally, we fix the budget[3] at $b = \beta \sum_{j=1}^{k} m_j c_j = \beta \times 630$, where $\beta \in \{0.01, 0.05, 0.1, 1, 2, 5, 8, 10\}$. For each budget value, we vary the deadline as $d = \beta 630^{\frac{i}{5}}$ (hence $K = 630^{1-\frac{i}{5}}$) for $0 \leq i \leq 5$. This leads to 6 deadline values following a geometric progression between two extreme cases

[1] https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls

[2] https://cloud.google.com/compute/pricing

[3] 630 represents the budget required to enroll all 60 processors for one time unit.

$d = b$ and $d = \beta$. The first case is when the deadline is sufficiently large to exhaust the entire budget by selecting any single processor. The second case is when the deadline is so tight that all processors must be used to exhaust the budget. Altogether, we have 8 budget and 6 deadline values, hence 48 configurations. For each configuration, each strategy is run 10 times on 100 randomly drawn platform instances (the mean of the distribution is fixed, and we draw the value of the standard deviation as discussed above, except for exponential and half-normal distributions).

The numbers of successes are reported in boxplots, each of which consists of a bold line for the median, a box for the quartiles, whiskers that extend at most to 1.5 times the interquartile range from the box and additional points for outliers. We start with a summary table covering all distributions, and then focus on lognormal distributions.

Table III: Performance ratio of all orderings over FRACTIONAL.

Ordering	Mean	Median	Q10%	Q90%
GREEDY	0.9977	0.9994	0.9668	1.0272
GREEDY$^{\text{ltd}}$	0.6047	0.8385	0	0.9998
COST	0.6943	0.9507	0.0522	1.0208
COST$^{\text{ltd}}$	0.5587	0.7634	0	1.0016
EXPECT	0.7766	0.9717	0.1556	1.0124
EXPECT$^{\text{ltd}}$	0.3642	0	0	0.9973

2.4.4 Result synthesis for all distributions

In Table III, OPTRATIO is chosen as cutting threshold heuristic on each processor. We use $b = 630$ (hence $\beta = 1$) and $x_{\text{CV}} = 3$. For each distribution, we have 6 values of d. For exponential and half-normal, the standard deviation is given when we select the mean, so we run only one platform configuration. For the other five probability distributions (lognomal, uniform, gamma, inverse-gamma and Weibull), we draw 100 platform configurations, as mentioned above. Each configuration is run 10 times, which leads to a total of 30,120 experiments. Now, for each heuristic, we proceed as follows: for each experiment, we record the ratio of the number of successful tasks achieved by the processor selection heuristics over the number of successful tasks achieved by FRACTIONAL; this leads to the statistical values reported in Table III: mean, median, 10% Quantile and 90% Quantile.

Table III shows that GREEDY performs very well overall. Its ratio is close to 1, not only for the mean value, but even for the 10% quantile. In other words, GREEDY has a performance close to that of FRACTIONAL; it also consistently outperforms all the other processors selection heuristics.

For each heuristic, the non-limited version is always much better than the limited one. Because limited versions readily discard each processor for which there is not enough budget to run until the deadline, they are at risk of wasting a big fraction of the budget and then produce a bad, even null result. Indeed, there is a large difference between the mean and median values for the limited versions, showing that there are many results close to 0. Results for the 90% quantile are good for all heuristics, and even larger than 1, while FRACTIONAL represents an

asymptotic upper bound. Here is the explanation: a sixth of experiments are for $d = 1$. In this case, all the heuristics enroll all the 60 processors to execute tasks. As the execution time of a task is randomly drawn, some heuristics can have a better result than FRACTIONAL.

2.4.5 Lognormal distribution

In this section, we focus on lognormal distributions and further study the impact of several parameters. A lognormal distribution is a natural candidate to model task execution times, because it has been advocated to model file sizes [25], and task costs could naturally obey this distribution too. Moreover, the lognormal distribution is positive, it has a tail that extends to infinity and the logarithm of the data values are normally distributed.

For a lognormal(α, β) distribution, the density function is $f(x) = \frac{e^{-\frac{(\log(x)-\alpha)^2}{2\beta^2}}}{x\beta\sqrt{2\pi}}$ for $x \in [0, \infty)$, the mean is $\mu = e^{\alpha+\frac{\beta^2}{2}}$ and the standard deviation is $\sigma = e^{\alpha+\frac{\beta^2}{2}}\sqrt{e^{\beta^2}-1}$. To ensure a given expected value μ and standard deviation σ, we set $\alpha = \log(\mu) - \log(\sigma^2/\mu^2+1)/2$ and $\beta = \sqrt{\log(\sigma^2/\mu^2+1)}$.

Cutting threshold heuristics

First, we compare the performance of the different cutting threshold heuristics in Figure 2.1. We report results for $b = 630$ and the 6 corresponding deadline values. We can find that, for lognormal distribution, OPTRATIO, QUANTILE (0.1) and QUANTILE (0.2) have usually better results than others. This is because the threshold calculated by OPTRATIO is usually between 10^{-1} and 10^{-3} in our case and the threshold provided by QUANTILE (0.1) and QUANTILE (0.2) is closer to this value than other heuristics. The results confirm the observations made with homogeneous processors [12]: OPTRATIO achieves the best success rate, and is significantly better than the baseline NEVERINTERRUPT. This leads to choose OPTRATIO as the cutting threshold heuristic in all the following experiments.

Varying budget and deadline values

Figure 2.2 reports results for the 48 (b, d) pairs. We make several observations.

First, when K is fixed, multiplying b and d by a value $\beta > 1$ only changes the absolute value of the result (there is a proportional relationship between β and the number of successful tasks), but the global outcome remains the same: the same processors are chosen, and the ratio of the results of each heuristic over FRACTIONAL is not modified. This shows that, for a lognormal distribution with μ and σ chosen as in our experiment, $\beta = 1$ is enough to simulate a problem instance with large b and d values. In the following experiment, we keep $b = 630$ and vary x_{CV}.

Second, in Figure 2.2, we see the impact of the deadline constraint by varying both d and K with a constant b (in each column of the figure). With the extreme case when K is large (i.e., $K = \sum_i c_i$), all methods select all processors, which exhausts the budget when reaching the deadline. The alignment of all boxplot values in the figure for $d = \beta$ confirms this effect. Moreover, all methods choose processors in a predefined order until the sum of c_i of selected processors reaches K, which means we must choose more processors with large values of K. As processors are ordered by their yield in the FRACTIONAL method, the larger the value of K, the smaller the average yield of chosen processors. However, with larger deadlines, the processor choice becomes critical and only GREEDY has a gain similar to FRACTIONAL. In other words, the difference between these two latter methods and the other ones increases as the parallelism

K decreases. As the deadline is less constrained, GREEDY can select only the processors with best yield. With $K \leq 13.2$, the gain is less remarkable because the best processors are all already enrolled. Only small deadlines impose the selection of inefficient processors to exhaust the budget before the deadline. Thus, having larger deadlines provides little benefit.

Third, in all instances, GREEDY, COST and EXPECT are respectively better than or similar to GREEDY$^{\text{ltd}}$, COST$^{\text{ltd}}$ and EXPECT$^{\text{ltd}}$. These last three methods even have some zero values. As explained in Section 2.4.4, this is because these methods enroll processors (in different orders) until the last processor that does not exceed the budget when executed up to the deadline d. Thus, the first processor is abandoned if the budget to execute this processor exceeds b. In this scenario, no processor is chosen by the method, and the number of successful tasks is zero.

Fourth, we observe that GREEDY remains the most efficient resource selection heuristic even for small values of the budget (when $\beta < 1$). This is good news, because we had proven the asymptotic efficiency of GREEDY but needed to check that it maintained its superiority for the whole range of budget and deadline values (even though the number of successes is no longer proportional to the budget for smaller values).

Impact of variability heterogeneity

Figure 2.3 demonstrates the dependence between the level of variability heterogeneity controlled by x_{CV} and the performance disparity between the resource selection heuristics. When $x_{\text{CV}} = 0$, all processors have the same yield as they have the same ratio CV, thus all heuristics are similar. As x_{CV} increases, the difference between FRACTIONAL and all other methods except GREEDY expands up to a factor three for the median performance. Note that the maximum number of successful tasks increases with x_{CV}, especially for FRACTIONAL and GREEDY heuristics, because the methods manage to select processors with the best yield. In particular, it is possible to perform twice as much tasks with $x_{\text{CV}} = 2.5$ than with $x_{\text{CV}} = 0$ because some processors in a platform configuration can have a higher yield.

Impact of the number of processors

Figure 2.4 presents the variation of performance when varying the number of processors for each task type m_j. We can find that, while fixing the budget, the performance of all heuristics increases with m_j (and M). The difference between heuristics also increases with the number of processors, and GREEDY has always a performance very close to FRACTIONAL.

Impact of unit cost

Figures 2.5, 2.6 and 2.7 show the performance of all heuristics when the unit cost is respectively defined as following: $c_j = s_j$, $c_j = 1.5s_j$ and $c_j = s_j^{1.5}$. In order to make the different sets comparable, we increase the budget respectively to $b = 1.5 * 630 = 945$ and $b = 630^{1.5} \approx 2795$ for the latter two experimental sets. The results in these figures confirm that GREEDY performs very closely to FRACTIONAL, and in the cases of $c_j = 1.5s_j$ and $c_j = s_j^{1.5}$, the heuristics have the same variation as $c_j = s_j$ while varying K and x_{CV}. This proves the robustness of our heuristic.

2.4.6 Summary

All the above results confirm that GREEDY reaches better performance than the other resource selection heuristics, up to a factor three on average.

2.5 Conclusion

In this chapter, we have dealt with the problem of scheduling stochastic tasks on a cloud platform under both deadline and budget constraints. On each enrolled processor, we use several cutting threshold heuristics to decide when to interrupt tasks. The main difficulty is to select the best subset of processors so as to maximize the expected number of tasks that are successfully executed. We have assessed the complexity of this resource selection optimization problem, showing that it is NP-hard, and also designing GREEDY, a greedy algorithm whose performance is proved to be a 2-approximation. GREEDY sorts the processors by non-decreasing yield and then determines the optimal number of processors to enroll when considering them in this order. On the practical side, we have conducted an extensive set of experiments, with several standard probability distributions for task execution times. These experiments show that, (i) as in the homogeneous case, OPTRATIO has the best performance within all cutting threshold heuristics in the heterogeneous case, and (ii) GREEDY significantly outperforms other approaches based on simple heuristics, and reaches an absolute performance close to the upper bound established in the chapter.

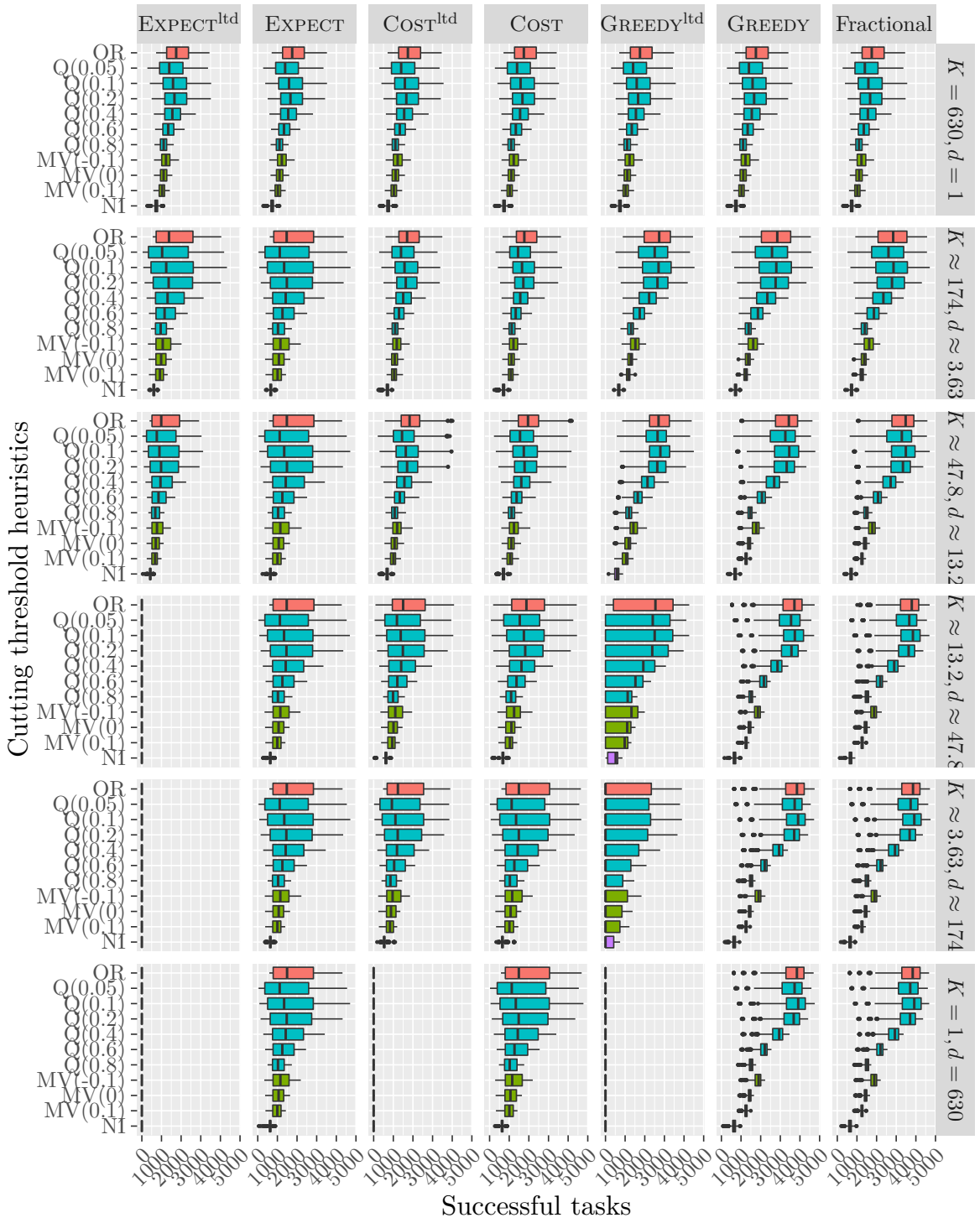


Figure 2.1: Number of successfully executed tasks for different resource selection and cutting threshold heuristics, with $m_j = 10$, $M = 60$, $c_j = s_j$, $b = 630$. Execution times follow a lognormal distribution with $x_{\text{CV}} = 3$.

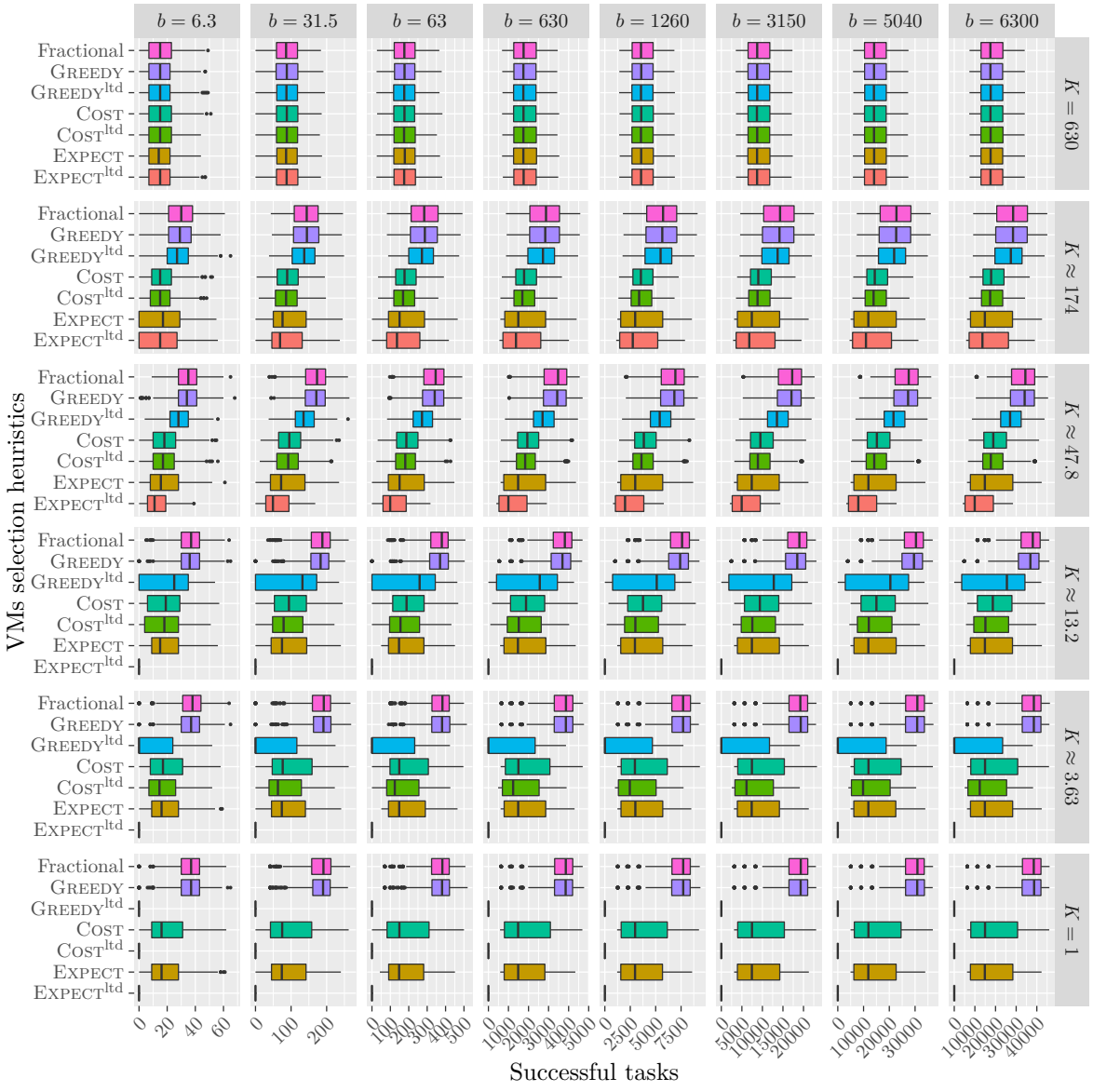


Figure 2.2: Number of successfully executed tasks for resource selection heuristics with OptRatio, $m_j = 10$, $M = 60$, $c_j = s_j$. Execution times follow a lognormal distribution with $x_{\text{CV}} = 3$.

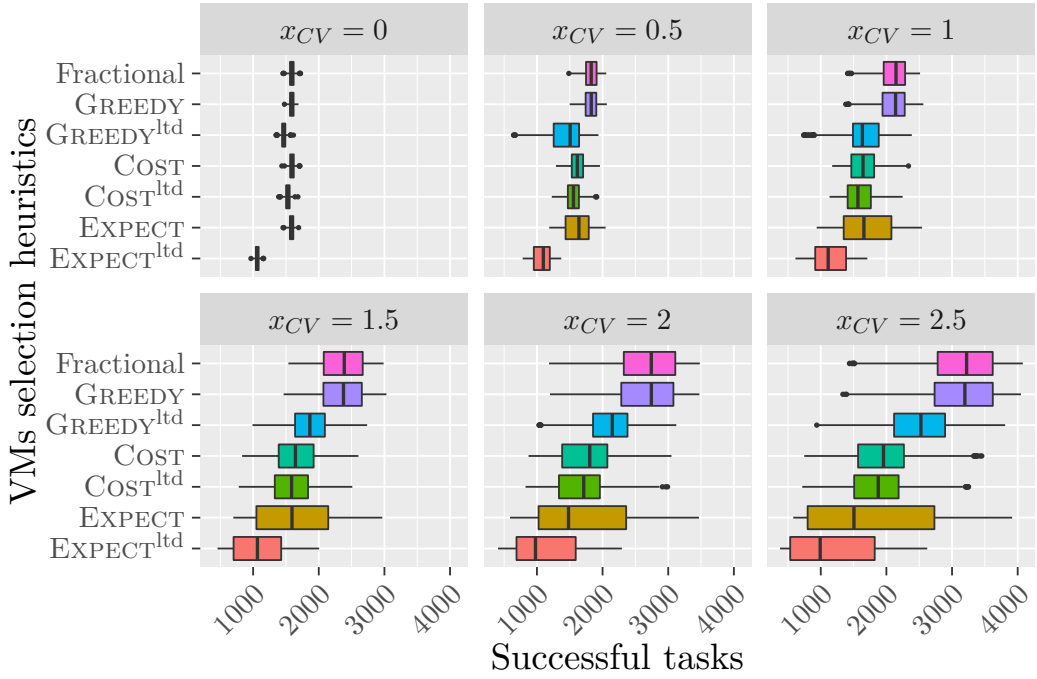


Figure 2.3: Number of successfully executed tasks for resource selection heuristics with OptRatio, $m_j = 10$, $M = 60$, $c_j = s_j$, $b = 630$ and $d \approx 13.2$. Execution times follow a lognormal distribution with x_{CV} varying.

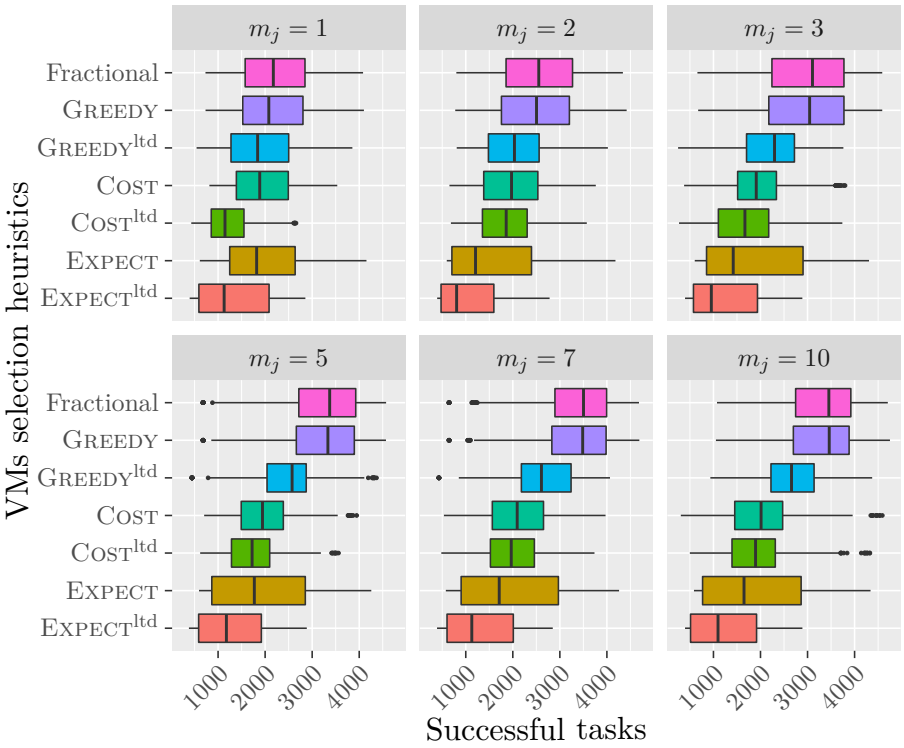


Figure 2.4: Number of successfully executed tasks of different methods to choose VMs when interrupting tasks with heuristic OPTRATIO. Each of the 100 platforms is generated with $m_j = 1, 2, 3, 5, 7, 10$, $M = 6, 12, 18, 30, 42, 60$, $c_j = s_j$ and is used 10 times with $b = 630$ and $d \approx 13.2$ ($K \approx 47.8$). Execution times follow lognormal distributions with $x_{\text{CV}} = 3$.

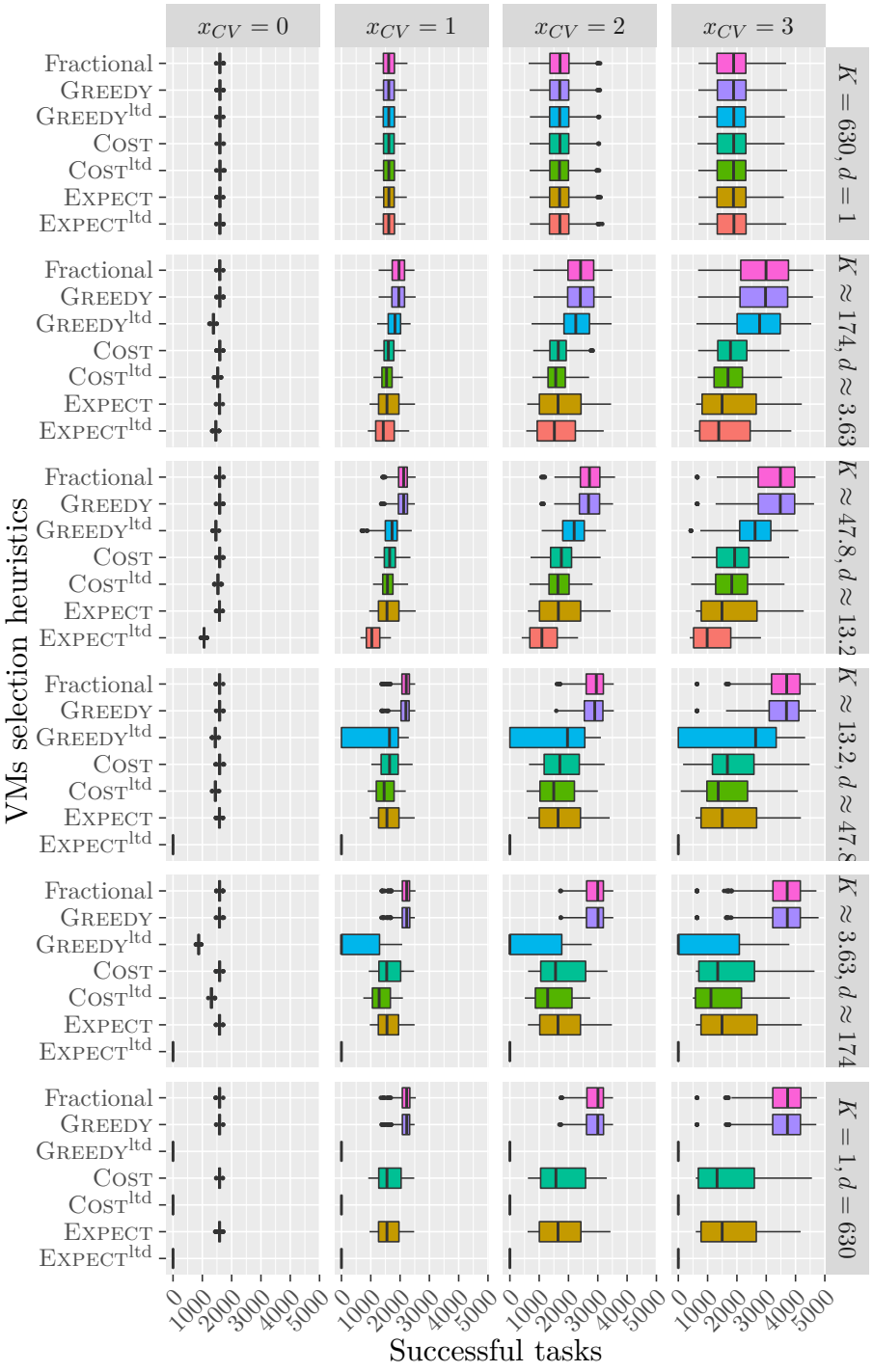


Figure 2.5: Number of successfully executed tasks of different methods to choose VMs when interrupting tasks with heuristic OptRatio. Platforms are generated with $m_j = 10$, $M = 60$, $c_j = s_j$, $b = 630$. The values for d and K follows a geometric progression between 1 and $b = 630$. Execution times follow a lognormal distribution.

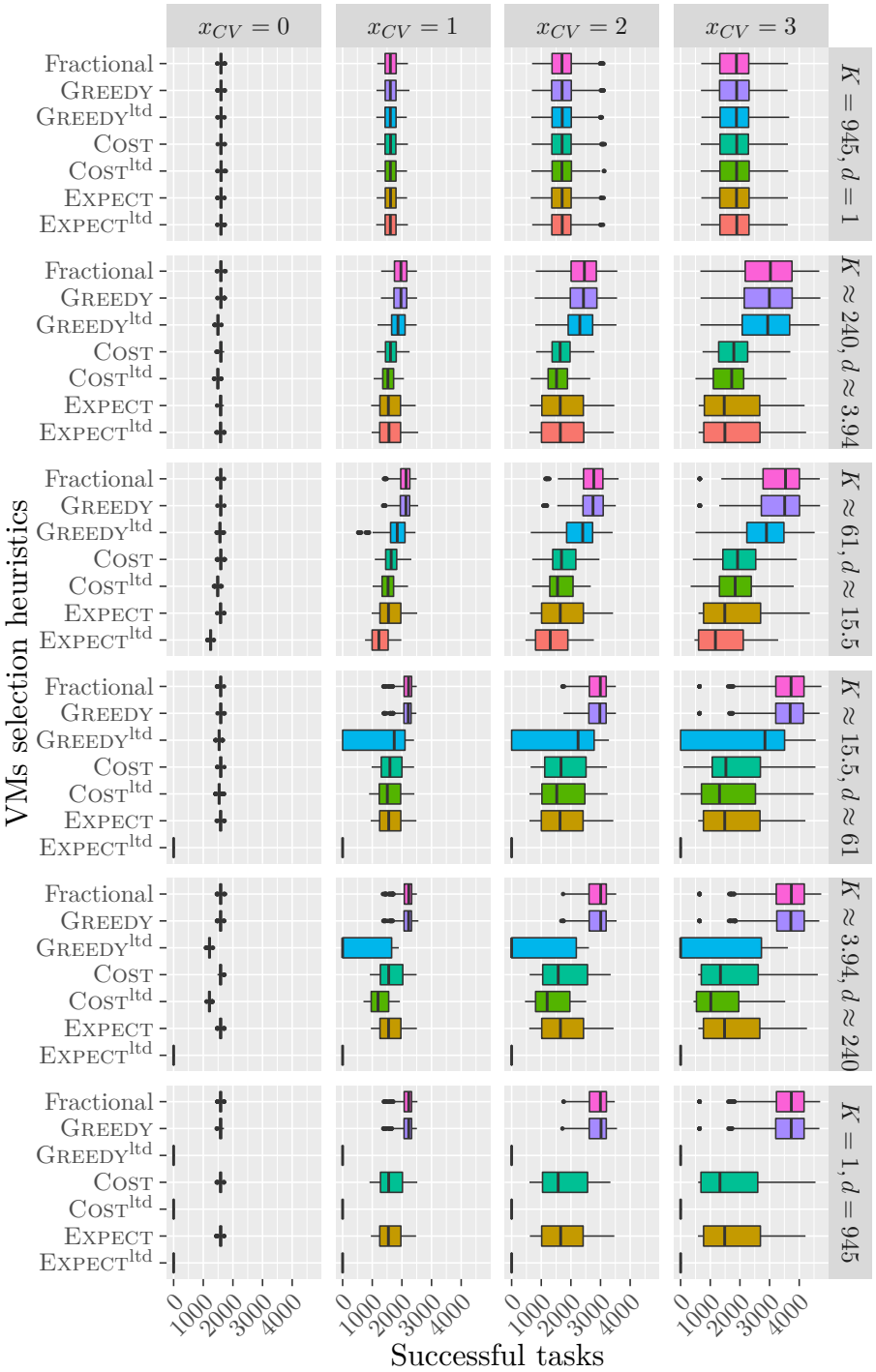


Figure 2.6: Number of successfully executed tasks of different methods to choose VMs when interrupting tasks with heuristic OptRatio. Platforms are generated with $m_j = 10$, $M = 60$, $c_j = 1.5s_j$ and $b = 945$. The values for d and K follows a geometric progression between 1 and $b = 945$. Execution times follow a lognormal distribution.

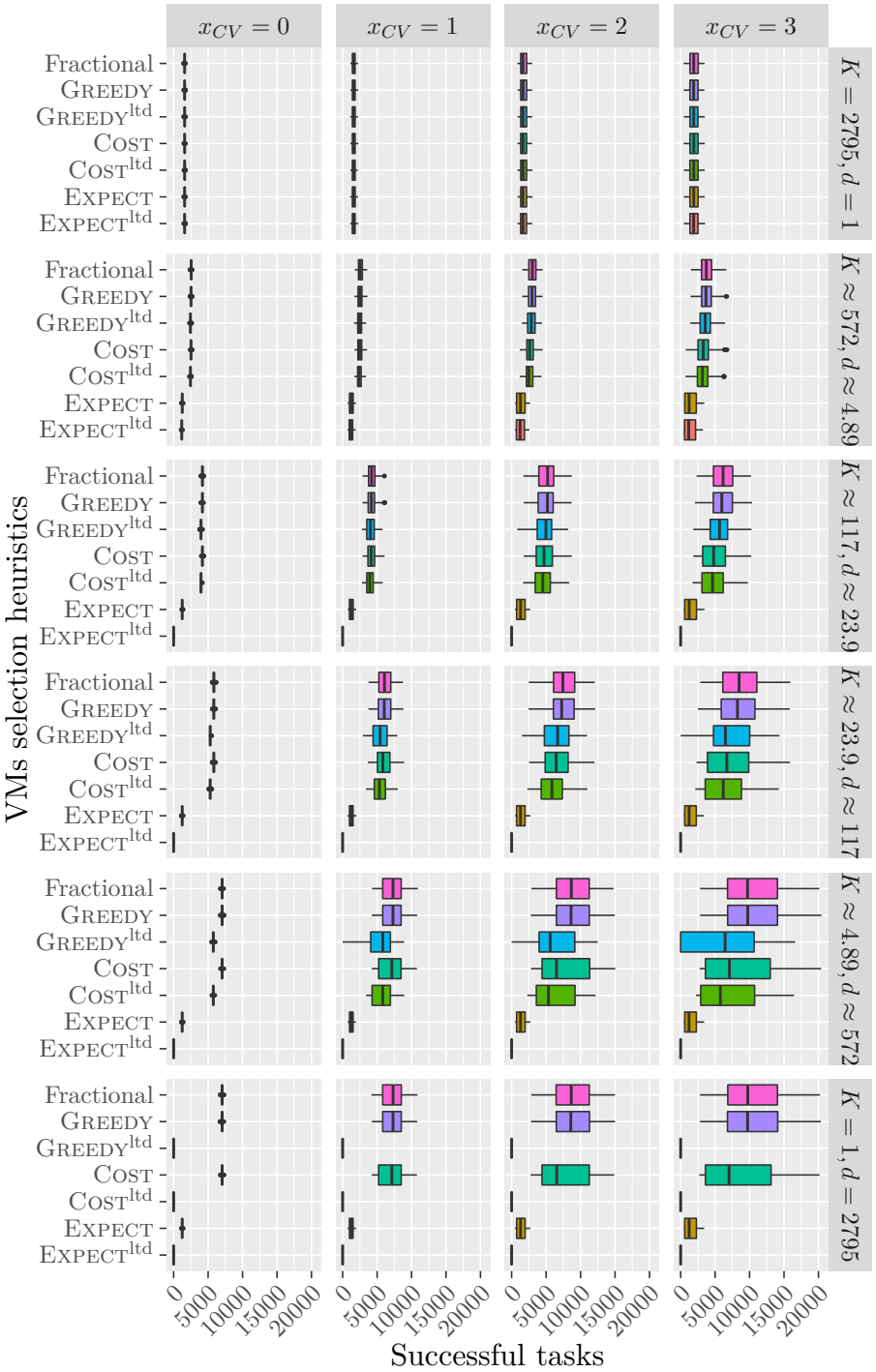


Figure 2.7: Number of successfully executed tasks of different methods to choose VMs when interrupting tasks with heuristic OptRatio. Platforms are generated with $m_j = 10$, $M = 60$, $c_j = s_j^{1.5}$ and $b \approx 2795$. The values for d and K follows a geometric progression between 1 and $b \approx 2795$. Execution times follow a lognormal distribution.

Chapter 3

Scheduling stochastic tasks with unknown probability distribution

3.1 Introduction

Similarly as the work in Chapter 2, this chapter builds also upon the previous work introduced in Section 1.1 where we tackle the dramatically simpler problem where the distribution $\mathcal{D}$ is known. In that case, we proposed an analytical method to compute the optimal threshold l. The main focus of this chapter is to investigate efficient strategies when the distribution $\mathcal{D}$ is unknown.

The only information known is that the task execution times are independent and identically distributed (IID) random variables obeying the same probability distribution, but this distribution is unknown. Similarly to the previous work, the scheduler has both a deadline constraint d and a budget constraint b. At any time, and on each enrolled processor, the scheduler can decide whether to interrupt a long-running task T to start a new task T'.

In this non-clairvoyant setting, what is the optimal strategy? Intuitively, the scheduler must first decide how many processors to enroll. Then, the scheduler needs to acquire some information about task execution times by letting several tasks run until completion on each processor. At some point, the scheduler synthesizes the information acquired so far and will decide for a scheduling policy. This policy could be either to allow all tasks to run until completion, or to define a cutting threshold l after which every long-running task should be interrupted. The cutting threshold l can be recomputed dynamically as the execution progresses until the deadline d is reached or the budget b is exhausted, whichever comes first. Each of the above decisions involves a complicated trade-off. The main problem is to determine when and how to compute a first cutting threshold l (with the possibility that $l = +\infty$, meaning that all tasks are allowed to run until completion). Again, there is a trade-off. Deciding for the threshold early can lead to an imprecise estimation because it is based on little information, but this would avoid to consume a significant fraction of the deadline and of the budget before interrupting any task. On the contrary, deciding for the threshold later during the execution leads to making a more accurate decision, at the risk of having wasted resources unduly. Altogether, these are several complicated trade-offs to achieve. The key is to be able to compute a good threshold without bias, and this chapter introduces several strategies to determine a good threshold, and at the right moment in the execution.

As already mentioned in Chapters 1 and 2, the problem is very closely related to imprecise computations, where tasks are divided into a mandatory and an optional part. Our work

perfectly corresponds to the optimization of the processing of the optional parts. However, the probability distribution $\mathcal{D}$ of task execution times is unknown before processing in this work, and can be only determined through sampling many tasks. Unfortunately, in our scheduling problem, letting the scheduler sample many tasks without interruption to learn, say, the mean and standard deviation of the distribution, can prove very costly: it will consume a significant part of the budget and will prove suboptimal for any distribution requiring a small cutting threshold l, such as lognormal distributions.

To the best of our knowledge, this work constitutes the first attempt to address this challenging problem. The major contributions of this work are the following:

- We design a set of scheduling heuristics that use different estimators of the cutting threshold l, and that refine this estimation periodically as the execution progresses.
- We show how to use to the Kaplan-Meier estimator [54] to account for long-running tasks when estimating the threshold l.
- We introduce several methods for deciding when to recompute the threshold.
- We report a comprehensive set of simulation results that compare the heuristics for various budget and deadline values, using up to 14 different probability distributions.

The rest of the chapter is organized as follows. We detail the framework and objective in Section 3.2. We provide scheduling heuristics for the unknown distribution in Sections 3.3 and 3.4: Section 3.3 is devoted to methods for computing the cutting threshold accurately, while Section 3.4 focuses on when to recompute it. We compare the heuristics in Section 3.5, assessing their performance for an extensive set of simulation parameters. Finally, we provide concluding remarks in Section 3.6.

3.2 Problem definition

We consider a cloud platform composed of identical processors. The execution time of a task on a processor obeys an unknown probability distribution $\mathcal{D}$. Without loss of generality, we assume it costs one budget unit to execute a task for one second on any processor, and we have a total budget b and an overall deadline d. As we are on the homogeneous platform in this work, as mentioned in the previous work 1.1, the number of processors used can be simply calculated with $M = \left\lceil \frac{b}{d} \right\rceil$. Therefore, in this chapter, we use the number of processors M, instead of K in Chapter 2, to express the ratio of budget to deadline. When considering the asymptotic behavior of policies, we assume that budget b and deadline d grow toward infinity. Main notations are summarized in Table I. We assume executions to be non-preemptive: if the execution of a task is interrupted, all the work done (and the budget spent) so far for that task is lost. Our objective is to maximize the total number of tasks successfully completed under the budget and deadline limits. To drive the design of our scheduling policies, we use an instantaneous version of this objective, namely the yield ($\mathcal{Y}$), which is defined as the expected number of tasks completed per unit of budget spent. This is equal to the expected success rate per second ($\mathcal{E}$), as we spend one budget unit per second.

All our scheduling policies are required to have polynomial complexity. Since a solution to the problem is the list of the tasks that are executed, either partially or successfully (for each of these tasks, the scheduler made a decision), the size of the problem is proportional to that number of tasks. This number in turn is proportional to the budget (or deadline), divided by the expectation of the (unknown) probability distribution $\mathcal{D}$, since the average execution time until completion of a task is $\mu(\mathcal{D})$. Furthermore, the scheduling policies will make decisions

Table I: Summary of notations.

b	budget
d	deadline
M	number of processors used
$\mathcal{D}$	probability distribution of task execution times
μ, σ	mean, standard deviation of $\mathcal{D}$
$\mathcal{Y}$	expected number of tasks completed per unit of budget spent
$\mathcal{E}$	expected success rate per second

and compute a cutting threshold several times during the whole execution; we require that the number of such decisions be constant, and they will typically be taken each time a prescribed percentage of the budget is spent. The motivation here is to cap the overhead incurred by the scheduler by forbidding to recompute a threshold at each execution of a new task.

3.3 Threshold estimation for unknown distributions

We have seen in section 1.1 that when the distribution of task execution times is known, the optimal policy is a fixed-threshold strategy that interrupts tasks, and that the choice of the cutting threshold can have a very significant impact on the system performance. Now the question is: how do we find the optimal cutting threshold when the distribution is unknown?

In order to acquire information on the distribution of task execution times, the single option is to execute some tasks and record their execution times. We will consider the problem of deciding how many tasks to execute in Section 3.4. For the sake of the argument, let us assume that we have already launched the execution of several tasks, that some executions have already completed, some are still running, and some were interrupted. For instance, in the toy example presented on Figure 3.1 we have two processors, four tasks, and we want to take a decision at time 20. One task has executed for 5 seconds, one for 16; two tasks have not yet completed (the tasks in red), having run, respectively, for 15 and 4 seconds so far. The question is then: how do we estimate the distribution of task execution times based on this data?



Figure 3.1: Toy example with two processors, two successfully completed tasks (in blue) and two not-yet-completed tasks (in red) at time 20.

There are two types of approaches. In the first type, we would try to guess some characteristics of the distribution. For instance, we could claim that "task execution times likely follow an exponential distribution". Then, we would look for the exponential distribution that better fits the data, for instance using a maximum likelihood estimation. If our initial guess was lucky, we should end up with a good result. However, the underlying distribution may be either a lognormal distribution, or a multimodal one, or even not resemble any of the most

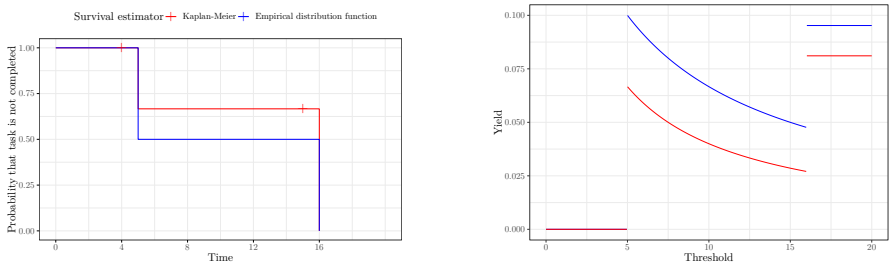


Figure 3.2: Probability of survival (left) and yield (right) for the toy example of Figure 3.1 when using the empirical distribution function (blue) or the Kaplan-Meier estimator (red).

used probability distributions. Rather than relying on potentially unlucky guesses, we aim at designing a robust approach which would deliver high quality results, regardless of the underlying distribution. Therefore, our approach belongs to the second type of approaches, sometimes called "nonparametric" statistics. We are not going to make any assumption on the underlying distribution. Section 3.3.1 details a naive approach that only considers the execution times of tasks that have completed. This approach has the advantage of simplicity. However, as exemplified by the toy example on Figure 3.1, it can ignore a significant share of the data, and in particular long-running tasks. The question on how to take into account tasks that have not yet completed has been thoroughly research in the field of . . . medical research! In Section 3.3.2, we show that our problem is exactly the statistical medical problem known as *survival analysis with right-censored data*, even if the concepts and wordings are quite different. We also show how to use its classical solution, the Kaplan-Meier estimator, to solve our problem more accurately.

3.3.1 The empirical distribution function

The naive approach only considers the execution times of completed tasks and uses the associated *empirical distribution function* [90], along with the case of known probability. Consider an example where there are k different task execution times, $w_1 < w_2 < ... < w_k$, and where n_i tasks have the execution time w_i. Then, using the empirical distribution function, a task has an execution time w_i with probability $p_i = \frac{n_i}{\sum_{j=1}^{k} n_i}$. Using these probabilities, we search in the set $\{w_1, ..., w_k\}$ the value maximizing the yield, using Equation 1.1.

The main advantage of this approach is its simplicity. The toy example on Figure 3.1 illustrates its main drawback: there maybe many tasks whose information is ignored, namely the tasks that have not yet completed. This drawback induces a bias by ignoring long-running tasks.

3.3.2 Survival analysis and the Kaplan-Meier estimator

In medical research, biostatisticians have to answer questions like: "What is the probability that a patient will still be alive 5 years after receiving a cancer diagnosis?" To answer such a question, biostatisticians analyse the data of many individual patients. Some of these data will be complete: they will have both the time of diagnosis and the time of death of the patient. However, at the time of the analysis, some patients enrolled in the dataset will still be alive. The status of some other patients may be unknown because contact with them has been lost (e.g., they have moved away). In both cases observations are incomplete. One only knows the

time of diagnosis and the last time the patient was known to be alive. Hence, one only knows a lower bound on the time the patient has survived after the diagnosis. These incomplete "lower-bound" data are called *right-censored data* and the question addressed by biostatisticians is that of *survival analysis with right-censored data*. This problem is exactly ours, only the vocabulary changes:

- instead of survival times, we have execution times;
- instead of diagnosis times, we have start times;
- at the time of analysis, instead of patients still alive, we have tasks still running;
- at the time of analysis, instead of patients with unknown whereabouts, we have tasks that have been terminated by the scheduler before completion.

We can therefore use the tool to solve survival analysis with right-censored data, that is the Kaplan-Meier estimator [1, 54]. Nowadays, survival analysis with the Kaplan-Meier estimator is widely used in biostatistics [17, 40, 99], and in a variety of other domains such as engineering [81], economics [64], etc. We refer the interested reader to [1] for a thorough overview of survival analysis.

Consider an example where there are k different task execution times, $w_1 < w_2 < ... < w_k$. Here, execution times can be the execution times of tasks that have completed, like the values 5 and 16 in the example of Figure 3.1. They can also be censored execution times, like the values 4 and 15 in that example. Let d_i be the number of tasks that *die* at time w_i, that is, the number of tasks whose execution time is exactly w_i. Let r_i be the number of individual *at risks* just prior to time w_i, that is, the number of tasks whose execution time is greater than or equal to w_i. The survival function, $\mathcal{S}(t)$, is the probability that life is longer than t: $\mathcal{S}(t) = Pr(X > t)$. The Kaplan-Meier estimator gives us:

$$\mathcal{S}(t) = \prod_{w_i \leq t} \left(1 - \frac{d_i}{r_i}\right). \tag{3.1}$$

Using this estimator, we can then rewrite Equation 1.1 as:

$$\mathcal{Y}(t) = \frac{1 - \mathcal{S}(t)}{\sum_{j=1}^{\mathbb{I}(t)} (\mathcal{S}(w_{i-1}) - \mathcal{S}(w_i)) w_j + \mathcal{S}(w_{\mathbb{I}(t)}) t}$$

where $\mathbb{I}(t)$ is the index of the largest task execution time smaller than or equal to t: $\mathbb{I}(t) = k$ if $t \geq w_k$, and $w_{\mathbb{I}(t)} \leq t < w_{\mathbb{I}(t)+1}$ otherwise (with $w_0 = 0$ and $\mathcal{S}(w_0) = 1$).

We illustrate this estimator with the toy example of Figure 3.1:

w_i	d_i	r_i	$1 - \frac{d_i}{r_i}$	$\prod_{j \leq i} \left(1 - \frac{d_j}{r_j}\right)$
4	0	4	1	1
5	1	3	$\frac{2}{3}$	$\frac{2}{3}$
15	0	2	1	$\frac{2}{3}$
16	1	1	0	0

The resulting function is presented in red on the left-hand side of Figure 3.2, alongside the probabilities associated to the empirical distribution function (in blue). Red ticks indicate the presence of censored data. For the empirical distribution function, the probability that the execution time of a task exceeds 5 seconds is 50%, while it is 66.6% for the Kaplan-Meier estimator. When we plug these different probability functions in Equation 1.1, we obtain

the yields depicted on the right-hand side of Figure 3.2. In this toy example, the empirical distribution function claims that the optimal cutting threshold is 5, when the survival analysis claims that it is 16.

Note that, in the product of Equation (3.1), only the times corresponding to actual (non-censored) execution times matter. Execution times that only correspond to censored times each contribute a value of 1 in the product (see the table above). Note also that if there is no censored data, we have $r_{i-1} - r_i = d_{i-1}$ and $\mathcal{S}(t)$ simplifies into

$$\mathcal{S}(t) = \prod_{w_i \leq t} \frac{r_i - d_i}{r_i} = \prod_{w_i \leq t} \frac{r_{i+1}}{r_i} = \frac{r_j}{r_0}$$

where j is the smallest index such that $w_j > t$. In other words, when there is no censored data, the empirical distribution function and the Kaplan-Meier estimator coincide.

We can use the survival function to compute the mean and variance of the execution times. Recall that $\mathcal{S}(t) = Pr(X > t)$. Hence, $Pr(X = w_j) = Pr(X \in]w_{j-1}, w_j]) = \mathcal{S}(w_{j-1}) - \mathcal{S}(w_j))$ for $1 \leq j \leq k$ (with $w_0 = 0$ and $\mathcal{S}(w_0) = 1$, as stated above). We derive that:

$$\mu = \mathbb{E}[X] = \sum_{j=1}^{k} (\mathcal{S}(w_{j-1}) - \mathcal{S}(w_j))w_j + \mathcal{S}(w_k)w_k.$$

$$\begin{aligned} \sigma^2 &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 \\ &= \sum_{j=1}^{k} (\mathcal{S}(w_{j-1}) - \mathcal{S}(w_j))w_j^2 + \mathcal{S}(w_k)w_k^2 - \mu^2 \end{aligned}$$

3.4 Taking decisions

In Section 3.3, we have shown how we can use data from the execution of tasks to define the best cutting threshold. In this section we focus on how and when to acquire the data needed to compute a cutting threshold, possibly many different times as the execution progresses.

In order to acquire information on the distribution of task execution times, the only solution is to execute some tasks and to record their execution times. In this process, we have to make a classical trade-off. On the one hand, we should execute a sufficiently large number of tasks until completion, in order to be sure that the set of observed execution times is indeed representative of the underlying distribution. On the other hand, we should execute as few tasks as possible before making a decision, to avoid wasting a significant share of the budget on running tasks until completion if the optimal threshold is a "short" one. We start by designing policies that try to guess the good tradeoff before launching any task. Then, we present a policy that tries to automatically infer that tradeoff.

3.4.1 *One-size-fits-all* policies

The simplest strategies will try to guess, without interrupting any task, the "right" tradeoff. Consider a strategy that spends 10% of the overall budget running tasks up to completion before computing the optimal threshold: it can still hope to achieve a 90% overall efficiency. Indeed, it can achieve such a good performance just by applying the optimal policy for the actual distribution of task execution times during the remaining 90% of the budget. As this looks promising, this is the basis of our first two strategies:

1. we pick a priori a percentage p;
2. we run tasks on processors until we have spent the fraction $p \times b$ of the overall budget;
3. we compute the cutting threshold either using the empirical distribution function for strategy EMPIRICAL, or survival analysis for strategy SURVIVAL;
4. we then apply the cutting threshold on all tasks until the budget is exhausted.

For 2), recall that we enroll $\left\lceil \frac{b}{d} \right\rceil$ processors. Hence, up to rounding artefacts, the fraction pb of the whole budget is spent when the fraction pd of the deadline is reached on each processor.

When the task average execution time is large and the observation budget pb is small, it may happen that no task has completed when the observation budget is exhausted. In such a case, we delay the computation of the threshold to after having spent $2pb$, and so on if this extended budget is also too small.

There are two obvious limitations to these first two strategies. First, once a threshold is computed, it is applied until the end. However, in the meantime, new tasks complete and some are interrupted, and we gather more information on the distribution. We should take the new available information into account. We propose to do that periodically, each time we have spent another fraction pb of the budget, by recomputing the threshold considering all the available data. This gives us two new strategies PERMPIRICAL and PERSURVIVAL.

The second limitation is due to the fact that when we compute the threshold, we have no idea how much the accumulated data is representative of the actual distribution of execution times. Therefore, we have no idea of the quality of the threshold that we compute. To remedy this, in the next section, we propose to automatically infer when to stop observing the distribution and to compute the threshold.

3.4.2 Automatic inference

We do not want to compute the threshold before ascertaining that the data we have acquired on the distribution of task execution times is "good enough". However, we do not want to spend the whole budget trying to acquire information. Hence we decide to rely on two parameters fixed a priori: a percentage p_{max} of the overall budget and a precision ϵ. The precision ϵ will guarantee that we have a good enough approximation of the data distribution because the mean value and standard deviation of the empirical distribution function have converged (up to the precision ϵ). In addition, the percentage p_{max} will be a large value guaranteeing that in extreme cases, we will eventually take a decision before running out of the budget. We will compute the threshold as soon as one of the two following conditions is met: either observing convergence of the empirical distribution function, or having spent a fraction $p_{max}\ b$ of the overall budget. In practice, each time a task completes, we recompute the mean value and standard deviation of the distribution. If both new values have a relative difference less than ϵ from previous ones, we assume the approximation of the distribution to have converged.

Once we have computed a cutting threshold, say after having spent a budget qb, we recompute it periodically each time we have spent $\max\{0.01, q\}b$ of the budget. We add the max for the cases where the budget is very large and the convergence very fast, in order to keep the number of decisions constant (as stated in Section 3.2). Obviously the new strategy can be implemented for both the empirical distribution function and the survival analysis. However, because of the superiority of the survival analysis (shown in Section 3.5), we implement it only for survival analysis, leading to the new strategy AUTOPERSURVIVAL.

3.5 Experiments

This section assesses the performance of the different strategies introduced in the previous sections. The experimental settings are detailed in Section 3.5.1, and results are presented in Section 3.5.2. All strategies were implemented in R. The corresponding source code, and all the data, are publicly available in [32].

3.5.1 Experimental methodology

The default settings are as follows. The deadline d can take the values 5, 10, 50, and 100. The cloud platform is composed of $M = 10$ identical processors, each with a unitary cost. As stated in Section 3.2, the budget b is defined as $b = Md$. Then, the budget b is evenly shared among the processors which all execute tasks until the deadline d. As discussed in Section 1.1, recall that a typical configuration enrolls $\left\lceil \frac{b}{d} \right\rceil$ processors.

We use different standard probability distribution functions to generate task execution times, namely uniform, exponential, log-normal, half-normal, truncated normal (truncated on $[0, +\infty)$), gamma, inverse-gamma, and Weibull distributions. In addition, multimodal distributions have been advocated to model jobs, file and object sizes [25]. Therefore, we also consider two types of bimodal distributions, either based on truncated normal distributions or on exponential distributions. For all the bimodal distributions, the two modes are equiprobable. For four of the distributions, we consider two different sets of parameters to illustrate different potential behaviors associated to the same type of distribution. These distributions are the gamma distribution, the log-normal, the bimodal exponential, and the bimodal truncated normal. To enable a direct comparison between all different distributions, we choose their parameters so that all distributions achieve a mean equal to 1. The detailed parameters of the distributions are presented in Table II.

In objective to avoid arbitrarily small task execution times, we add a constant $\delta = 0.05$ to all randomly generated task execution times. Therefore, for all the distributions under study, execution times will always have an average value of 1.05.

For each simulation setting, we generate 1000 random instances (i.e., sets of task execution times). In addition, we compare the result of the proposed strategies with two reference heuristics. NeverInterrupt is the baseline heuristic which let all tasks run up to completion. Oracle knows in advance the distribution used to generate task execution times and computes the optimal threshold using that knowledge. Oracle is thus an upper bound on the performance of any strategy. Therefore, the closer to Oracle's performance, the better the heuristic.

3.5.2 Results

We present the experimental results in two steps. At first, we present results for the *one-size-fits-all* heuristics: Empirical, Survival, PerEmpirical and PerSurvival, for all the distributions. PerSurvival turns out to be the best of the 4 heuristics in almost all studied cases. This is why, after that, we compare PerSurvival and AutoPerSurvival.

Before assessing the performance of the heuristics, we consider the distributions under study. Figure 3.3 presents, for each distribution, the theoretical yield achievable as a function of the cutting threshold. In Figure 3.3, we have ordered the distributions by non-increasing values of their cutting threshold. One can see that different distributions, or the same distribution with

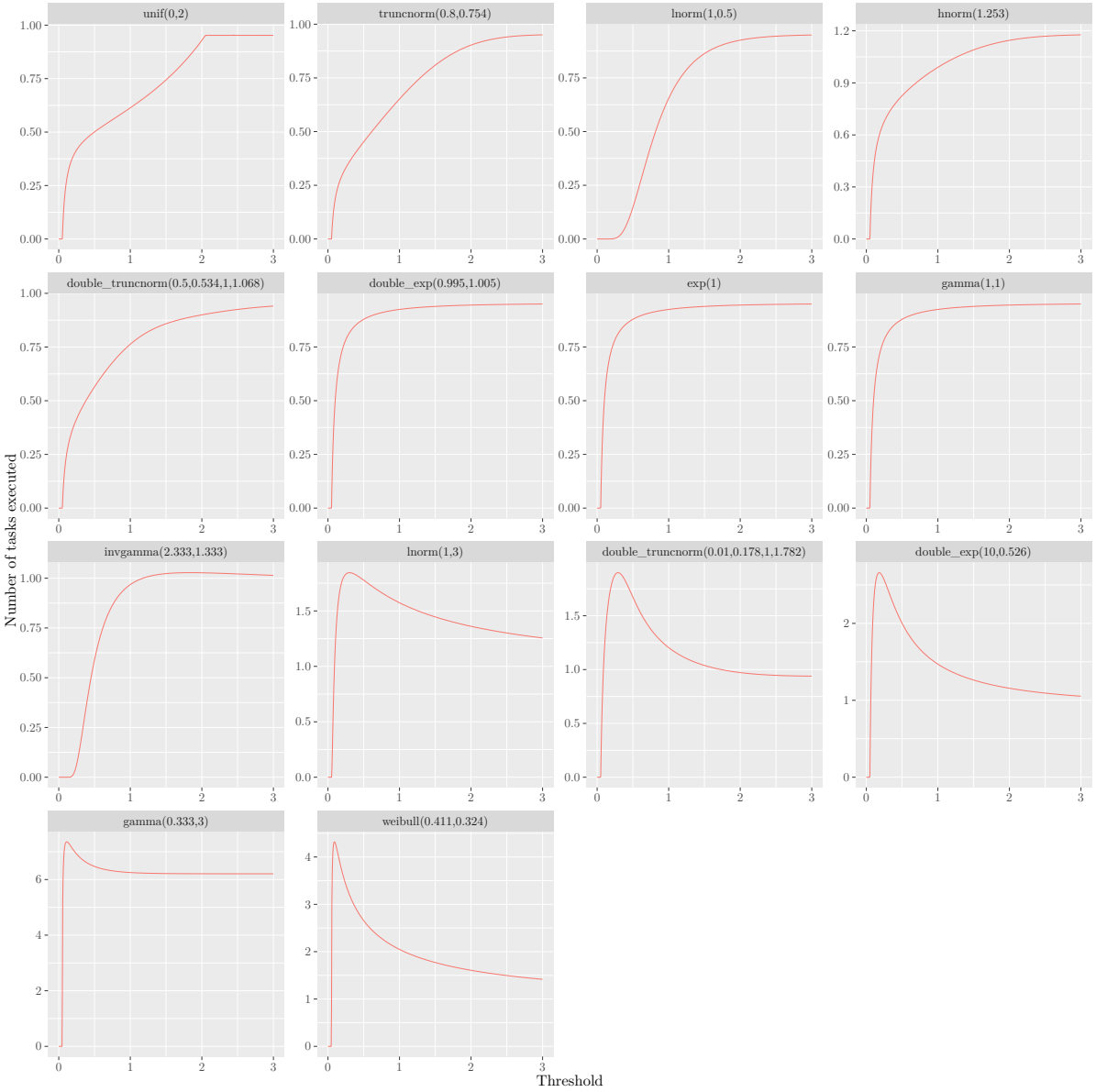


Figure 3.3: Theoretical yield when varying the cutting threshold for each distribution.

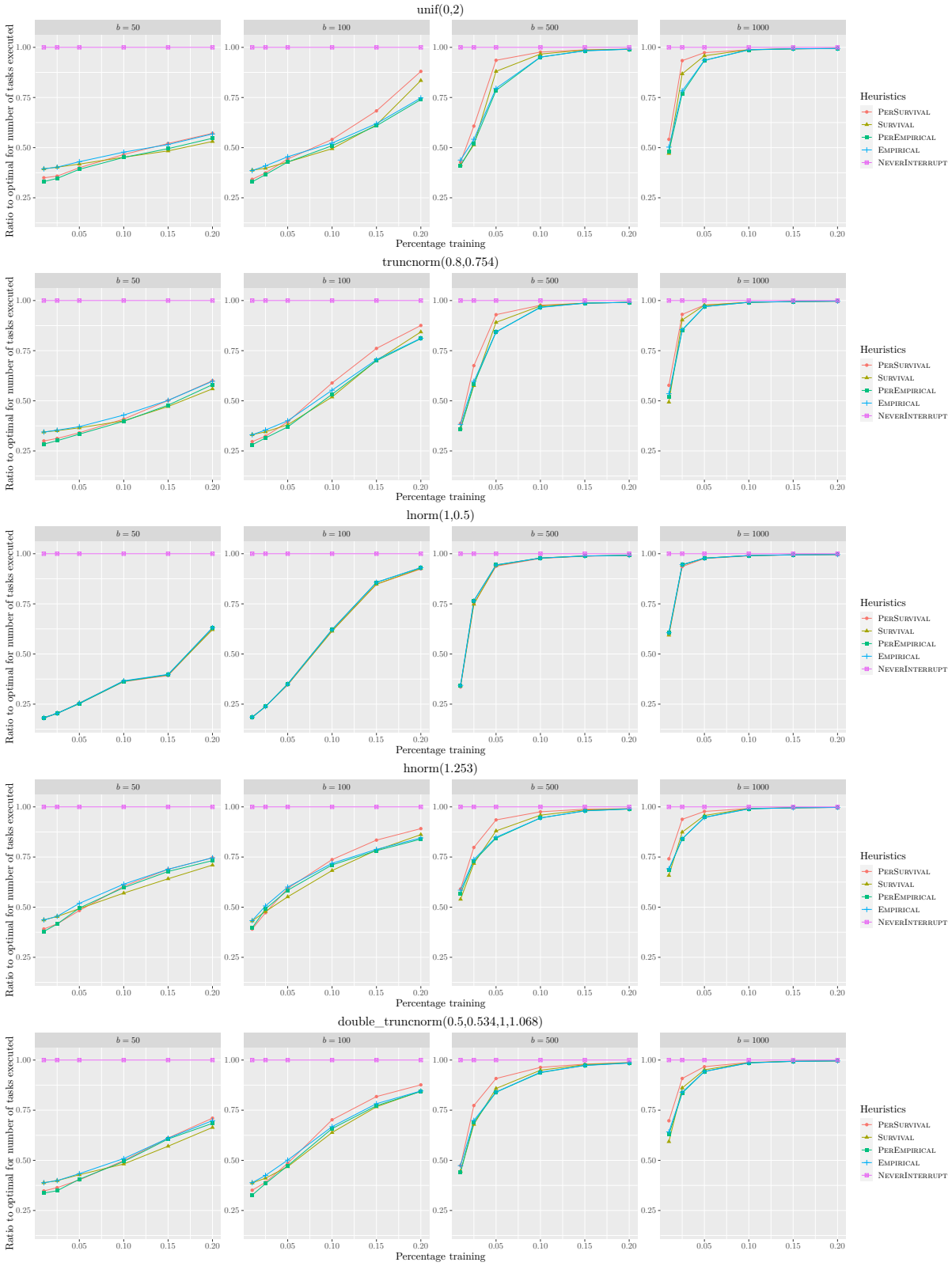


Figure 3.4: Ratio to Oracle of number of tasks successfully executed using different heuristics when varying $p$for each distribution (1/3).

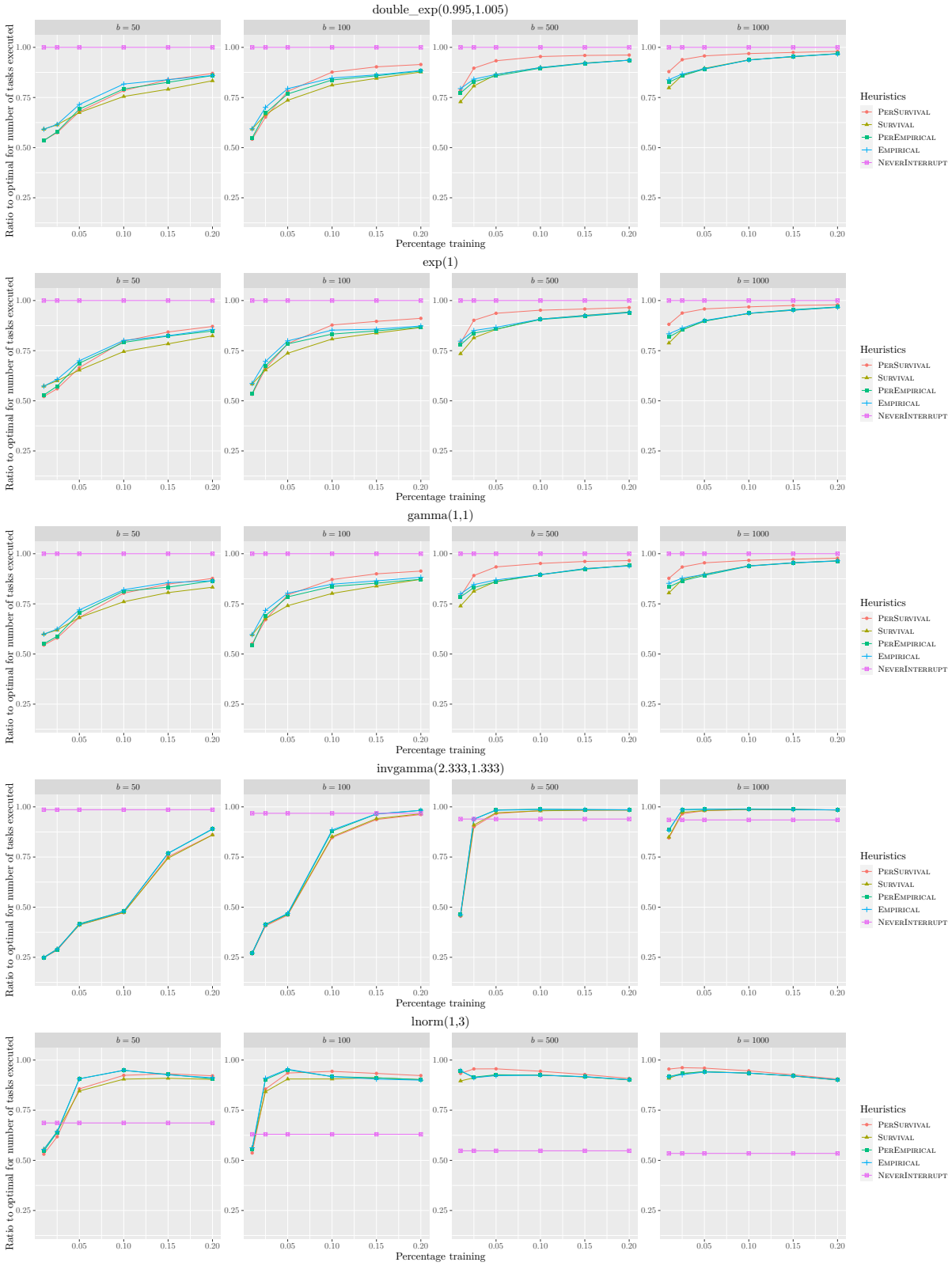


Figure 3.5: Ratio to Oracle of number of tasks successfully executed using different heuristics when varying $p$for each distribution (2/3).

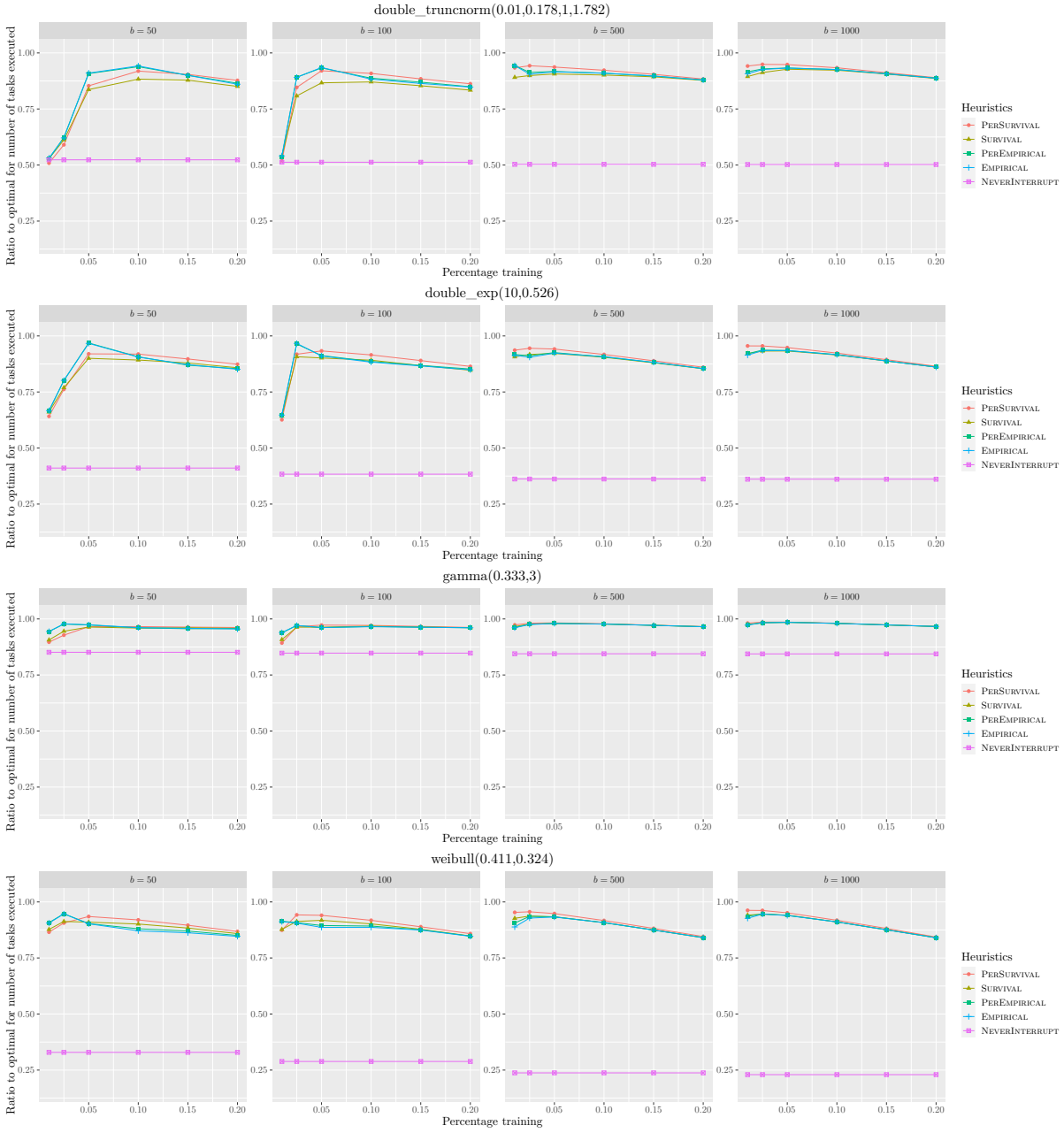


Figure 3.6: Ratio to Oracle of number of tasks successfully executed using different heuristics when varying $p$for each distribution (3/3).

Table II: Symbol and parameters for the distributions used in the simulations. (For all distributions μ is the mean and σ the standard deviation, except for the truncated normal and half-normal distributions where μ and σ are the mean and standard deviation of the original normal distribution.)

Symbol	Distribution	Parameters
double_exp(λ_1, λ_2)	Bimodal exponential	$\lambda_1 = \frac{1}{1.005} \approx 0.995, \lambda_2 = \frac{1}{0.995} \approx 1.005$
		$\lambda_1 = \frac{1}{0.1} = 10, \lambda_2 = \frac{1}{1.9} \approx 0.526$
double_truncnorm(μ_1, σ_1, μ_2, σ_2)	Bimodal truncated normal	$\mu_1 = 0.5, \sigma_1 \approx 0.534, \mu_2 = 1, \sigma_2 \approx 1.068$
		$\mu_1 = 0.01, \sigma_1 \approx 0.178, \mu_2 = 1, \sigma_2 \approx 1.782$
exp(λ)	Exponential	$\lambda = 1$
gamma(k, θ)	Gamma	$k = 1, \theta = 1$
		$k = \frac{1}{3} \approx 0.333, \theta = 3$
hnorm(σ)	Half-normal	$\sigma = \sqrt{\frac{\pi}{2}} \approx 1.253$
invgamma(α, β)	Inverse Gamma	$\alpha = \frac{7}{3} \approx 2.333, \beta = \frac{4}{3} \approx 1.333$
lnorm(μ, σ)	Log-normal	$\mu = 1, \sigma = 0.5$
		$\mu = 1, \sigma = 3$
truncnorm(μ, σ)	Truncated normal	$\mu = 0.8, \sigma \approx 0.754$
unif(a, b)	Uniform	$a = 0, b = 2$
weibull(k, λ)	Weibull	$k \approx 0.411, \lambda = \frac{1}{\Gamma(1+\frac{1}{k})} \approx 0.324$

different parameters, lead to different shapes of the yield function. For the first distributions in the figure, tasks should never be interrupted. For the following distributions, tasks should be interrupted, and sometimes quite early. Table III reports the optimal cutting threshold for each distribution. This variety of situations makes it challenging to determine a good cutting threshold when the distribution is unknown. In the remainder of this section, in order to ease the comparison of the behaviors of the different strategies for the different distributions, all graphs and tables report results with distributions ordered as in Figure 3.3.

One-size-fits-all heuristics

In Figures 3.4, 3.5, and 3.6, we plot the ratio of the number of tasks successfully executed by each heuristic, over the value achieved by Oracle. Hence, the closer to 1, the better. We plot the performance of each heuristic while varying the percentage p of the budget spent for the observation phase (namely $p = 1\%$, 2.5%, 5%, 10%, 15%, or 20%), and for the four different values of the budget b.

We observe that the performance of the different heuristics is strongly correlated to the shape of the yield functions, as illustrated by Figure 3.3. In particular, the performance of the heuristics evolve according to our ordering of the distributions. When the optimal threshold is infinite (i.e., for unif(0,2), truncnorm(0.8, 0.75), lnorm(1,0.5), hnorm(1.25), double_truncnorm(0.5,0.5,0.53,1,1.07), double_exp(0.5,1,1.01), exp(1) and gamma(1,1)), NeverInterrupt has the same performance as Oracle. Also, the performance of the other heuristics increases with p. This is easily explained, since the behavior of the heuristics during the observation phase is, by definition, that of NeverInterrupt. Moreover, the longer the observation phase, the higher the probability that the accumulated data will be of good quality and lead to deriving an efficient threshold.

When the optimal threshold is finite (i.e., for invgamma(2.33,1.33), gamma(0.33,3), lnorm(1,3), double_truncnorm(0.5,0.01,0.18,1,1.78), double_exp(0.5,10,0.53) and weibull(0.41, 0.32)), NeverInterrupt performs predictably worse. The lower the optimal threshold, the

Table III: Optimal cutting threshold for each distribution

Distribution	Optimal Threshold
unif(0,2)	∞
truncnorm(0.8, 0.754)	∞
lnorm(1,0.5)	∞
hnorm(1.253)	∞
double_truncnorm(0.5,0.534,1,1.068)	∞
double_exp(0.995,1.005)	∞
exp(1)	∞
gamma(1,1)	∞
invgamma(2.333,1.333)	1.842
lnorm(1,3)	0.300
double_truncnorm(0.01,0.178,1,1.782)	0.290
double_exp(10,0.526)	0.180
gamma(0.333,3)	0.110
weibull(0.411,0.324)	0.090

lower the performance of NeverInterrupt. Also, the larger the budget, the lower the performance of NeverInterrupt, even if the decrease is not always significant. For the other heuristics, the best value for p decreases. This is once again easily explained, because with larger values of p, the observation phase is longer, and thus the budget spent in a suboptimal mode is larger. The graphs are not decreasing from the start because a significant number of tasks must complete to make a decision close to the optimal one, rather than one that is heavily influenced by the random nature of the very few completion times available.

When the budget is large with respect to the average task execution time (e.g., $b = 1000$), many tasks complete before the end of the observation phase and we can deduce a relatively precise threshold. Hence, the four heuristics perform globally well. For instance, when $p = 10\%$, all heuristics achieve a performance that is at least 90% that of Oracle, whatever the distribution. For some distributions, some heuristics achieve a performance of 95% of this theoretical optimal. This is true even for distributions that, theoretically, need to be cut early, such as Weibull. Because we have enough budget to obtain high-quality threshold after the observation period (which costs 10% of the budget), for the rest of the execution time (90% of the budget), we achieve a performance close to the optimal. Therefore the overall results are very good although we do not interrupt tasks during the observation phase.

When the budget is either $b = 500$ or $b = 1000$, PerSurvival achieves the best performance, or a performance equivalent to that of the best of the four heuristics, except for the inverse-gamma distribution. For inverse-gamma, PerSurvival is sometimes very slightly below PerEmpirical for the same percentage p. However, these two heuristics achieve the same peak performance for that distribution.

On the contrary, when the budget is small with respect to the average task execution time (e.g., $b = 50$), the performance of all heuristics worsens. When $b = 50$ and $p = 10\%$, each of the 10 processors executes tasks for only 0.5 seconds during the observation phase. Hence, the threshold should be computed after very few tasks are completed, if any. It should therefore

not be a surprise that the results are then far from optimal. The best performance is achieved either for the distributions which have a small optimal threshold —and then the performance is rather good whatever the value of p— or when the value of p is large —which compensates from the fact that the budget is small. PERSURVIVAL remains the best heuristic when $b = 100$; when $b = 50$ there is no obvious heuristic of choice.

In conclusion, when the budget b is large with respect to the average task execution time, the four basic and periodic heuristics achieve a good performance (at least 90% of the optimal) if we choose carefully the parameter p (e.g., $p = 10\%$). Then, among the four heuristics, PERSURVIVAL achieves the best performance overall and also in most instances. When the budget is small, the performance of the heuristics worsens. The main reason is that for a same value of p, there are no longer enough completed tasks to make a relevant decision with respect to the threshold. When the budget is small, p should be large if tasks should never be interrupted and p should be small if tasks must be interrupted quickly. Obviously, before running any task we do not know what the average task execution time will be, what the cutting-threshold will be and, hence, how to adequately chose the value of p. The AUTOPERSURVIVAL policy aims at alleviating this problem.

AutoPerSurvival vs. PerSurvival

In Figures 3.7, 3.8 and 3.9, we compare the performance of AUTOPERSURVIVAL for different values of p_{max} (namely p_{max}= 10%, 20%, 30%, 40%, or 50%) when varying ϵ (namely, ϵ= 0.0010, 0.0025, 0.0050, 0.0100, 0.0250, 0.0500, and 0.1000). We added the performance of PERSURVIVAL using different values for p as a reference.

In all graphs, we observe that the performance of AUTOPERSURVIVAL is influenced by the interplay of the two parameters ϵ and p_{max}. When the value of ϵ is very small, we need a very large (in expectation) number of launched tasks to meet the ϵ criteria. This, in turn, will require to spend a large amount of budget for the observation phase. If ϵ is sufficiently small, on most instances this requirement will exceed the upper limit set by p_{max} on the budget spent during the observation phase. Hence, if ϵ is sufficiently small, the behavior of AUTOPERSURVIVAL is only dictated by the value of p_{max}. For instance, when $b = 50$, this is the case for the uniform distribution when $\epsilon \leq 0.0050$, and for the Weibull distribution when $\epsilon \leq 0.0025$. However, when the value of ϵ gets larger, convergence is reached sooner. Then an approximation of the data distribution deemed "good enough" (with respect to ϵ) is obtained before spending a share p_{max} of the budget. In that case, p_{max} does not play any role, and only ϵ has an influence on the observation period, and thus on the performance. For instance, when $b = 100$, this is the case for lnorm(1,3) when $\epsilon \geq 0.0250$. However, for the uniform distribution when $b = 100$, note that $p_{max} = 10\%$ still plays a role when $\epsilon = 0.1000$ which explains why AUTOPERSURVIVAL (10%, 0.1000) has a performance lower than that of the other AUTOPERSURVIVAL variants.

Furthermore, we see that the evolution of the performance depends upon the optimal cutting threshold. When the optimal cutting threshold is infinite, the smaller ϵ, the better the performance. Indeed, during the observation period, the optimal NEVERINTERRUPT strategy is implemented, and, later on, the cutting-threshold strategy is applied. This is particularly true when we have enough time before convergence (large values of p_{max} and of b). In such a case, there is no performance penalty in having a large observation period during which tasks are not interrupted. In contrast, for distributions with a short optimal cutting threshold, small ϵ values (and longer observation periods) waste more budget without interrupting tasks, and the performance decreases when ϵ decreases.

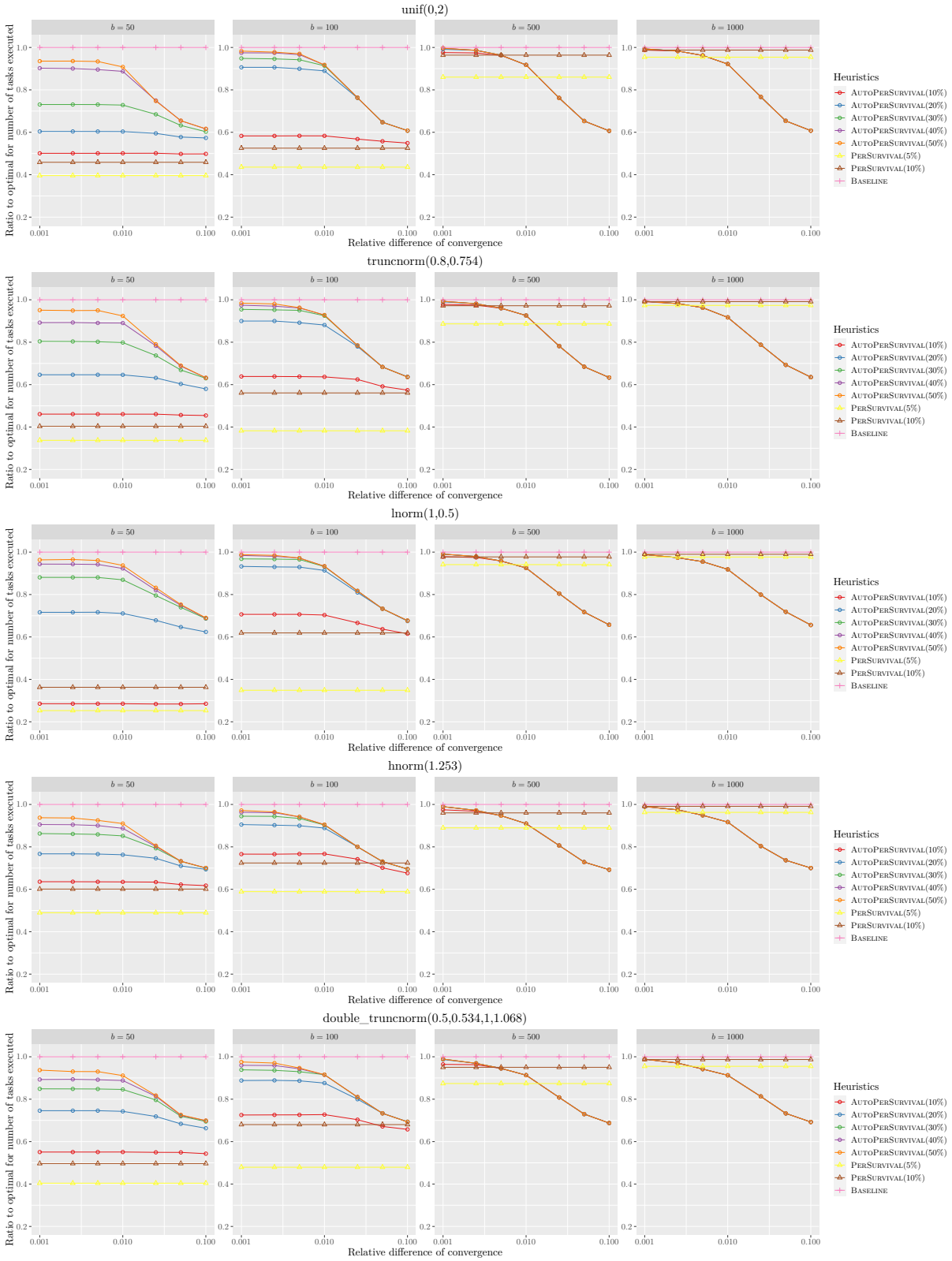


Figure 3.7: Number of successfully executed tasks for each distribution using AutoPerSurvival (different p_{max} when varying ϵ) and PerSurvival (different p) (1/3).

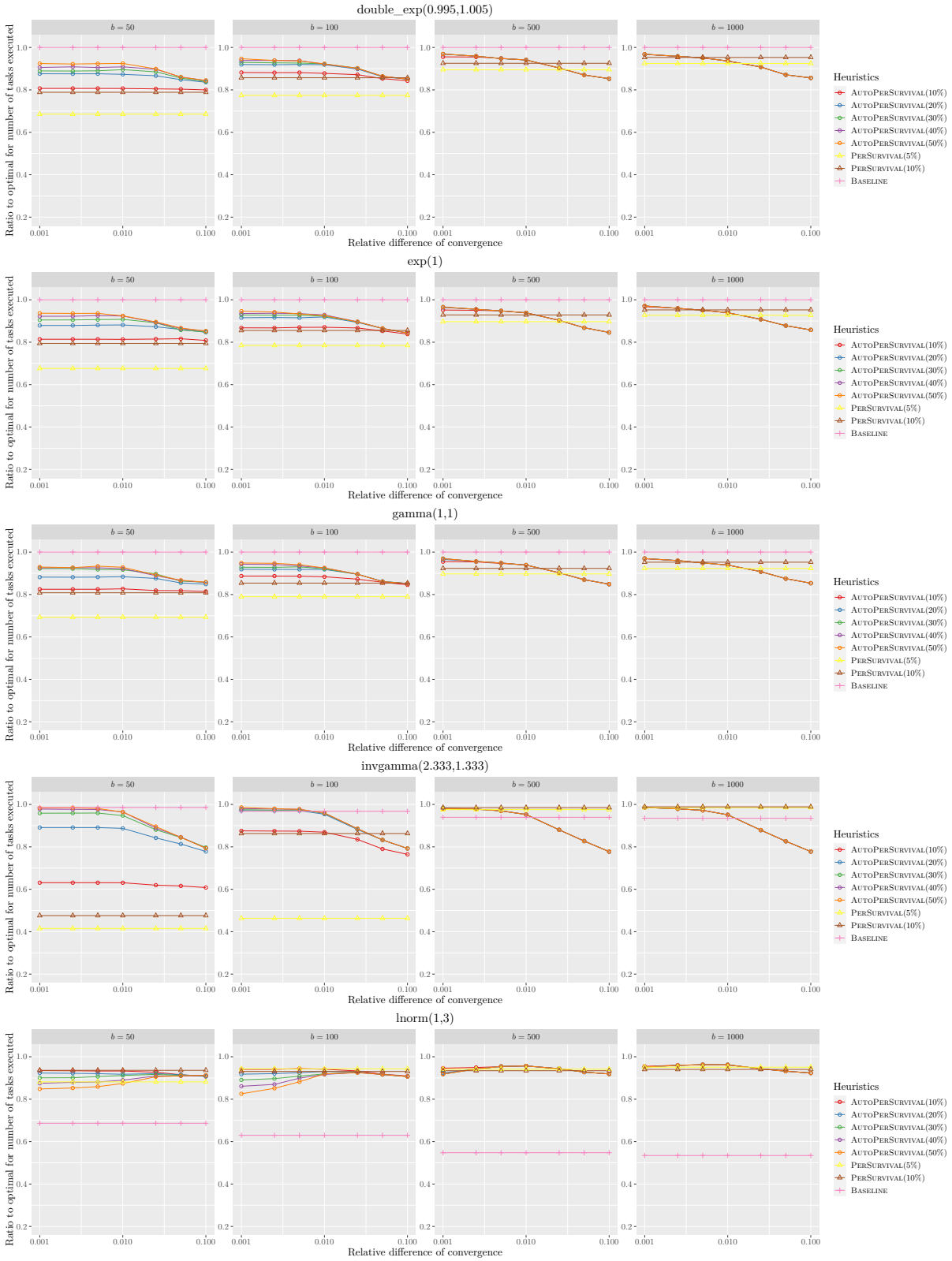


Figure 3.8: Number of successfully executed tasks for each distribution using AutoPerSurvival (different p_{max} when varying ϵ) and PerSurvival (different p) (2/3).

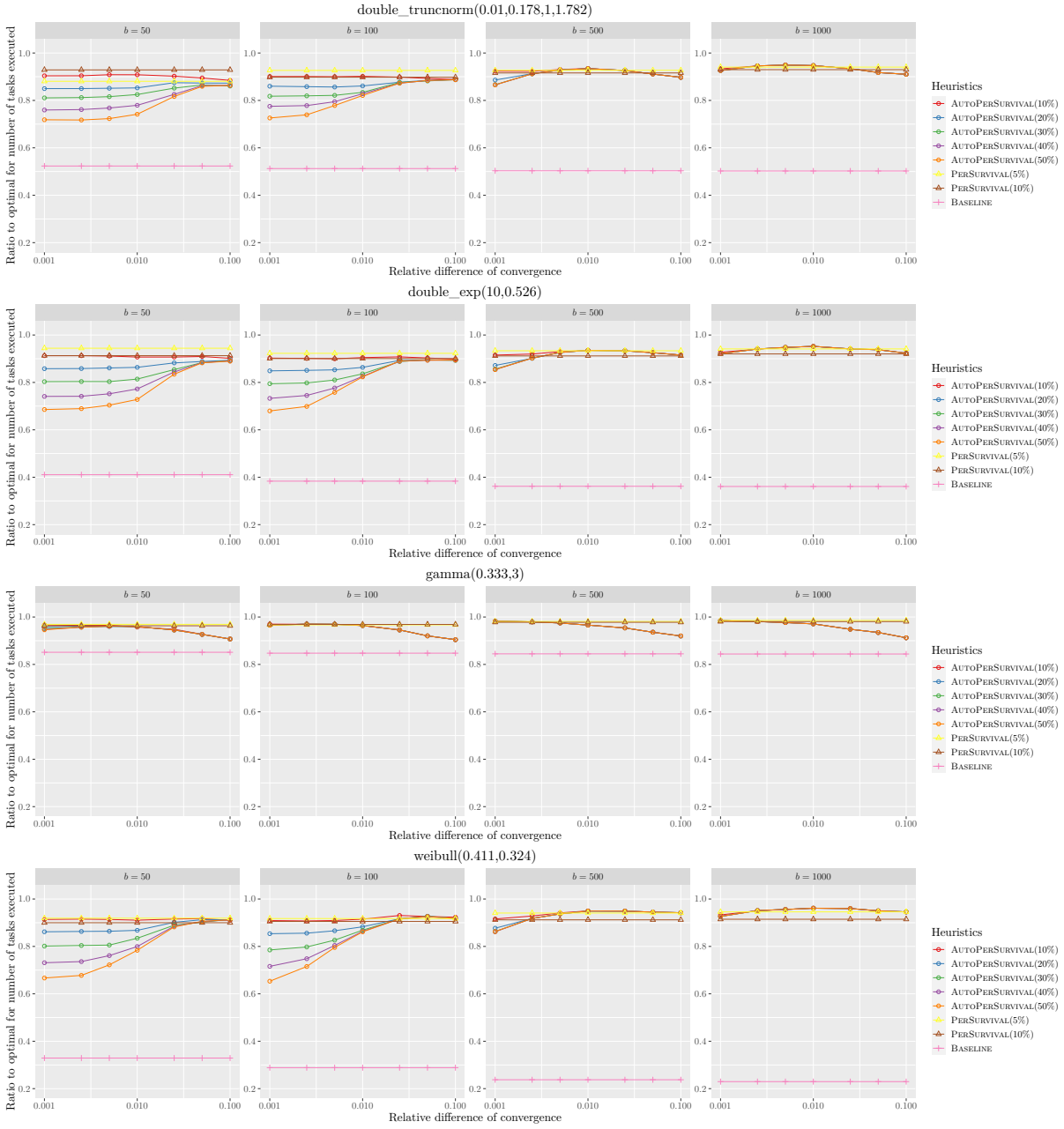


Figure 3.9: Number of successfully executed tasks for each distribution using AutoPerSurvival (different p_{max} when varying ϵ) and PerSurvival (different p) (3/3).

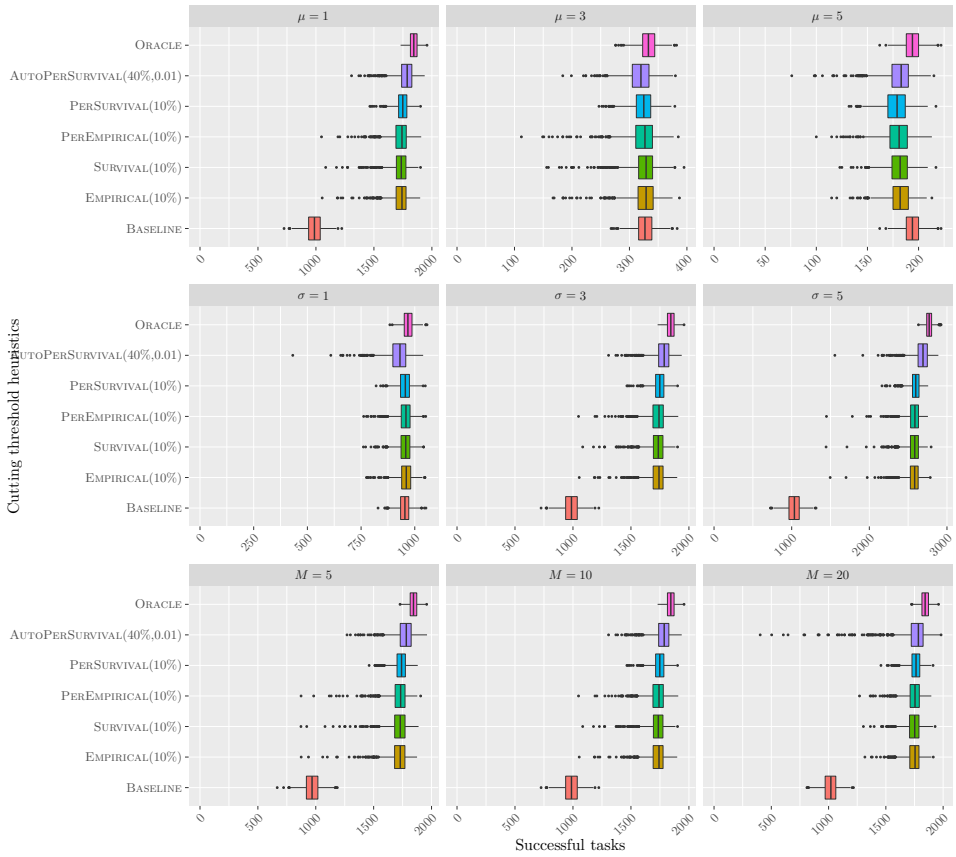


Figure 3.10: Number of successfully executed tasks for the different heuristics with a budget $b = 1000$ when task execution times follow a log-normal distribution. Unless otherwise specified, the expectation is $\mu = 1$, the standard deviation is $\sigma = 3$, and the number of processors is $M = 10$.

Globally, when the budget and deadline are large enough, AutoPerSurvival (when $\epsilon \leq 0.0100$) performs similarly to PerSurvival, and they both have a good performance (larger than 90%). In this case, all p_{max} values perform equally well. However, when the budget and deadline decrease, we already know that PerSurvival performs worse, and we observe that the performance of AutoPerSurvival is strongly correlated to the value of p_{max} and ϵ. Among the parameters tested, AutoPerSurvival (40%, 0.01) is a good choice, because it can successfully execute more than 77% of the tasks of the optimal heuristic Oracle, regardless of the distribution and the budget (deadline) values. In other words, using AutoPerSurvival (40%, 0.01) will always lead to good results, contrarily to all *one-size-fits-all* heuristics.

Stability of performance while varying μ, σ, and M

Figures 3.10 and 3.11 assess the performance of the different heuristics under a log-normal distribution of task execution times when $b = 1000$ (Figure 3.10) and $b = 50$ (Figure 3.11) for different values of the average task execution time (μ), of the standard deviation (σ,) and of the number of processors in the platform (M). We use a log-normal distribution because it has been advocated to model file sizes [25], and thus task costs can also be assumed to follow this

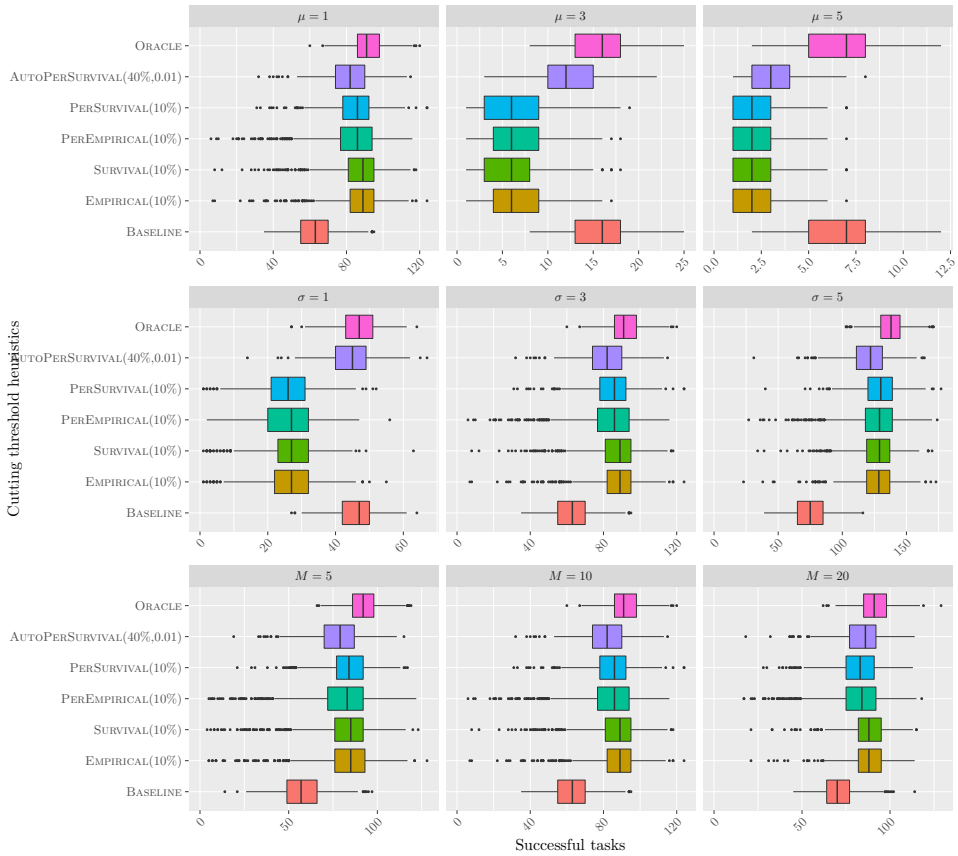


Figure 3.11: Number of successfully executed tasks for the different heuristics with a budget $b = 50$ when task execution times follow a log-normal distribution. Unless otherwise specified, the expectation is $\mu = 1$, the standard deviation is $\sigma = 3$, and the number of processors is $M = 10$.

Table IV: Ratio to ORACLE of number of tasks completed for each heuristic and each distribution with $\mu = 1$, $b = 1000$ and $d = 100$.

	NEVERINTERRUPT	AUTOPERSURVIVAL (40%,0.01)	PERSURVIVAL (10%)	SURVIVAL (10%)	PEREMPIRICAL (10%)	EMPIRICAL (10%)
unif(0,2)	1.0000	0.9217	0.9880	0.9887	0.9873	0.9881
truncnorm(0.8, 0.754)	1.0000	0.9170	0.9911	0.9914	0.9913	0.9914
lnorm(1,0.5)	1.0000	0.9186	0.9894	0.9909	0.9903	0.9909
lnorm(1.253)	1.0000	0.9168	0.9920	0.9920	0.9903	0.9903
double_truncnorm(0.5,0.534,1,1.068)	1.0000	0.9127	0.9881	0.9873	0.9860	0.9859
double_exp(0.995,1.005)	1.0000	0.9360	0.9690	0.9379	0.9370	0.9367
exp(1)	1.0000	0.9388	0.9685	0.9352	0.9362	0.9367
gamma(1,1)	1.0000	0.9391	0.9669	0.9397	0.9381	0.9391
invgamma(2.333,1.333)	0.9350	0.9512	0.9876	0.9871	0.9897	0.9881
lnorm(1,3)	0.5345	0.9620	0.9464	0.9352	0.9344	0.9351
double_truncnorm(0.01,0.178,1,1.782)	0.5023	0.9470	0.9338	0.9229	0.9259	0.9259
double_exp(10,0.526)	0.3610	0.9522	0.9236	0.9151	0.9169	0.9162
gamma(0.333,3)	0.8440	0.9711	0.9810	0.9801	0.9803	0.9799
weibull(0.411,0.324)	0.2296	0.9613	0.9183	0.9108	0.9109	0.9109

distribution. For the heuristics, we choose the parameters which achieved the best performance in the previous simulations: AUTOPERSURVIVAL is used with the parameters $p_{max} = 40\%$ and $\epsilon = 0.01$. For the four *one-size-fits-all* strategies, we use the same value to define the observation phase: $p = 10\%$.

In Figure 3.10, as the budget is big enough ($b = 1000$), all heuristics perform similarly and close to the optimal in all configurations. AUTOPERSURVIVAL may perform slightly better than the four other heuristics in most of the cases but the differences are minimal.

Figure 3.11 presents the more interesting case of a small budget $b = 50$ with respect to the average task execution time. The first row of subgraphs show the influence of the average task execution time, μ, on the performance of heuristics. Remark that for $b = 50$, $p = 10\%$, and $M = 10$, the observation phase for *one-size-fits-all* heuristics only lasts for 0.5 second, during which one expects that very few processors will be able to complete a task. This gets even more true when μ increases, and explains that the performance of the heuristics is decreasing. Nevertheless, the performance of AUTOPERSURVIVAL decreases more slowly than that of the other heuristics. For instance, when $\mu = 3$, the four *one-size-fits-all* heuristics already achieve a rather bad performance while AUTOPERSURVIVAL remains quite close to the optimal. The fact that AUTOPERSURVIVAL automatically adapts the length of its observation phase to the quality of the information that it gathers (here mainly the number of tasks that complete), enables it to achieve a graceful degradation of performance.

The second row of subgraphs shows the impact of the standard deviation σ. When σ varies, the optimal cutting-threshold varies. This is illustrated by the performance of NEVERINTERRUPT which decreases when σ increases, showing that the optimal threshold also decreases. All heuristics have similar performance when $\sigma = 3$ and $\sigma = 5$. However, only AUTOPERSURVIVAL achieves near optimal performance when $\sigma = 1$.

The third row of subgraphs show that varying the number of processors has no significant impact on the performance of the heuristics: all scenarios achieve near optimal performance. Overall, AUTOPERSURVIVAL (40%, 0.01) is a very robust heuristic, which overcomes the other heuristics in all settings, and which, in the most adverse scnearios, exhibits a graceful degradation of performance with respect to the theoretical optimal.

Conclusion

To summarize our findings, we finally present two tables showing the number of tasks completed by each heuristic for each distribution expressed as a fraction of the optimal performance (of ORACLE). We present results for a large budget (Table IV, $b = 1000$ and $d = 100$) and a small

Table V: Ratio to ORACLE of number of tasks succeeded for each heuristic and each distribution with $\mu = 1$, $b = 50$ and $d = 5$

	NEVERINTERRUPT	AUTOPERSURVIVAL (40%,0.01)	PERSURVIVAL (10%)	SURVIVAL (10%)	PEREMPIRICAL (10%)	EMPIRICAL (10%)
unif(0,2)	1.0000	0.8872	0.4659	0.4522	0.4513	0.4775
truncnorm(0.8, 0.754)	1.0000	0.8904	0.4089	0.4002	0.3981	0.4288
lnorm(1,0.5)	1.0000	0.9230	0.3620	0.3646	0.3633	0.3667
hnorm(1.253)	1.0000	0.8874	0.6046	0.5701	0.5974	0.6146
double_truncnorm(0.5,0.534,1,1.068)	1.0000	0.8881	0.4989	0.4818	0.4940	0.5088
double_exp(0.995,1.005)	1.0000	0.9086	0.7846	0.7552	0.7927	0.8172
exp(1)	1.0000	0.9235	0.7973	0.7456	0.7916	0.8010
gamma(1,1)	1.0000	0.9221	0.8033	0.7604	0.8133	0.8203
invgamma(2.333,1.333)	0.9858	0.9647	0.4729	0.4738	0.4790	0.4788
lnorm(1,3)	0.6865	0.8896	0.9238	0.9048	0.9484	0.9493
double_truncnorm(0.01,0.178,1,1.782)	0.5233	0.7794	0.9190	0.8831	0.9386	0.9420
double_exp(10,0.526)	0.4107	0.7725	0.9190	0.8925	0.9068	0.9057
gamma(0.333,3)	0.8513	0.9599	0.9658	0.9592	0.9603	0.9610
weibull(0.411,0.324)	0.3291	0.7995	0.9198	0.9014	0.8802	0.8711

one (Table V, $b = 50$ and $d = 5$) with respect to the average task execution time ($\mu = 1$). Obviously, we use the same heuristic parameters than previously: $\epsilon = 0.01$, $p_{max} = 40\%$, and $p = 10\%$.

Table IV shows that, with large values of budget and deadline, all heuristics perform well. Indeed, with the chosen parameters all heuristic achieve at least 91% of the performance of the optimal. Among the *one-size-fits-all* heuristics, PERSURVIVAL performs best and is the most robust, but the difference between these heuristics is not always significant. On average the performance of AUTOPERSURVIVAL and PERSURVIVAL are pretty similar.

Table V presents the result when budget and deadline are small. In this case all *one-size-fits-all* heuristics achieve very low performance, below 40% for each of them (for the log-normal distribution). On the contrary, AUTOPERSURVIVAL always achieves good to very good performance: its worse case is 77% of the optimal. Once again, this shows the great robustness of AUTOPERSURVIVAL (40%, 0.01).

3.6 Conclusion

In this work, we have studied the problem of maximizing the number of tasks successfully executed on a cloud platform under deadline and budget constraints. When task execution times obey a probability distribution that is known before execution, previous results showed that long-running tasks must be interrupted at some optimal cutting threshold τ, and provided techniques to determine its value. Some probability distributions call for a very short threshold τ while others have a large or infinite one. The main difficulty in this study is that the probability distribution of task execution times is unknown to the scheduler. We designed a set of scheduling heuristics to estimate the cutting threshold τ, some of which making use of the Kaplan-Meier estimator. We also assessed different decision mechanisms to recompute the threshold as the execution progresses. On the practical side, extensive simulations show that our best heuristic AUTOPERSURVIVAL (40%, 0.01) achieves good performance for a wide spectrum of probability distributions and parameter sets. In the worst scenario, it can execute 79% of tasks that an omniscient oracle (knowing the distribution) would be able to complete.

Chapter 4

Efficient task-dropping strategies for firm real-time systems

4.1 Introduction

As mentioned in the introduction, we pass to firm real-time tasks in this chapter. In the classical setting of the problem with firm real-time tasks, there are several periodic tasks that are input to the system. Each periodic task is composed of instances that are released with a given period and deadline. All instances of a periodic task have same duration. The objective is to maximize the number of task instances that successfully complete before their deadline. There are many variations: for instance, some tasks may have two different types, skippable or not [68], and only skippable tasks are allowed to miss their deadline.

In this chapter, we revisit the problem with firm real-time tasks in a framework closer to that of the previous work and in Chapters 2 and 3. We deal with a single periodic task and assume that all task instances are skippable, which simplifies the scheduling, but we no longer assume that all instances have the same execution time, which dramatically complicates it. As in the previous work and in Chapter 2, we assume that the task execution times obey some probability distribution $\mathcal{D}$ known in advance, and we provide experiments with a wide range of standard distributions.

Specifically, tasks arrive periodically and enter automatically the waiting queue for follow-up allocation: instance number i, $i \geq 1$, is released at time $r_i = \tau \times (i-1)$ and must complete not later than time $\delta_i = r_i + d$, where τ is the period and d the deadline. More precisely, if task i is not completed by δ_i, it is considered as failed, and any resource spent to execute part of it has been wasted. We consider a parallel platform with identical processors. The system can decide whether, when, and on which processor to start the execution of each task. It can also decide, at any instant, to interrupt the execution of a (long) running task and to launch a new one, or to drop a task still in the waiting queue. Dealing with such stochastic execution times introduces new challenges: If the support of the distribution $\mathcal{D}$ is unbounded, some tasks may execute for an arbitrarily long duration, thereby putting the following tasks at risk. However, if we decide to interrupt a task to launch a new one, the time already spent to execute it is lost, and there is no guarantee that the new task will complete faster than the interrupted one. Note that the problem is similar to scheduling overloaded systems that allow skips [57].

Although the task model varies compare to Chapters 2 and 3, we can still find some common points: this task model also assumes that some tasks may not be executed in the end. On the

other hand, the optimization objective remains to maximize the fraction of tasks that will be completed before their deadline.

Within this framework, scheduling decisions become complicated. When a processor is idle, we need to decide which task in the waiting queue should be launched. Recently released tasks have a longer deadline, but maybe the probability to complete a more ancient task successfully is high enough to be worth the try. Furthermore, we need to decide whether and when to interrupt long-lasting tasks; at a given time-step, this decision depends upon how long the task has been executing and upon how many tasks are in the waiting queue, and since when.

This work introduces and evaluates several heuristics to solve this challenging scheduling problem. We first deal with the single processor case, and use as reference the heuristic NeverKill that launches tasks in the order of arrivals and never interrupt any task before its deadline is reached. We show that our best heuristics have a significantly better performance than NeverKill in nearly all cases. Then we extend these best heuristics to the multiprocessor case, showing that the gain in performance remains. The major contributions of this work are the following:

- We design several heuristics, that dynamically decide to interrupt some (long-lasting) tasks and launch new ones, based upon a variety of criteria including current length of execution time, remaining time until deadline and duration of time spent in waiting queue since release;
- We construct a Markov chain based upon a discretization of time and probability values, we show that the chain is both aperiodic and irreducible, and we compute the optimal throughput via the limit distribution of the chain;
- We conduct an experimental evaluation based on a comprehensive set of simulations scenarios, showing that our best heuristics achieve a significant gain over the whole spectrum of application and platform parameters.

The rest of the chapter is organized as follows. Section 4.2 provides a description of the optimization problem under study. The design of our heuristics is detailed in Section 4.3. The Markov chain is constructed and solved in Section 4.4. Section 4.5 is devoted to a comprehensive experimental comparison of the heuristics. Finally, Section 4.6 gives concluding remarks and hints for future work.

4.2 Problem framework

In this work, we are considering the case where the system comprises a set of M identical processors m_1, ..., m_M. We assume that the processors are never voluntarily left idle. We consider a task instance set $\mathcal{T}$, in which all instances are independent, and their execution times follow the very same distribution $\mathcal{D}$. We consider discretized distribution while establishing theoretical equations for our problem, and we use continuous distribution during our simulation. Let X be the random variable for the execution time of a task. Then, let us define $p_t = \mathbb{P}(X = t)$. In other words, p_t is the probability that the execution duration of a task is t. As described before, tasks arrive periodically with period τ. Task i arrives at time $r_i = (i-1)\tau$, and has a window of size d to be executed. In other words its deadline is $\delta_i = r_i + d$. If $d - \tau \leq 0$, that is if $d \leq \tau$, there is no overlap between tasks, and it is obvious that we can let all tasks to be executed until their deadline. Therefore, in our problem, we assume that $d > \tau$.

The objective is to maximize the throughput of tasks successfully completed. In other words, we want to maximize the number of tasks successfully completed before their deadline per unit of time. As there is one new task arriving every τ units of time, it is equivalent to maximize the throughput in successfully completed tasks and to maximize the probability that a task arriving in the system is successfully completed.

Thus, we need to find a heuristic which solves at first the following question: On which processor to execute each task? After that, comes the second question: Whether and when to execute the task on its attributed processor and, if it is allowed to run, for how long time?

After defining a heuristic, in order to evaluate the probability of success of a task, we need to define the following notations:

- $s_{i,j}$: number of time units after the release of task i at which the processor m_j will become available for task i (i.e., the absolute time at which m_j becomes available for task i is $r_i + s_{i,j}$).
- $\mathcal{S} = (s_1, ..., s_M)$: the state of the system. The state of the system is defined by the list of the times at which the different processors will be available after the release of the task under consideration.
- $p_{\mathcal{S}}$: the probability that state $\mathcal{S}$ occurs.
- Φ_i: the set of all possible states of the system at the release date of task i.
- $P_{i,j}^{avail}(t)$: the probability that processor m_j will become available for task i t time units after its release time (i.e., $\mathbb{P}(s_{i,j} = t)$). As there is no value to execute a firm real-time task after its deadline, it will obviously be stopped at the deadline even if it is not completed yet. Therefore, every processor will be available at the latest $d - \tau$ time units after task i is released, because this is the deadline for the previous task, task $i - 1$.
- c_i: if task i is successfully finished, $c_i = \mathit{True}$; if task i is interrupted or is never started, $c_i = \mathit{False}$.
- $P_{i,j}^{comp}(t)$: the probability that task i will complete successfully if processor m_j becomes available t time units after its release time (i.e., $\mathbb{P}(c_i | s_{i,j} = t)$).

Consider a heuristic which will choose to map task i on some processor $h(i, \mathcal{S})$ if the system is the state $\mathcal{S}$ when task i is released. Then the probability of success of task i under this heuristic is:

$$\sum_{\mathcal{S} \in \Phi_i} p_{\mathcal{S}} P_{i,h(i,\mathcal{S})}^{comp}(s_{i,h(i,\mathcal{S})})$$

Our objective is to find a heuristic which maximizes this value for *any* task.

4.3 Heuristics

In order to solve the problems defined above, we designed a two-phase heuristic. It comprises at first the task mapping phase, in which we decide which processor is going to execute which task. And then, we will pass to the task admission phase, in which we must decide, for each processor and each task attributed to it, whether to start it and if started, whether to interrupt its execution at some point or let it run either through completion or until it reaches its deadline.

Task mapping. We consider in this work the two following attribution heuristics: the Round robin strategy (RoundRobin) and the Earliest start time strategy (EarliestStartTime). Under the RoundRobin strategy, task i is to be executed by processor $h(i) = 1 + (i \mod M)$. After that, the M processors process their sets of tasks fully independently. On the other hand, under the EarliestStartTime strategy, task i will be executed by any processor $h(i)$ which satisfies $s_{i,h(i)} = \min_{1 \leq j \leq M} s_{i,j}$.

Task admission. We define three criteria about whether and for how long to execute tasks:

- Upper bound on starting times ($s_{\max}$): We define $s_{\max}$ as the upper bound on the start time of a task. Task i cannot start later than time $r_i + s_{\max}$. As described above, a processor is available for task i at the latest at time $r_i + d - \tau$. There is no sense to set the $s_{\max}$ to a value larger than $d - \tau$, because this is equivalent to the case of $s_{\max} = d - \tau$. Thus, we assume that $0 \leq s_{\max} \leq d - \tau$.

- Upper bound on execution times ($l_{\max}$): In this variant, we do not allow a task to run for longer than a time $l_{\max}$. Obviously, if $l_{\max} \geq d$, this is equivalent to not having any upper bound on the execution time of tasks. On the other hand, there is no sense to take a $l_{\max}$ value smaller than or equals to τ. In this case, each task will be interrupted before the next task is released and the processor will be left idle. Hence, in the following we assume that we have $\tau < l_{\max} \leq d$.

- Upper bound on interruption times ($d_{\max}$): In this variant, tasks are interrupted $d_{\max}$ time units after its release time, instead of allowing them to execute until their deadline. Similarly as in the case of $l_{\max}$, we can also assume that $\tau < d_{\max} \leq d$.

In our heuristics, we can combine the above three criteria arbitrarily, and we can obtain in total eight algorithms:

- NeverKill: This is the heuristic in which we do not interrupt any tasks voluntarily. A task can only be thrown by its deadline if it is not started or completed earlier. In other words, there is not any $s_{\max}$, $l_{\max}$ or $d_{\max}$ constraints.

- Other heuristics (Smax+Lmax+Dmax, Smax+Lmax, Smax+Dmax, Lmax+Dmax, Smax, Lmax, Dmax): In these heuristics, only the criteria appearing in the heuristic name are enabled. When a criterion is disabled in an algorithm, we set its value to d (for $l_{\max}$ and $d_{\max}$) or $d - \tau$ (for $s_{\max}$), which means that there is no restriction on this criterion. In contrast, if a criterion is enabled, we need to find the value which has the best expected performance, within its valid value range described above. After that, tasks will be started, stopped or dropped according to the criterion value.

Different criteria values ($s_{\max}$, $l_{\max}$ and $d_{\max}$) leads to different number of tasks successfully completed. Given a parameter set (period τ, deadline d and distribution of task execution times $\mathcal{D}$) and the set of criteria enabled $\mathcal{C}$, we need to design an algorithm to find the combination of values of criteria in $\mathcal{C}$ which can achieve the best performance.

In order to reduce the complexity of our algorithm, we define a quantum duration q. We discretize a period τ into $\frac{\tau}{q}$ quanta. A processor has $\frac{d-\tau}{q} + 1$ possible states when task i arrives: it can be available 0, 1, ..., $\frac{d-\tau}{q}$ quanta after the release of the task. If we find the probability that each of these states appear while releasing task i, where i tends to infinity, we can then

calculate the asymptotic expected probability that a task is executed successfully, because the probability of task execution time is known. Thus, given a parameter set (τ, d and $\mathcal{D}$) and a combination of criteria values ($s_{\max}$, $l_{\max}$ and $d_{\max}$), we can construct a Markov chain to present these states, and then calculate its limit distribution to find the asymptotic expected probability that a task can be successfully executed. If we traverse all possible values of enabled criteria, we can find the best combination of criteria values, for which the asymptotic expected probability is the largest. Algorithm 1 presents this procedure. On the other hand, if we have only one criteria enabled in our heuristic, we can hypothesize that there exists only one local maximum of performance, and we can check through experiments that this hypothesis is correct. Thus we can use a binary search to find the best criterion value. This can reduce the execution time of heuristics. This procedure is presented in Algorithm 2. A more detailed explanation about the sub-algorithm calculating the asymptotic expected success probability of tasks is presented in Section 4.4.

What needs to be added is, as tasks arrive periodically and their deadline is always d time units after their release time, we will treat them in a First-In First-Out order: When task $i-1$ is successfully completed or is interrupted, we treat always task i. Evidently, task i can not be executed at all and be dropped directly because of some task admission criteria. In this case, we will pass immediatly to task $i+1$.

4.4 Resolution of Markov chain

In the previous sections, we defined our problem and we presented the heuristics used in the paper. In this section, we continue to consider a theoretical world, and use a Markov chain to represent the states that we can encounter during the execution, in order to find the expected result of the objective function.

We consider different task admission heuristics for each of our attribution strategy. For RoundRobin, as we can consider each processor independently, in Section 4.4.1, we establish at first a linear system for the single processor case, and then extend the result to RoundRobin in multiple processors case. For EarliestStartTime, the situation becomes more complicated, because the task attribution on one processor is no more periodic. In Section 4.4.2, we try to merge equivalent states and describe them, in order to reduce the computational cost.

4.4.1 RoundRobin strategy

In this section, we give at first the Markov chain resolution method in the single processor case. After that, we talk about how to extend the result to RoundRobin heuristics in the multiple processor case.

Equations of NeverKill and single criterion heuristics

In this section, we have a single processor platform. All tasks are attributed to processor m_1. We establish the equations for NeverKill and for each case while enabling a single task admission criterion ($s_{\max}$, $d_{\max}$, $l_{\max}$) to present the success probability $P_{i,1}^{comp}(t)$ and the available probability $P_{i,1}^{avail}(t)$ in terms of p_t and $P_{i-1,1}^{avail}(t)$. As we discretized the system into quanta of duration q. In the following equations, t can take any multiple of q between 0 and d.

Algorithm 1: Search of the best criteria values given a parameter set

Input: q: quantum duration, $\mathcal{C}$: criteria enabled
τ: task releasing period, d: task deadline, $\mathcal{D}$: distribution of task execution times
Output: c_{best}: table which contains the best combination of criteria values

```
1  begin
2      if "s_max" ∈ C then
3          S_max = {0, q, 2q, ..., d − τ}
           /* s_max can take value between 0 and (d−τ)/q quanta */
4      else
5          S_max = {d − τ}
           /* s_max is not constrainted, and is set to the maximal value */
6      if "l_max" ∈ C then
7          L_max = {τ + q, τ + 2q, ..., d}
           /* l_max can take value between τ/q and d/q quanta */
8      else
9          L_max = {d}
           /* l_max is not constrainted, and is set to the maximal value */
10     if "d_max" ∈ C then
11         D_max = {τ + q, τ + 2q, ..., d}
           /* d_max can take value between τ/q and d/q quanta */
12     else
13         D_max = {d}
           /* d_max is not constrainted, and is set to the maximal value */
14     perf_best = 0
       /* maximal expected performance */
15     c_best = [0, 0, 0]
       /* criteria values for which the maximal performance is reached */
16     for s_max ∈ S_max do
17         for l_max ∈ L_max do
18             for d_max ∈ D_max do
19                 c_current = [s_max, l_max, d_max]
20                 perf_tmp = ResolveMarkovChain(q, c_current, τ, d, D)
                   /* Function to resolve the Markov chain, and to return the expected
                      success probability according to the input parameter set */
21                 if perf_tmp > perf_best then
22                     c_best = [s_max, l_max, d_max]
23                     perf_best = perf_tmp
                   /* Renew c_best and perf_best if we find a better performance */
24     return c_best
```

Algorithm 2: Binary search of the best values given a criterion and a parameter set

Input: q: quantum duration, $crit$: (the only) criterion enabled ($s_{\max}$:0, $l_{\max}$:1, $d_{\max}$:2)
τ: task releasing period, d: task deadline, $\mathcal{D}$: distribution of task execution times
Output: c_{best}: best criterion value found

begin
  $c_{LB} = [d - q, d, d]$
  $c_{LB}[crit] = 0$
  $c_{UB} = [d - q, d, d]$
  /* Define initial value of lowerbound c_{LB} and upperbound c_{UB} according to criterion enabled */
  $perf_{LB} = ResolveMarkovChain(q, c_{LB}, \tau, d, \mathcal{D})$
  $perf_{UB} = ResolveMarkovChain(q, c_{UB}, \tau, d, \mathcal{D})$
  /* Calculate the expected performance of the lowerbound and the upperbound */
  while $c_{UB}[crit] - c_{LB}[crit] > q$ **do**
    $c_{target} = [d - q, d, d]$
    $c_{target}[crit] = \lfloor \frac{c_{UB}[crit] - c_{LB}[crit]}{2q} \rfloor * q$
    $perf_{target} = ResolveMarkovChain(q, c_{target}, \tau, d, \mathcal{D})$
    if $perf_{LB} \leq perf_{target}$ *and* $perf_{target} \leq perf_{UB}$ **then**
      /* If the function is apparently non-decreasing, we focus on the second half of the interval */
      $c_{LB} = c_{target}$
      $perf_{LB} = perf_{target}$
    else if $perf_{LB} \geq perf_{target}$ *and* $perf_{target} \geq perf_{UB}$ **then**
      /* If the function is apparently non-increasing, we focus on the first half of the interval */
      $c_{UB} = c_{target}$
      $perf_{UB} = perf_{target}$
    else if $perf_{LB} \geq perf_{target}$ *and* $perf_{target} \leq perf_{UB}$ **then**
      /* If the middle point is the minimum of the three tested values, according to our hypothesis, the maximum of the three tested values correspond to the optimum */
      if $perf_{UB} \geq perf_{LB}$ **then**
        $c_{best} = c_{UB}[crit]$
      else
        $c_{best} = c_{LB}[crit]$
      return c_{best}
    else
      /* The performance at the target point is the maximum of the three tested values. We have no way to know on which half the absolute maximum resides. We test the next point to know what the "derivative" is at the target point */
      $c_{next} = [d - q, d, d]$
      $c_{next}[crit] = c_{target}[crit] + q$
      $perf_{target} = ResolveMarkovChain(q, c_{next}, \tau, d, \mathcal{D})$
      if $perf_{target} \geq perf_{next}$ **then**
        /* If the performance at the target point is higher, the maximum is no later than the target point */
        $c_{UB} = c_{target}$
        $perf_{UB} = perf_{target}$
      else
        /* Otherwise, it is no earlier than the target point */
        $c_{LB} = c_{target}$
        $perf_{LB} = perf_{target}$
  if $perf_{LB} \geq perf_{UB}$ **then**
    $c_{best} = c_{LB}[crit]$
  else
    $c_{best} = c_{UB}[crit]$
  return c_{best}

NeverKill $P_{i,1}^{comp}(t)$ is easy to establish and is common to all tasks:

$$P_{i,1}^{comp}(t) = \mathbb{P}(X \leq d-t) = \sum_{k=1}^{\frac{d-t}{q}} p_{kq}$$

One can remark that

$$P_{i,1}^{comp}(d-\tau) = \sum_{k=1}^{\frac{\tau}{q}} p_{kq}$$

and that for $0 \leq t < d-\tau$

$$P_{i,1}^{comp}(t) = P_{i,1}^{comp}(t+q) + p_{d-t}$$

As for $P_{i,1}^{avail}(t)$, if $i = 1$, we have $P_{1,1}^{avail}(0) = 1$. In other cases, we have to consider two limit cases and one intermediary case:

Case $P_{i,1}^{avail}(0)$. For the processor to be available at the release time of task i, task $i-1$ must have successfully completed at the latest at that time and, therefore, it must have started at least one quantum earlier.

$$P_{i,1}^{avail}(0) = \sum_{s=0}^{\frac{\tau}{q}-1} \mathbb{P}(X \leq \tau - sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\frac{\tau}{q}-1} \left(\sum_{k=1}^{\frac{\tau}{q}-s} p_{kq} \right) P_{i-1,1}^{avail}(sq)$$

Case $P_{i,1}^{avail}(t)$ with $q \leq t \leq d-\tau-q$. For the processor to be available at the time $r_i + t = r_{i-1}+\tau+t$, the execution of task $i-1$ must complete exactly at that time. The processing of task $i-1$ starts at time $r_{i-1}+sq$ (i.e., s quanta after its release time) where $0 \leq sq \leq d-\tau$, and where $r_{i-1} + sq < r_i + t = r_{i-1} + \tau + t$, because task $i-1$ must start at least one quantum before task i.

$$P_{i,1}^{avail}(t) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} \mathbb{P}(X = t+\tau-sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} p_{t+\tau-sq} P_{i-1,1}^{avail}(sq)$$

Case $P_{i,1}^{avail}(d-\tau)$. At time $r_i + d - \tau = r_{i-1} + d$, task $i-1$ reaches its deadline and will be stopped whether it is completed or not. We just consider all possible starting times for task $i-1$ and the probability that it is executed until the deadline.

$$P_{i,1}^{avail}(d-\tau) = \sum_{s=0}^{\frac{d-\tau}{q}} \mathbb{P}(X \geq d-sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\frac{d-\tau}{q}} \left(1 - \sum_{k=1}^{\frac{d}{q}-s-1} p_{kq} \right) P_{i-1,1}^{avail}(sq)$$

Criterion $s_{\max}$ enabled In this case, the latest allowed start time for each task is $s_{\max}$. Recall that we assume $s_{\max} \leq d-\tau$.

The equation for the probability of success is the same as previously as long as $t \leq s_{\max}$. Otherwise, the task will not be executed and the probability is always 0.

$$P_{i,1}^{comp}(t) = \begin{cases} \mathbb{P}(X \leq d-t) = \sum_{k=1}^{\frac{d-t}{q}} p_{kq} & \text{if } t \leq s_{\max} \\ 0 & \text{otherwise} \end{cases}$$

The calculation of the probability of availability becomes more complicated:

Case $P_{i,1}^{avail}(0)$. In this case, the equation is the same as for the NeverKill heuristic:

$$P_{i,1}^{avail}(0) = \sum_{s=0}^{\frac{\tau}{q}-1} \mathbb{P}(X \leq \tau - sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\frac{\tau}{q}-1} \left(\sum_{k=1}^{\frac{\tau}{q}-s} p_{kq} \right) P_{i-1,1}^{avail}(sq)$$

Case $P_{i,1}^{avail}(t)$ with $q \leq t \leq d - \tau - q$. For the processor to be available at the time $r_i + t = r_{i-1} + \tau + t$, the execution of task $i-1$ must complete exactly at that time or processor is available for task $i-1$ at that time, but is killed because of $s_{\max}$. As we assumed that $s_{\max} \leq d - \tau$, the processing of task $i-1$ can start at time $r_{i-1} + sq$ where $0 \leq sq \leq s_{\max}$. We have two cases to consider:

$q \leq t \leq s_{\max} - \tau$ **or** $d - 2\tau + q \leq t \leq d - \tau - q$**.** In this variant, only the first case described above is possible: the execution of task $i-1$ completes exactly at that time. It is not possible that the processor is available for task $i-1$ and it is immediately be killed by $s_{\max}$ within this interval because: (i) if $t \leq s_{\max} - \tau$, the available time for task i is earlier than $r_i + s_{\max} - \tau = r_{i-1} + s_{\max}$. If the processor is available for task $i-1$ exactly at that time, it will not be killed by the $s_{\max}$, but will start to run. (ii) if $t > d - 2\tau$, the available time for task i is strictly later than $r_i + d - 2\tau = r_{i-2} + d$. In this case, all tasks before $i-1$ have reached their deadline earlier, and the processor cannot be available for task $i-1$ exactly at that time.

$$P_{i+1,1}^{avail}(t) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{s_{\max}}{q}\}} \mathbb{P}(X = t+\tau-sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{s_{\max}}{q}\}} p_{t+\tau-sq} P_{i-1,1}^{avail}(sq)$$

$s_{\max} - \tau + q \leq t \leq d - 2\tau$**.** In this variant, both situations described above are possible. Thus, we need to add a term to the equation of the first variant, which corresponds to the following case: the processor is available for task $i-1$ at time $r_{i-1} + t + \tau = r_i + t$ and the task is killed because $s_{\max}$ is already passed. The processor is thus available for task i.

$$\begin{aligned} P_{i+1,1}^{avail}(t) &= P_{i,1}^{avail}(t+\tau) + \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{s_{\max}}{q}\}} \mathbb{P}(X = t+\tau-sq) P_{i-1,1}^{avail}(sq) \\ &= P_{i,1}^{avail}(t+\tau) + \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{s_{\max}}{q}\}} p_{t+\tau-sq} P_{i-1,1}^{avail}(sq) \end{aligned}$$

Case $P_{i,1}^{avail}(d-\tau)$. Similarly as for the NeverKill heuristic, at time $r_i + d - \tau = r_{i-1} + d$, task $i-1$ reaches its deadline and will be stopped whether it is completed or not. The difference is that, task $i-1$ cannot be started after time $r_{i-1} + s_{\max} = r_i + s_{\max} - \tau$. Thus, we consider possible starting times for task $i-1$ and the probability that it is executed until the deadline.

$$P_{i,1}^{avail}(d-\tau) = \sum_{s=0}^{\frac{s_{\max}}{q}} \mathbb{P}(X \geq d - sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\frac{s_{\max}}{q}} \left(1 - \sum_{k=1}^{\frac{d}{q}-s-1} p_{kq} \right) P_{i-1,1}^{avail}(sq)$$

Criterion $l_{\max}$ enabled In this variant, we do not allow a task to be executed for longer than a duration of $l_{\max}$. Recall that we assume $l_{\max} \geq \tau$.

The probability of success is similar to that of NeverKill, but is limited because of $l_{\max}$:

$$P_{i,1}^{comp}(t) = \mathbb{P}(X \leq \min\{d-t, l_{\max}\}) = \sum_{k=1}^{\min\{\frac{d-t}{q}, \frac{l_{\max}}{q}\}} p_{kq}$$

As for the probability of availability, we need to discuss different cases:

Case $P_{i,1}^{avail}(0)$. As we have $l_{\max} \geq \tau$, task $i-1$ cannot be interrupted between time r_{i-1} and r_i. Thus, the only possibility that the processor is available at the release time of task i is that task $i-1$ has successfully completed at the latest at that time.

$$P_{i,1}^{avail}(0) = \sum_{s=0}^{\frac{\tau}{q}-1} \mathbb{P}(X \leq \tau - sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\frac{\tau}{q}-1} \left(\sum_{k=1}^{\frac{\tau}{q}-s} p_{kq} \right) P_{i-1,1}^{avail}(sq)$$

Case $P_{i,1}^{avail}(t)$ with $q \leq t \leq d-\tau-q$. For the processor to be available at the time $r_i + t = r_{i-1} + \tau + t$, the execution of task $i-1$ must complete exactly at that time or task $i-1$ must be killed at that time because of $l_{\max}$. The processing of task $i-1$ starts at a time $r_{i-1} + sq$ where $0 \leq sq \leq d-\tau$, and where $r_{i-1} + sq < r_i + t = r_{i-1} + \tau + t$, and where $t + \tau - sq \leq l_{\max}$ because the execution of task $i-1$ lasts at most $l_{\max}$. Altogether, this gives us the constraints:

$$\max\{0, t+\tau-l_{\max}\} \leq sq \leq \min\{d-\tau, \tau+t-1\}.$$

We have two cases to consider:

$q \leq t \leq l_{\max} - \tau - q$**.** In this case, task $i-1$ cannot be killed because of the $l_{\max}$ limit. This is because, if $t < l_{\max} - \tau$, the processor will be available for task i strictly earlier than $r_i + l_{\max} - \tau = r_{i-1} + l_{\max}$. At that time, task $i-1$ is generated for strictly less than $l_{\max}$ time units. Thus, it can not be interrupted because of the upper bound of execution time $l_{\max}$. Hence, we only need to consider the case that task $i-1$ is completed exactly at that time, as in the case of NeverKill.

$$P_{i,1}^{avail}(t) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} \mathbb{P}(X = t+\tau-sq) P_{i-1,1}^{avail}(sq) = \sum_{s=0}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} p_{t+\tau-sq} P_{i-1,1}^{avail}(sq)$$

$l_{\max} - \tau \leq t \leq d - \tau - q$**.** In this case, the second situation described above is also possible: if task $i-1$ is started at time $r_{i-1} + t + \tau - l_{\max}$ and is not completed before $r_{i-1} + t + \tau = r_i + t$, it will be killed, and the processor will be available for task i at that time.

$$\begin{aligned} P_{i,1}^{avail}(t) &= \mathbb{P}(X \geq l_{\max}) P_{i-1,1}^{avail}(t+\tau-l_{\max}) + \sum_{s=\frac{t+\tau-l_{\max}}{q}+1}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} \mathbb{P}(X = t+\tau-sq) P_{i-1,1}^{avail}(sq) \\ &= \left(1 - \sum_{k=1}^{\frac{l_{\max}}{q}-1} p_{kq}\right) P_{i-1,1}^{avail}(t+\tau-l_{\max}) + \sum_{s=\frac{t+\tau-l_{\max}}{q}+1}^{\min\{\frac{t+\tau}{q}-1, \frac{d-\tau}{q}\}} p_{t+\tau-sq} P_{i-1,1}^{avail}(sq) \end{aligned}$$

Case $P_{i,1}^{avail}(d-\tau)$**.** At time $r_i + d - \tau = r_{i-1} + d$, task $i-1$ reaches its deadline and will be stopped whether it is completed or not. With the limit of $l_{\max}$, task $i-1$ must be started after time $r_{i-1} + d - l_{\max}$. Otherwise, it is already interrupted earlier. Thus, we just consider all possible starting times for task $i-1$ and the probability that it is executed until the deadline.

$$P_{i,1}^{avail}(d-\tau) = \sum_{s=\frac{d-l_{\max}}{q}}^{\frac{d-\tau}{q}} \mathbb{P}(X \geq d - sq) P_{i-1,1}^{avail}(sq) = \sum_{s=\frac{d-l_{\max}}{q}}^{\frac{d-\tau}{q}} \left(1 - \sum_{k=1}^{\frac{d}{q}-s-1} p_{kq}\right) P_{i-1,1}^{avail}(sq)$$

Criterion $d_{\max}$ **enabled** This variant is very similar to NeverKill. We can obtain each equation by replacing d in the case of NeverKill by $d_{\max}$.

In the following sections, we will at first present the establishment of the linear system (and the transition matrix) based on the above equations, which is used to find the asymptotic result of probabilities of availability. After that, for multi-criteria enabled cases, we will use a simple example to show that the transition matrix of these cases can be easily built from that when single criterion is enabled.

Building the linear system

In order to find the asymptotic behavior of $P_{i,1}^{avail}(t)$ for each time t (i.e., $i \to \infty$), we use the equations above to establish a system of linear equations. We consider how the probabilities evolve from the execution of the $(i-1)$-th task to the execution of the i-th. Hence, we will express $P_{i,1}^{avail}(0), ..., P_{i,1}^{avail}(d-\tau)$ as functions of $P_{i-1,1}^{avail}(0), ..., P_{i-1,1}^{avail}(d-\tau)$. Note that at time $d-\tau$ after the release of task i, task $i-1$ reaches its deadline and the processor becomes available anyway.

We define as following the column vector of probability of availability for task i:

$$\pi_i = \begin{bmatrix} P_{i,1}^{avail}(0) \\ P_{i,1}^{avail}(q) \\ P_{i,1}^{avail}(2q) \\ ... \\ P_{i,1}^{avail}(d-\tau) \end{bmatrix}$$

We use the matrix A such that $A_{x,y} = \mathbb{P}(s_{i,1} = xq | s_{i-1,1} = yq)$ (probability that the availability time of m_1 equals to x quanta for task i knowing that the availability time equals to y quanta for task $i-1$). Note that A is the transpose of the transition matrix of the Markov chain, and we use it because it is more handy to deal with column vectors rather than with row vectors. We will end up with a system of the form $\pi_i = A \times \pi_{i-1}$, more precisely:

$$\begin{bmatrix} P_{i,1}^{avail}(0) \\ P_{i,1}^{avail}(q) \\ P_{i,1}^{avail}(2q) \\ ... \\ P_{i,1}^{avail}(d-\tau) \end{bmatrix} = A \times \begin{bmatrix} P_{i-1,1}^{avail}(0) \\ P_{i-1,1}^{avail}(q) \\ P_{i-1,1}^{avail}(2q) \\ ... \\ P_{i-1,1}^{avail}(d-\tau) \end{bmatrix}$$

In order to compute the asymptotic probabilities of availability, we should solve the linear system:

$$\begin{cases} \pi_\infty = A \times \pi_\infty \\ \sum_{k=0}^{\frac{d-\tau}{q}} P_{\infty,1}^{avail}(kq) = 1 \end{cases}$$

where

$$\pi_\infty = \begin{bmatrix} P^{avail}_{\infty,1}(0) \\ P^{avail}_{\infty,1}(q) \\ P^{avail}_{\infty,1}(2q) \\ \dots \\ P^{avail}_{\infty,1}(d-\tau) \end{bmatrix}$$

is the vector of asymptotic probabilities of availability. After that, we can calculate the asymptotic probability of success of our tasks.

Special case of multi-criteria heuristics

In this section, we use a simple example to present the building of matrix A while using a multi-criteria heuristics.

We set the parameters as following: $q = 1$, $\tau = 4$, $d = 12$, $s_{\max} = 5$, $l_{\max} = 6$, $d_{\max} = 10$.

At first, here is the matrix A for the NeverKill heuristic. The first row of the matrix contains elements for $P^{avail}_{i,1}(0)$, until the last row for $P^{avail}_{i,1}(d-\tau)$.

$$\begin{bmatrix}
\sum_{k=1}^{4} p_k & \sum_{k=1}^{3} p_k & \sum_{k=1}^{2} p_k & p_1 & 0 & 0 & 0 & 0 & 0 \\
p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 & 0 \\
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 \\
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 \\
p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 \\
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 \\
p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 \\
p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 \\
1-\sum_{k=1}^{11} p_k & 1-\sum_{k=1}^{10} p_k & 1-\sum_{k=1}^{9} p_k & 1-\sum_{k=1}^{8} p_k & 1-\sum_{k=1}^{7} p_k & 1-\sum_{k=1}^{6} p_k & 1-\sum_{k=1}^{5} p_k & 1-\sum_{k=1}^{4} p_k & 1-\sum_{k=1}^{3} p_k
\end{bmatrix}$$

The matrix A of $s_{\max}$ enabled case has exactly the same first $1+\frac{s_{\max}}{q}$ columns as the matrix for NeverKill. The remaining $\frac{d-\tau-s_{\max}}{q}$ columns only contain zeros except for the elements $A_{s-\frac{\tau}{q},s}$, for $q + s_{\max} \leq sq \leq d - \tau$, which are all equal to 1. These values represent the cases that task $i-1$ is not launched at such a start time sq and the processor is given immediately to task i for which it is $\frac{\tau}{q}$ quanta "sooner" with respect to its release time.

$$\begin{bmatrix}
\sum_{k=1}^{4} p_k & \sum_{k=1}^{3} p_k & \sum_{k=1}^{2} p_k & p_1 & 0 & 0 & 0 & 0 & 0 \\
p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 & 0 \\
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 1 & 0 & 0 \\
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & 0 & 1 & 0 \\
p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & 0 & 0 & 1 \\
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & 0 & 0 & 0 \\
p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & 0 & 0 & 0 \\
p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & 0 & 0 & 0 \\
1-\sum_{k=1}^{11} p_k & 1-\sum_{k=1}^{10} p_k & 1-\sum_{k=1}^{9} p_k & 1-\sum_{k=1}^{8} p_k & 1-\sum_{k=1}^{7} p_k & 1-\sum_{k=1}^{6} p_k & 0 & 0 & 0
\end{bmatrix}$$

Below is the matrix A in the case when $l_{\max}$ is enabled. Let A' be the matrix of NeverKill under the same parameter values. Recall that $A_{x,y}$ is the probability that the availability time

equals to $r_i + xq$ for task i knowing that the availability time equals to $r_{i-1} + yq$ for task $i-1$. We have three cases to consider:

- For $y \geq x + \frac{\tau - l_{\max}}{q} + 1$, we can deduce that $(r_i + xq) - (r_{i-1} + yq) < l_{\max}$. This means that the difference between the available times of tasks i and $i-1$ is smaller than $l_{\max}$. In this case, the execution time of $i-1$ is less than $l_{\max}$, and it will not be interrupted. The only possible situation is, as in NeverKill, that task $i-1$ has exactly finished at that time. Hence, we have $A_{x,y} = A'_{x,y}$.
- Similarly, for $y = x + \frac{\tau - l_{\max}}{q}$, we have $(r_i + xq) - (r_{i-1} + yq) = l_{\max}$. This means that task $i-1$ will be interrupted at that time, even if it is not successfully finished. Thus, the probabilities of cases that the execution time of $i-1$ is larger than $l_{\max}$ in NeverKill are all summed up in this term while enabling $l_{\max}$. Thus we can deduce that, $A_{x,y} = 1 - \sum_{y'=0}^{y-1} A'_{x,y'}$.
- Otherwise, the difference between the available times of tasks i and $i-1$ is larger than $l_{\max}$, processor cannot be available for task i exactly at that time, because $i-1$ must be successfully completed or interrupted earlier, thus $A_{x,y} = 0$.

$$\begin{bmatrix}
\sum_{k=1}^{4} p_k & \sum_{k=1}^{3} p_k & \sum_{k=1}^{2} p_k & p_1 & 0 & 0 & 0 & 0 & 0 \\
p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 & 0 \\
1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 \\
0 & 1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 \\
0 & 0 & 1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & p_1 & 0 \\
0 & 0 & 0 & 1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & p_1 \\
0 & 0 & 0 & 0 & 1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 \\
0 & 0 & 0 & 0 & 0 & 1 - \sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 - \sum_{k=1}^{5} p_k & 1 - \sum_{k=1}^{4} p_k & 1 - \sum_{k=1}^{3} p_k
\end{bmatrix}$$

The matrix A when $d_{\max}$ is enabled has exactly the same first $\frac{d_{\max} - \tau}{q}$ rows as the matrix for NeverKill. For the $\frac{d_{\max} - \tau}{q} + 1$-th row, we have $xq = d_{\max} - \tau$. This is the case that processor is available for task i at time $r_i + d_{\max} - \tau = r_{i-1} + d_{\max}$, where task $i-1$ should be interrupted by $d_{\max}$, and all available times larger than this value are not possible. Thus, we can deduce that $A_{\frac{d_{\max} - \tau}{q}, y} = 1 - \sum_{y'=1}^{\frac{d_{\max} - \tau}{q}} A_{\frac{d_{\max} - \tau}{q}, y'}$, and the remaining $\frac{d - d_{\max}}{q}$ rows only contain zeros.

$$\begin{bmatrix}
\sum_{k=1}^{4} p_k & \sum_{k=1}^{3} p_k & \sum_{k=1}^{2} p_k & p_1 & 0 & 0 & 0 & 0 & 0 \\
p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 & 0 \\
p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 \\
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 \\
p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & 0 \\
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 \\
1-\sum_{k=1}^{9} p_k & 1-\sum_{k=1}^{8} p_k & 1-\sum_{k=1}^{7} p_k & 1-\sum_{k=1}^{6} p_k & 1-\sum_{k=1}^{5} p_k & 1-\sum_{k=1}^{4} p_k & 1-\sum_{k=1}^{3} p_k & 1-\sum_{k=1}^{2} p_k & 1-p_1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

As for the case where the three criteria are all enabled ($s_{\max}$, $l_{\max}$ and $d_{\max}$), it is easy to find the corresponding matrix by composing those of the above heuristics. At first, we can start from the matrix of $l_{\max}$ enabled case. And then, we replace the last $\frac{d-\tau-s_{\max}}{q}$ columns by that of $s_{\max}$. Finally, as in the case when $d_{\max}$ is enabled, we replace the last $\frac{d-d_{\max}}{q}$ rows by 0, and the $\frac{d_{\max}-\tau}{q}+1$-th row ($x=\frac{d_{\max}-\tau}{q}$) is calculated as following: $A_{x,y}=1-\sum_{y'=1}^{\frac{d_{\max}-\tau}{q}} A_{x,y'}$. And we can obtain the matrix:

$$\begin{bmatrix}
\sum_{k=1}^{4} p_k & \sum_{k=1}^{3} p_k & \sum_{k=1}^{2} p_k & p_1 & 0 & 0 & 0 & 0 & 0 \\
p_5 & p_4 & p_3 & p_2 & p_1 & 0 & 0 & 0 & 0 \\
1-\sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & p_1 & 1 & 0 & 0 \\
0 & 1-\sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & p_2 & 0 & 1 & 0 \\
0 & 0 & 1-\sum_{k=1}^{5} p_k & p_5 & p_4 & p_3 & 0 & 0 & 1 \\
0 & 0 & 0 & 1-\sum_{k=1}^{5} p_k & p_5 & p_4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1-\sum_{k=1}^{5} p_k & 1-\sum_{k=1}^{4} p_k & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

For other cases where we enable only two of the criteria, we can proceed similarly as for this example, and just skip the criteria disabled in each case.

Multi-processor case

Recall that, under the multiple processor case, for the RoundRobin heuristic, task i will be executed by processor $1+(i \mod M)$. Thus, we can treat the M processors fully independently. Let us consider a case with period τ, deadline d, and the distribution of execution times $\mathcal{D}$. We can simply consider each processor as a system with period $M\tau$, deadline d, and distribution $\mathcal{D}$. And then, we can calculate the asymptotic success probability of each task according to its attributed processor. Finally, we can compute the performance by summing up the success probability of all tasks.

4.4.2 EarliestStartTime strategy

For EarliestStartTime strategy, the M processors become dependent. Thus, we need to treat simultaneously the states of all the processors while considering any task.

Recall that $s_{i,j}$ is the time at which machine m_j will be available to start task i. For the EarliestStartTime heuristic, task i will be executed by processor $h(i)$ which satisfies $s_{i,h(i)} = \min_{1 \leq j \leq M} s_{i,j}$.

As mentioned above, we can define the state of the system at the release time of task i by a list of the available times of all the processors: $\mathcal{S}_i = (s_{i,1}, ..., s_{i,M})$. We need to determine every possible state list $\mathcal{S}_i$ and the probability that it is reached, by considering $\mathcal{S}_{i-1}$, the state list at the release time of task $i-1$.

Equations

We assume that task $i-1$ is allocated to processor $m_{h(i-1)}$. Thus, for any processor m_j such that $j \neq h(i-1)$, the available time of the processor is not changed before and after the releasing of task $i-1$. Therefore, these processors will be available for task i at time $r_{i-1} + s_{i-1,j} = r_i + s_{i-1,j} - \tau$. Therefore:

$$s_{i,j} = \max\{0, s_{i-1,j} - \tau\} \text{ if } j \neq h(i-1).$$

As for processor $m_{h(i-1)}$, we will consider each of our task admission criteria differently.

NeverKill For processor $m_{h(i-1)}$, there exists two cases: (i) with probability p_t, the execution of task $i-1$ will last for a duration of t where $q \leq t \leq d - s_{i-1,h(i-1)}$. In this case, task $i-1$ will be completed successfully at time $s_{i-1,h(i-1)} + t$. Hence, processor $m_{h(i-1)}$ will be available for task i at time $s_{i,h(i-1)} = \max\{0, s_{i-1,h(i-1)} + t - \tau\}$ with probability p_t. (ii) if the execution time of task $i-1$ is greater than $d - s_{i-1,h(i-1)}$, it will be interrupted by the deadline at time $r_{i-1} + d = r_i + d - \tau$. This happens with probability $1 - \sum_{k=1}^{\frac{d - s_{i-1,h(i-1)}}{q}} p_{kq}$, and the available time for task i can be defined by $s_{i,h(i-1)} = d - \tau$.

Altogether, we get:

$$s_{i,h(i-1)} = \begin{cases} \max\{0, s_{i-1,h(i-1)} + t - \tau\} \text{ with probability } p_t, \text{ for } q \leq t \leq d - s_{i-1,h(i-1)} \\ d - \tau \text{ with probability } 1 - \sum_{k=1}^{\frac{d - s_{i-1,h(i-1)}}{q}} p_{kq} \end{cases}$$

Criterion $s_{\max}$ enabled In this case, task $i-1$ cannot be started later than time $r_{i-1} + s_{\max}$, and we have to consider two situations: (i) the available time for task $i-1$, $s_{i-1,h(i-1)}$, is greater than $s_{\max}$, task $i-1$ is not launched. Thus, the available time of processor $m_{h(i-1)}$ for task i is calculated as other processors not handling task $i-1$. (ii) otherwise, task $i-1$ is started successfully, and $s_{i,h(i-1)}$ can be calculated as for NeverKill. Altogether, we have:

If $s_{i-1,h(i-1)} > s_{\max}$**.**

$$s_{i,h(i-1)} = \max\{0, s_{i-1,h(i-1)} - \tau\}$$

If $s_{i-1,h(i-1)} \leq s_{\max}$**.**

$$s_{i,h(i-1)} = \begin{cases} \max\{0, s_{i-1,h(i-1)} + t - \tau\} \text{ with probability } p_t, \text{ for } q \leq t \leq d - s_{i-1,h(i-1)} \\ d - \tau \text{ with probability } 1 - \sum_{k=1}^{\frac{d - s_{i-1,h(i-1)}}{q}} p_{kq} \end{cases}$$

Criterion $l_{\max}$ enabled In this variant, a task can be interrupted either by the deadline or by the threshold $l_{\max}$. Recall that we assume $\tau < l_{\max} \leq d$. We need to rewrite the equation of NeverKill to take into account $l_{\max}$:

$$s_{i,h(i-1)} = \begin{cases} \max\{0, s_{i-1,h(i-1)} + t - \tau\} \text{ with probability } p_t, \text{ for } q \leq t \leq \min\{d - s_{i-1,h(i-1)}, l_{\max}\} \\ \min\{d - \tau, s_{i-1,h(i-1)} + l_{\max} - \tau\} \text{ with probability } 1 - \sum_{k=1}^{\min\{\frac{d-s_{i-1,h(i-1)}}{q}, \frac{l_{\max}}{q}\}} p_{kq} \end{cases}$$

in which the first term corresponds do the case where task $i-1$ is successfully finished before the deadline and before the $l_{\max}$ upper bound, and the second term is the case that task $i-1$ is interrupted by the deadline or the $l_{\max}$ threshold.

Criterion $d_{\max}$ enabled This variant is very similar to NeverKill, we just need to replace d by $d_{\max}$, because we interrupt uncompleted tasks at time $d_{\max}$ instead of at their deadline.

$$s_{i,h(i-1)} = \begin{cases} \max\{0, s_{i-1,h(i-1)} + t - \tau\} \text{ with probability } p_t, \text{ for } q \leq t \leq d_{\max} - s_{i-1,h(i-1)} \\ d_{\max} - \tau \text{ with probability } 1 - \sum_{k=1}^{\frac{d_{\max} - s_{i-1,h(i-1)}}{q}} p_{kq} \end{cases}$$

Probability of states changing

In this section, we enable all three task attribution criteria ($s_{\max}$, $l_{\max}$, $d_{\max}$). We can consider other cases as a special case of this one, in which disabled criteria are set to their maximal value (d for $l_{\max}$ and $d_{\max}$; $d - \tau$ for $s_{\max}$).

As mentioned above, the state of the system at the release time of task i can be defined by an available time state list of all processors: $\mathcal{S}_i = (s_{i,1}, ..., s_{i,M})$. We will calculate the probability that each state list $\mathcal{S}_i$ is reached, based on $\mathcal{S}_{i-1}$, the state list of task $i-1$. Note that the following two states: $\mathcal{S} = (s_1, s_2, ..., s_M)$ and $\mathcal{S}' = (s_M, s_{M-1}, ..., s_1)$, are equivalent and indistinguishable. Therefore, we can merge the $M!$ equivalent configurations, identical up to the ordering of processors, to reduce the computational cost of the implementation.

Thus, for task $i-1$, we only need to consider the state lists $\mathcal{S}_{i-1} = (s_{i-1,1}, ..., s_{i-1,M})$ which are lexicographically ordered: $s_{i-1,1} \leq s_{i-1,2} \leq ... \leq s_{i-1,M}$. According to the definition of EarliestStartTime, task $i-1$ will be attributed to processor m_1. In order to facilitate the representation, we define a function *Sorted* which has as input a state list, and has as output the non-decreasing list with the same elements as its input.

All possible available time state lists and the probability that they are reached are presented as following:

- If $s_{i-1,1} > s_{\max}$, task $i-1$ is not started because of the $s_{\max}$:

$$\mathcal{S}_i = (s_{i-1,1} - \tau, s_{i-1,2} - \tau, ..., s_{i-1,M} - \tau) \text{ with probability } 1.$$

- If $s_{i-1,1} \leq s_{\max}$, task $i-1$ will be started, and we have the following different cases:
  - Task $i-1$ is successfully finished before the release of task i:

* If $s_{i-1,1} < \tau$,

$$\mathcal{S}_i = (0, s_{i-1,2}-\tau, ..., s_{i-1,M}-\tau) \text{ with probability } \sum_{k=1}^{\min\{\frac{\tau-s_{i-1,1}}{q}, \frac{d_{\max}-s_{i-1,1}}{q}, \frac{l_{\max}}{q}\}} p_{kq}$$

* If $s_{i-1,1} \geq \tau$,

$$\mathcal{S}_i = (0, s_{i-1,2} - \tau, ..., s_{i-1,M} - \tau) \text{ with probability } 0$$

– Task $i-1$ is successfully finished after the release of task i. For $\max\{q, \tau - s_{i-1,1}+q\} \leq t \leq \min\{d_{\max} - s_{i-1,1}, l_{\max}\}$:

$$\mathcal{S}_i = \mathit{Sorted}((s_{i-1,1} + t - \tau, s_{i-1,2} - \tau, ..., s_{i-1,M} - \tau)) \text{ with probability } p_t$$

– Task $i-1$ is interrupted either by the $d_{\max}$ or the $l_{\max}$ threshold:

* If $s_{i-1,1} + l_{\max} < d_{\max}$, task $i-1$ will be interrupted by reaching $l_{\max}$:

$$\mathcal{S}_i = \mathit{Sorted}((l_{\max}+s_{i-1,1}-\tau, s_{i-1,2}-\tau, ..., s_{i-1,M}-\tau)) \text{ with probability } 1-\sum_{k=1}^{\frac{l_{\max}}{q}} p_{kq}$$

* If $s_{i-1,1} + l_{\max} \geq d_{\max}$, task $i-1$ will be interrupted by reaching the $d_{\max}$:

$$\mathcal{S}_i = \mathit{Sorted}((d_{\max}-\tau, s_{i-1,2}-\tau, ..., s_{i-1,M}-\tau)) \text{ with probability } 1-\sum_{k=1}^{\frac{d_{\max}-s_{i-1,1}}{q}} p_{kq}$$

- All other states $\mathcal{S}_i$ cannot be reached.

4.4.3 Limit behavior of the Markov chain

In this section we will prove that the Markov chains that we consider are both irreducible and aperiodic. Since their number of states is always finite, this is equivalent to say that these Markov chains are regular. It is well known that a regular finite Markov chain admits a unique limit distribution [41].

For the sake of readability, we will restrict the proof to the case where there is a single processor. The multi-processor case is quite similar.

In this section, all variables related to time are assumed to be integer multiples of the quantum duration q. On the other hand, we assume that all possible execution times for a task do actually happen. That is, for any duration $l \in [q; d]$, $p_l > 0$.

In the following, s will denote the state when the considered task i arrives. That is, the processor will be available for task i at time r_i+s. Then t denote the time at which the processor will be available for task $i+1$. We are going to first recall a few remarks on the values of $s_{\max}$, $l_{\max}$ and $d_{\max}$, and will deal with two degenerate cases.

We have already assumed in Sections 4.2 that the deadline is strict larger than the period, that is $d > \tau$. We also discussed in Section 4.3 the range that each task admission criteria can take value: $s_{\max}$ can take value in the range $[0; d-\tau]$, $l_{\max}$ and $d_{\max}$ can take value in the range $[\tau + q; d]$.

We now show that all states are reachable. The latest possible start time for task i is $r_i + s_{\max}$. Then its execution for last at the longest either for a time $l_{\max}$ or until time $r_i + d_{\max}$, whichever is reached first. Therefore, the processor will be available for task $i+1$ at the latest at time $\max\{0, \min\{s_{\max} + l_{\max}, d_{\max}\} - \tau\} = \min\{s_{\max} + l_{\max}, d_{\max}\} - \tau$. Therefore, the range of (theoretically) possible states is $[0; \min\{s_{\max} + l_{\max}, d_{\max}\} - \tau]$. Let us "forget" about $d_{\max}$ for a short while. The processor is available for task i at time $r_i + s$. Then, because $l_{\max} > \tau$ and $p_{\tau+q} > 0$, then with probability $p_{\tau+q}$ the execution of task i will last for a time $\tau + q$ and the processor will be available for task $i+1$ at time $r_i + s + \tau + q = r_{i+1} + s + q$. Therefore, if the state of the system is s for task i, it will be in state $s + q$ for task $i+1$ with probability $p_{\tau+q} > 0$. This is true as long as $s \leq s_{\max}$ and $s + \tau + q \leq d_{\max}$.

Let us first consider the case $s+\tau+q = d_{\max}$. If we reached it, this means that $s_{\max}+l_{\max} \geq d_{\max}$, because $s_{\max} \geq s$ and $l_{\max} \geq \tau + q$. Therefore, the possible range of values is $[0; d_{\max} - \tau]$, and we have established that all these states are reachable. Now consider the other case. Namely, we reached the case $s = s_{\max}$ without, in the process, having tasks interrupted because of the value of $d_{\max}$. Then, with probability $p_{\tau+j} > 0$, with $j \in [q; \min\{l_{\max}; d_{\max} - s_{\max}\} - \tau]$, the execution of task i lasts until the time $r_i + s_{\max} + \tau + j = r_{i+1} + s_{\max} + j$. Hence, all states in the range $[s_{\max} + q; \min\{s_{\max} + l_{\max}; d_{\max}\} - \tau]$ are reachable. Therefore, all possible states are reachable from the initial state, the state $s = 0$.

Now consider any state s. With probability $p_q > 0$, the next state is $\max\{0, s - \tau\} < s$. Hence, from any state a strictly "smaller" state is reachable. Therefore, there is a path from any state to the state 0. This shows that the Markov chain is irreducible.

There is a loop from state 0 to itself. Indeed, with probability $\sum_{k=1}^{\frac{\tau}{q}} p_{kq} > 0$ a task execution takes no longer than the period τ. Hence, if the state before the execution of the task was 0, this is also the case after its execution. Hence the period of state $s = 0$ is one. And because the chain is irreducible, all states have period one too. Hence the Markov chain is aperiodic, which concludes the proof.

4.5 Performance evaluation

In the experiments, we will test the accuracy of our model and evaluate the performance of different criteria, on either single processor and multiple processors cases. In Section 4.5.1, we describe the parameters and settings used during the experimental campaign. Then we present our results in Section 4.5.2.

4.5.1 Experimental methodology

In this section, we present the parameter sets used in our experiments. The default settings are as follows. For the theoretical Markov chain model, the quantum duration q takes a value between 0.001 and 0.1. On the other hand, for the simulations, the number of tasks released E varies between 10 and 10^7. The task period τ varies between one quantum duration q and 2. Let $N = \frac{d}{\tau}$ be the ratio of deadline d to period τ. N is chosen between 2 and 14. When we present our results in the multiple processors case, we vary our number of processors between 2 and 6.

The standard probability distribution functions used to generate task execution times are almost the same as in Chapter 3. We only add a second Weibull functions with shape $k = 1.5$. As in the previous section, to enable a direct comparison between all different distributions, we choose their parameters so that all distributions achieve a mean equal to 1. But different

Table I: Symbol and parameters for the distributions used in the simulations. (For all distributions μ is the mean and σ the standard deviation, except for the truncated normal and half-normal distributions where μ and σ are the mean and standard deviation of the original normal distribution.)

Symbol	Distribution	Parameters
double_exp(λ_1, λ_2)	Bimodal exponential	$\lambda_1 = \frac{1}{1.005} \approx 0.995, \lambda_2 = \frac{1}{0.995} \approx 1.005$
		$\lambda_1 = \frac{1}{0.1} = 10, \lambda_2 = \frac{1}{1.9} \approx 0.526$
double_truncnorm(μ_1, σ_1, μ_2, σ_2)	Bimodal truncated normal	$\mu_1 = 0.5, \sigma_1 \approx 0.534, \mu_2 = 1, \sigma_2 \approx 1.068$
		$\mu_1 = 0.01, \sigma_1 \approx 0.178, \mu_2 = 1, \sigma_2 \approx 1.782$
exp(λ)	Exponential	$\lambda = 1$
gamma(k, θ)	Gamma	$k = 1, \theta = 1$
		$k = \frac{1}{3} \approx 0.333, \theta = 3$
hnorm(σ)	Half-normal	$\sigma = \sqrt{\frac{\pi}{2}} \approx 1.253$
invgamma(α, β)	Inverse Gamma	$\alpha = \frac{7}{3} \approx 2.333, \beta = \frac{4}{3} \approx 1.333$
lnorm(μ, σ)	Log-normal	$\mu = 1, \sigma = 0.5$
		$\mu = 1, \sigma = 3$
truncnorm(μ, σ)	Truncated normal	$\mu = 0.8, \sigma \approx 0.754$
unif(a, b)	Uniform	$a = 0, b = 2$
weibull(k, λ)	Weibull	$k \approx 0.411, \lambda = \frac{1}{\Gamma\left(1+\frac{1}{k}\right)} \approx 0.324$
weibull(k, λ)	Weibull	$k = 1.5, \lambda = \frac{1}{\Gamma\left(1+\frac{1}{k}\right)} \approx 1.108$

from Chapter 3, we will not add a transition to the execution times in this work. The detailed parameters of the distributions are recalled in Table I.

In addition, we present the performance of our heuristics as percentage of tasks successfully completed on time. Therefore, the larger the performance, the better the heuristic.

4.5.2 Results

We will present our results as following: At first, in Section 4.5.2, we study the size needed to minimize the simulation error due to an imprecise evaluation, and we decide the framework that will be used in the following experiments. After that, Section 4.5.2 is a first study of our algorithm. We are in the single processor case, and we focus on the value of q and on different task admission criteria. We choose in this section the parameter sets for our algorithm which are worth to be analyzed. And then, we evaluate our algorithms in detail in Section 4.5.2. Finally, we pass to the multiple processors case. We present in Section 4.5.2 the study of different task attribution criteria.

Evaluation of accuracy for the framework

In this section, we fix in advance several values of $s_{\max}$. For theoretical model and simulation, we study the execution time and expected performance while varying respectively the quantum duration q and number of tasks generated E. In order to simplify the presentation, we present the results of all parameters (deadline, period, distribution, etc.) in one figure, and the percentiles (and worst-case) are presented.

At first, in Figures 4.1 and 4.2, we can find the execution time needed to obtain the expected performance respectively using the Markov chain theoretical model and the simulations. We can observe that the execution time increases linearly with the scale of the problem. In order

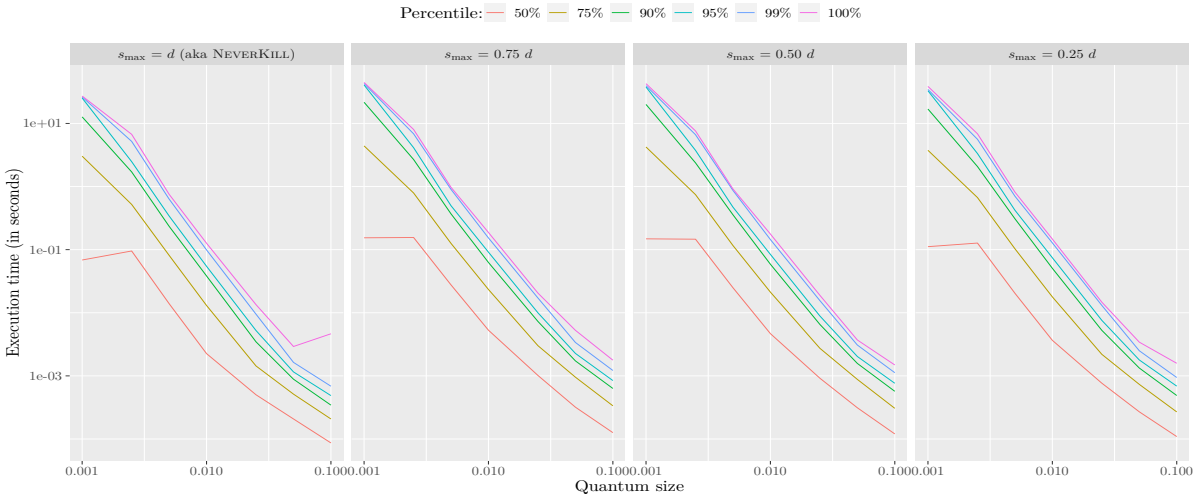


Figure 4.1: Execution times of the theoretical version while fixing different values of $s_{\max}$ for different quantum sizes. For each quantum size and each value of $s_{\max}$, many different periods, deadlines, and distributions are evaluated whose percentiles (and worst-case) are presented.

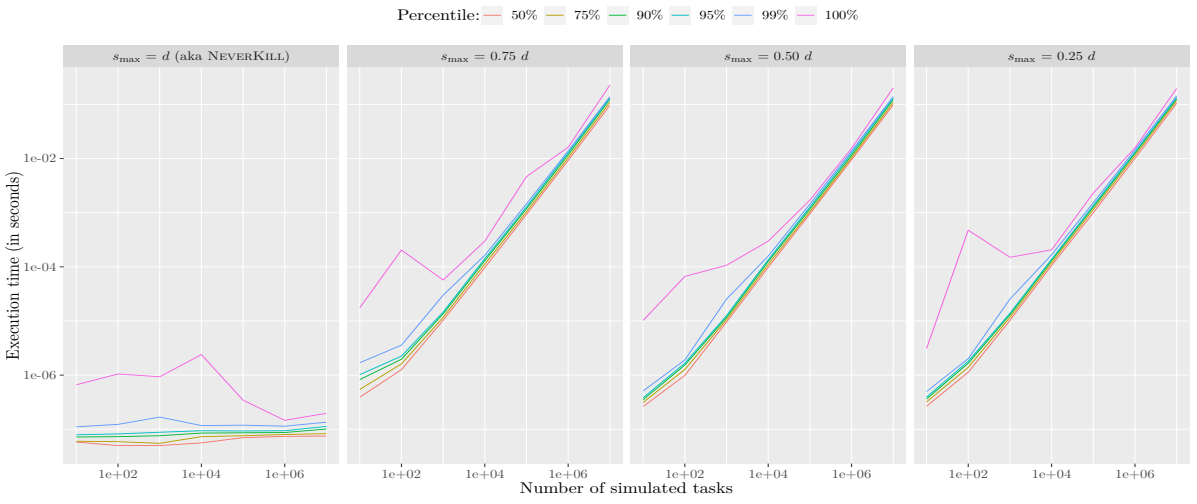


Figure 4.2: Execution times of the simulated version while fixing different values of $s_{\max}$ for different number of simulated tasks. For each number of tasks and each value of $s_{\max}$, many different periods, deadlines, and distributions are evaluated whose percentiles (and worst-case) are presented.

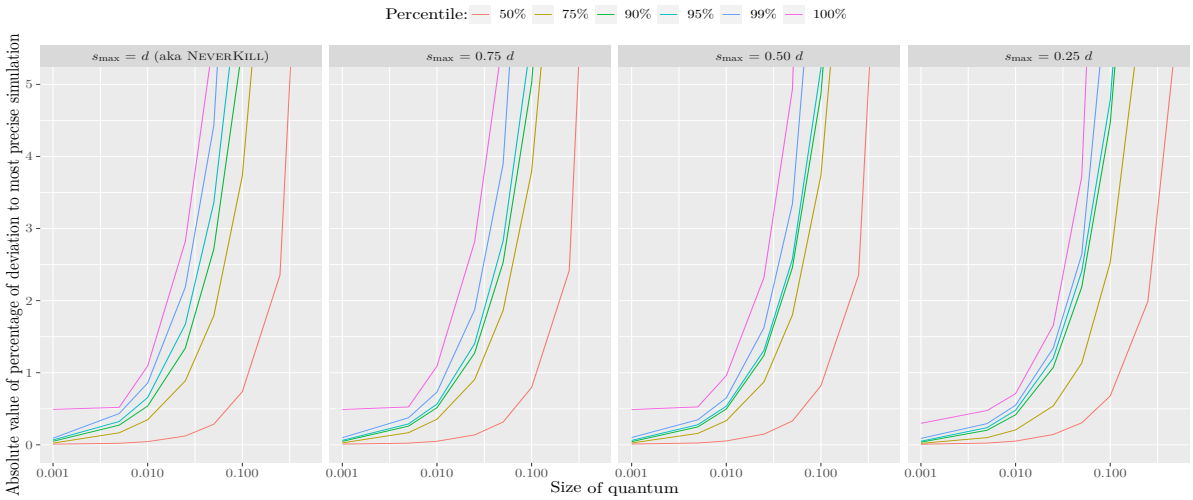


Figure 4.3: Absolute value of percentage of deviation from the performance of the heuristic SMAX, while fixing different values of $s_{\max}$, predicted for different quantum values, to the performance simulated with 10^7 tasks. For each quantum size and each value of $s_{\max}$, many different periods, deadlines, and distributions are evaluated whose percentiles (and worst-case) are presented.

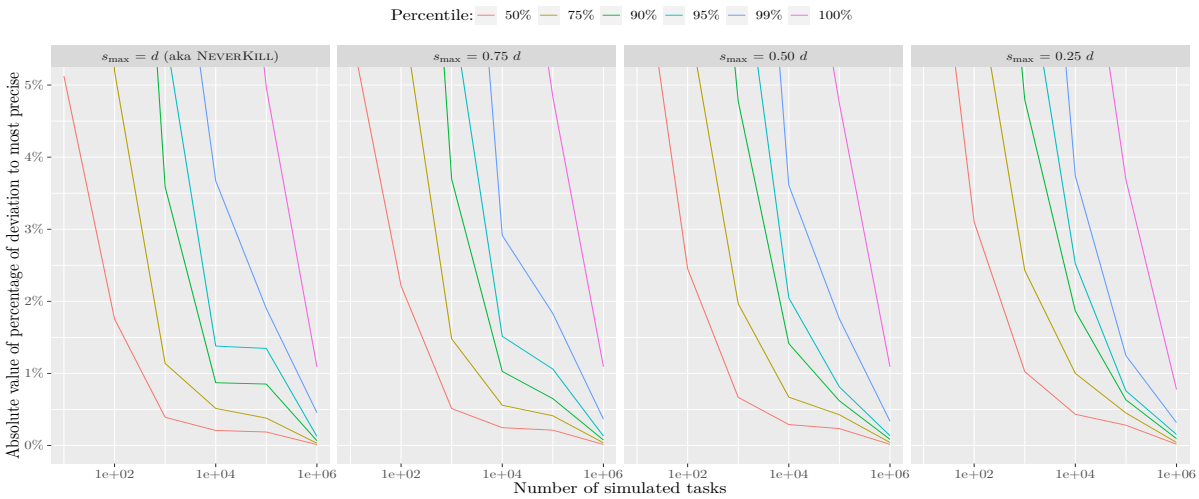


Figure 4.4: Absolute value of percentage of deviation from the performance of the heuristic SMAX, while fixing different values of $s_{\max}$, predicted for different number of tasks simulated, to the performance simulated with 10^7 tasks. For each number of tasks and each value of $s_{\max}$, many different periods, deadlines, and distributions are evaluated whose percentiles (and worst-case) are presented.

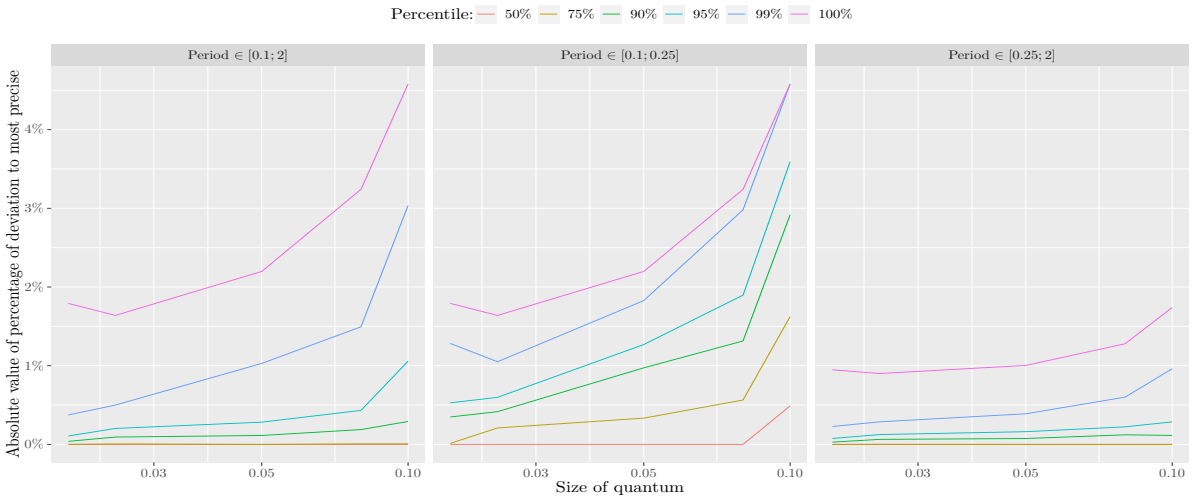


Figure 4.5: Absolute value of percentage of deviation from the performance simulated with 10^6 tasks predicted for heuristic SMAX whose parameter $s_{\max}$ was defined by a Markov chain using a quantum 0.001, when then quantum is larger. For each quantum size, many different periods, deadlines, and distributions are evaluated whose percentiles (and worst-case) are presented.

to establish a relatively correct evaluation framework, we need to find a trade-off between the result precision and the execution time.

Thus, we will compare, for the theoretical model and for simulations, the performance obtained by different values of E or q, in order to find a parameter for which results are accurate enough and obtained relatively quickly. Figures 4.3 and 4.4 presents, respectively for different q and E, the absolute value of relative deviation in performance to the most precise simulation $E = 10^7$. We can find that, $E = 10^6$ has equivalent deviation with $q = 0.01$, and their performance is very close to that of the baseline $E = 10^7$ (under 1% in nearly all cases). As for the execution time, $q = 0.01$ executes 100 times slower than $E = 10^6$ in the worst case. On the other hand, the execution time of the simulations depends only on the number of tasks E, but neither on the deadline d nor on the period τ.

In conclusion, we choose to run simulations with $E = 10^6$ in our following experiments to evaluate the performance of the different heuristics.

Parametrization of the algorithms

The results of our algorithms depend on a variety of parameters. It will be unreadable to present them together on one figure. Thus, in this section, we will do a first summarized study and find the parameters of the algorithms which are worth to be analyzed in detail.

At first, we will use once again SMAX as heuristic. We calculate the best $s_{\max}$ for each value of q, and we compare the performance of these best $s_{\max}$ values while generating a simulation of 10^6 tasks. We choose to use a theoretical model to calculate the best $s_{\max}$ value because, compared to simulations, the theoretical model is more robust. As for simulations, the $s_{\max}$ value found can have a major deviation, especially in the case of a small number of successful tasks. We present in Figure 4.5 the absolute value of relative deviation in performance of different q with that of the most precise case $q = 0.001$. The sub-figure on the left is the summarized result of all parameter sets. We can find that, if we consider $q = 0.1$, the deviation

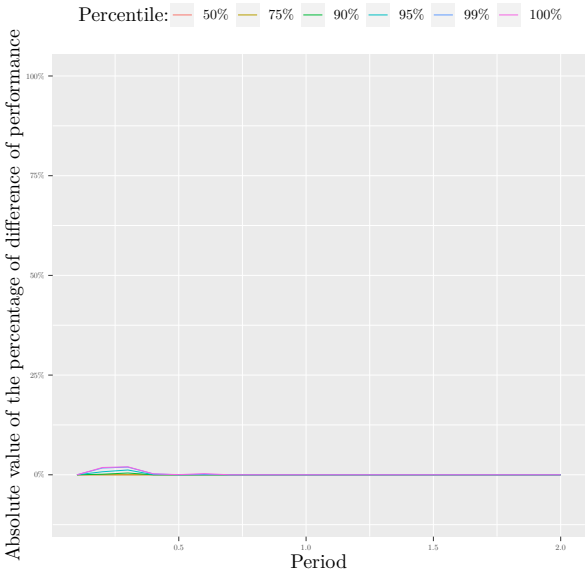


Figure 4.6: Absolute difference in the simulated performance (with 10^6 tasks) of the SMAX and SMAX+LMAX+DMAX heuristics. For each value of the period τ, the main percentiles (and worst-case) are presented.

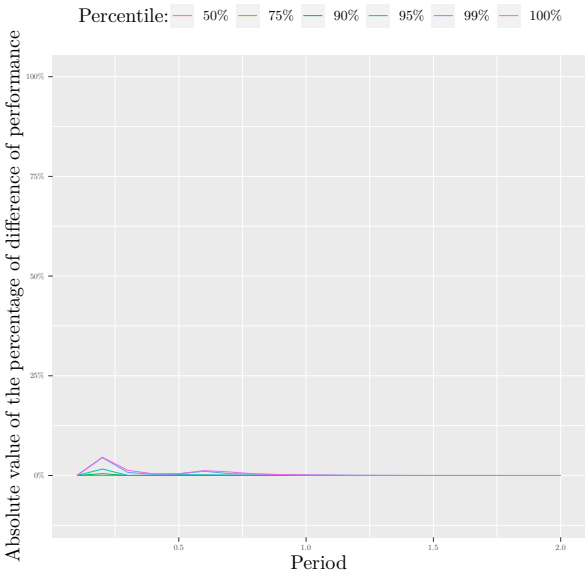


Figure 4.7: Absolute difference in the simulated performance (with 10^6 tasks) of the LMAX+DMAX and NEVERKILL heuristics. For each value of the period τ, the main percentiles (and worst-case) are presented.

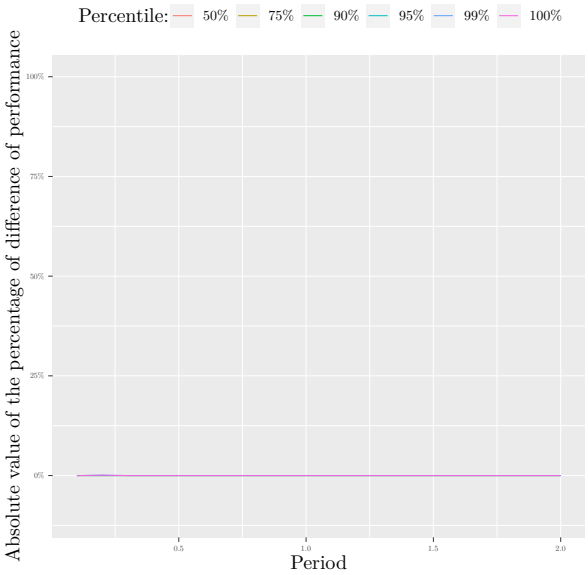


Figure 4.8: Absolute difference in the simulated performance (with 10^6 tasks) of the SMAX and BINSMAX heuristics. For each value of the period τ, the main percentiles (and worst-case) are presented.

is under 5% even in the worst case. In addition, according to the two other sub-figures, we can find that this worst case appears when the period τ is quite small (i.e.,the system is severely over-loaded). If we consider only period parameter between 0.25 and 2, we can find a worst case deviation of only 1.7%. Thus, we can conclude that $q = 0.1$ has larger deviation when period decreases, but the performance is still close to $q = 0.001$. It is a value acceptable considering both the execution time and the precision.

After fixing the q value to 0.1, we do the first study of different task admission criteria. Figures 4.6 and 4.7 show respectively the difference of performance between SMAX and SMAX+LMAX+DMAX (Figure 4.6) and between LMAX+DMAX and NEVERKILL (Figure 4.7). We vary only the period in the figures and we summarize all other parameters together, because we found that the difference of performance varies only with the period. We can observe that, in both comparisons, there is little difference (under 5% in the worst case) when the period is small. In other cases, the difference is almost equals to 0. Thus we can conclude that, SMAX has a performance similar to SMAX+LMAX+DMAX, while NEVERKILL has a performance similar to LMAX+DMAX. We can deduce that, LMAX and DMAX have almost no effect in our algorithms, and we will focus on SMAX and NEVERKILL in the following simulations.

Task admission criteria with single processor

In this section, we compare the performance of the chosen task admission heuristics (SMAX and NEVERKILL), while varying other parameters. For SMAX heuristic, similar to the experiments above, we use the theoretical model with $q = 0.1$ to find the best $s_{\max}$.

While researching for the best $s_{\max}$, as we vary only one criterion in this case, instead of iterating all possible values, we use binary search presented in Algorithm 2 which can significantly reduce the execution time. We name this variant of heuristic BINSMAX. Figure 4.8 shows that the difference between SMAX and BINSMAX is almost equals to 0 (under 0.1% in the worst case)

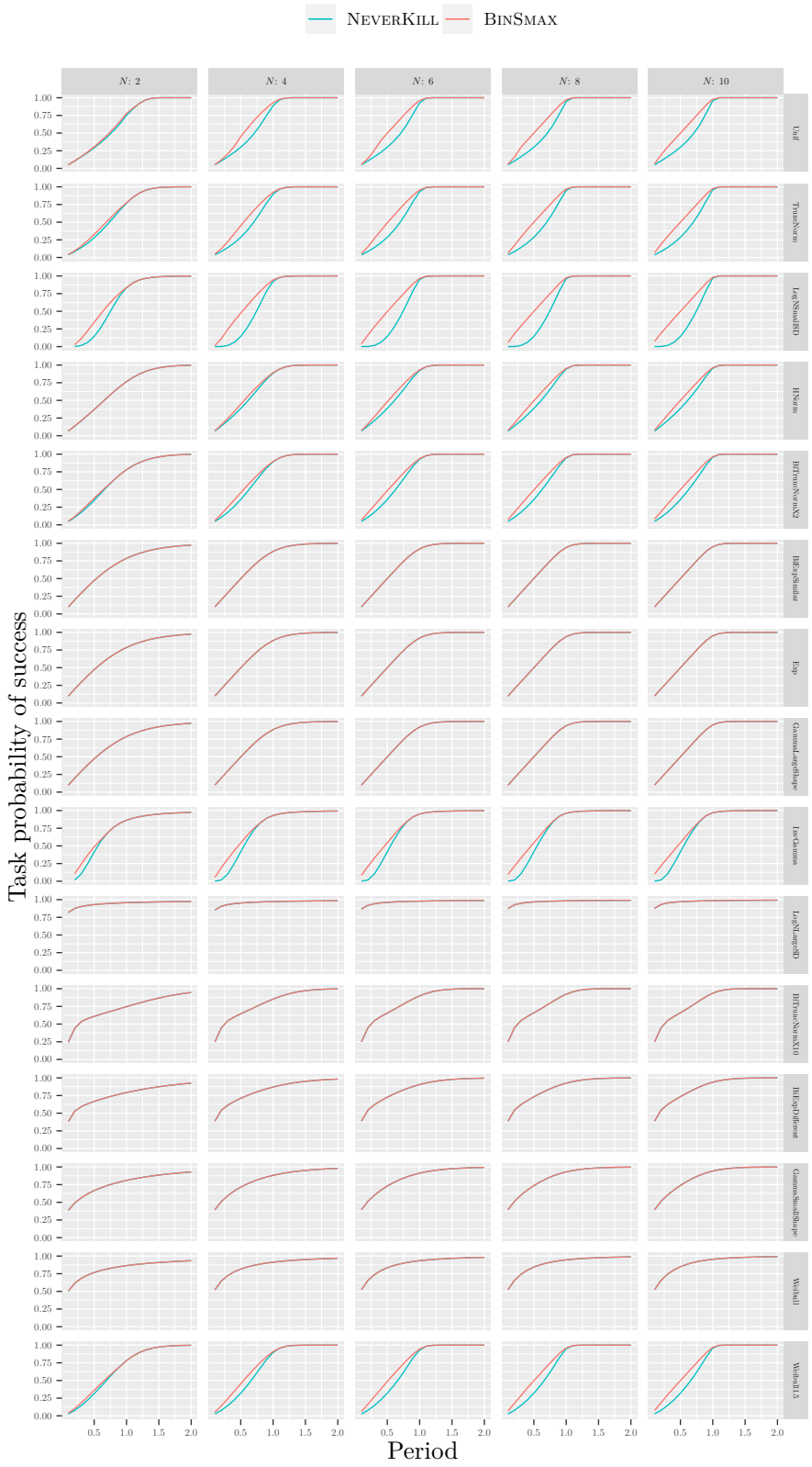


Figure 4.9: Absolute performance of task admission heuristics BinSmax and NeverKill when the quantum length is 0.1 time unit. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.

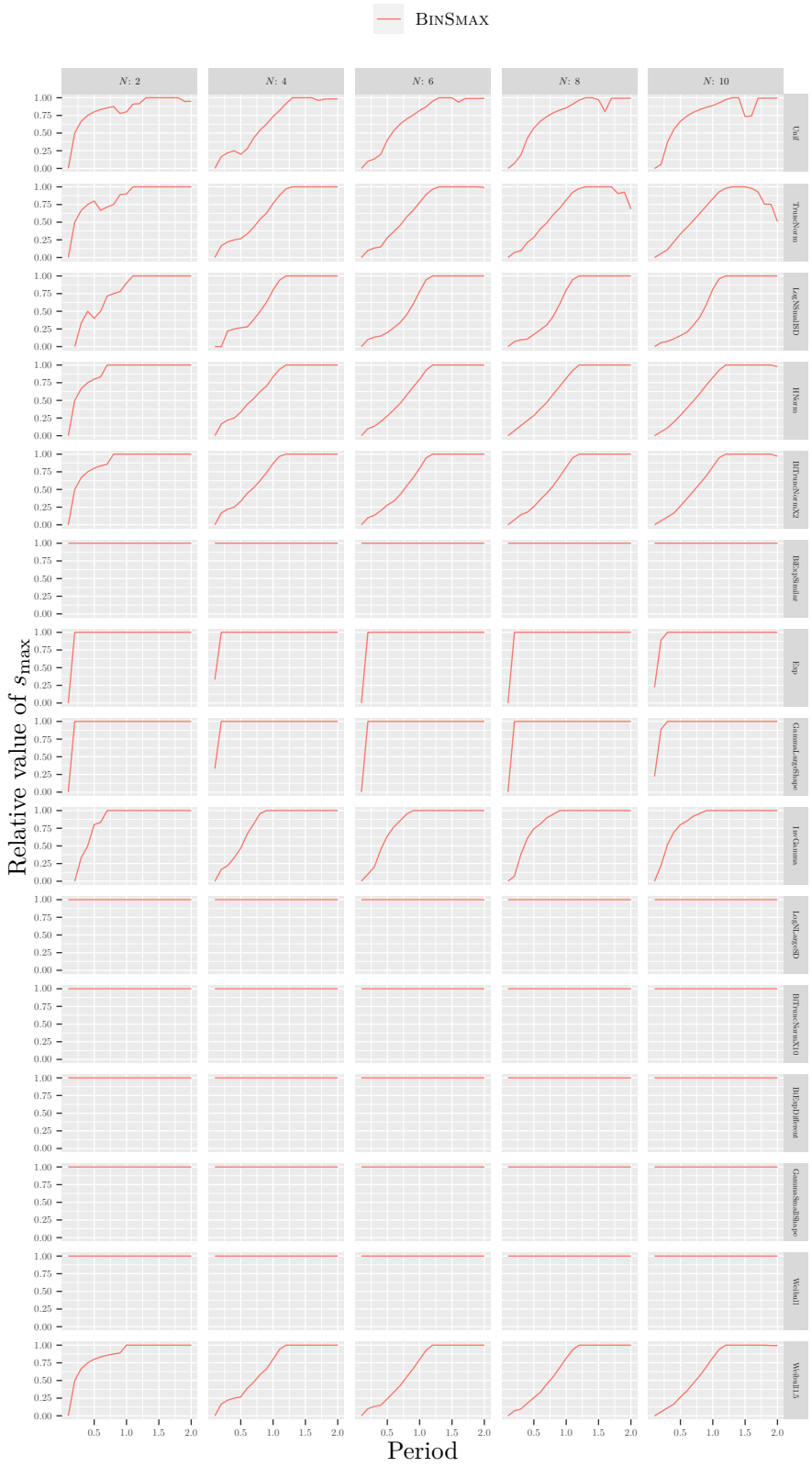


Figure 4.10: Best value for the $s_{\max}$ threshold for task admission heuristic BinSmax expressed as a fraction of the maximum meaningful value $d - \tau$, when the quantum length is 0.1 time unit.

even in the worst case. Hence, in the following experiments, we will replace SMAX by BINSMAX which provides the same quality of result, while the execution is significantly more quickly.

For both BINSMAX and NEVERKILL heuristics, we generate a simulation of $E = 10^6$ tasks to estimate the performance. Figure 4.9 shows the performance of the two heuristics while varying the period τ, the ratio of deadline to period N, and the distribution of task execution times $\mathcal{D}$. We can also find in Figure 4.10 the corresponding best $s_{\max}$ values.

At first, it is obvious that the performance of both heuristics increase with the period, because the system load decreases. Our SMAX heuristic performs better than NEVERKILL in nearly all cases, and the difference between the two heuristics is larger (up to 30% in performance) when the period is small.

After that, while fixing the period, our BINSMAX heuristic has a relatively low performance when the ratio of deadline to period is small (i.e., $N = 2$). But when $N > 4$, the performance remains stable when N increases. On the other hand, the performance of NEVERKILL does not vary with N.

As for the distribution of task execution times, we can distinguish two types: The first one takes exponential distribution as example, for which BINSMAX and NEVERKILL perform very closely, and the best $s_{\max}$ chosen is the largest possible value in nearly all cases. The second one can be represented by uniform distribution, for which BINSMAX performs obviously better than NEVERKILL when the period $\tau \leq 1$, and the best $s_{\max}$ value increases with the period. It's important to note that, in some cases, performance can be similar for very different values of $s_{\max}$. This is why some non-smooth curve appears in Figure 4.10.

In conclusion, we find that BINSMAX performs better than NEVERKILL, especially in the case of overload. This difference of performance can be up to 30%. Thus, we will choose SMAX (more precisely, BINSMAX, the variant of SMAX) as task admission heuristic, and we will focus on task attribution heuristic in the next section.

Task attribution criteria with multiple processors

In this section, we will pass to the multiple processor platform. As already mentioned above, we consider BINSMAX as task admission heuristic, and we will try to compare our task attribution heuristics. In the figures, we add also NEVERKILL heuristic as reference.

Figures 4.11 to 4.15 show the performance of ROUNDROBIN and EARLIESTSTARTTIME under different parameter sets. We can find that, when the number of processors increases, a few points are missing in the figures. This is because the calculation is too expensive in these cases, and we do not have enough time to finish them. But we can find that, the points missing have relatively large values of N and τ, where the performance of all heuristics equals to 1. Therefore, the missing of these points will not lead to loss of information. On the other hand, for BINSMAX heuristic, we use the theoretical model of ROUNDROBIN to find the best $s_{\max}$ values while executing simulation under EARLIESTSTARTTIME criteria. This is because the theoretical model of EARLIESTSTARTTIME is too complicated that it is not possible to find a result of best $s_{\max}$ in a reasonable time.

We can find that, similar to the single processor case, the performance of all heuristics increases with the period. While increasing the value of N, the performance increases and then becomes stable.

Within task attribution criteria, EARLIESTSTARTTIME performs better than ROUNDROBIN, but the difference becomes smaller when the number of processors increases. EARLIESTSTARTTIME and ROUNDROBIN perform similarly in most of the cases, smaller than that between

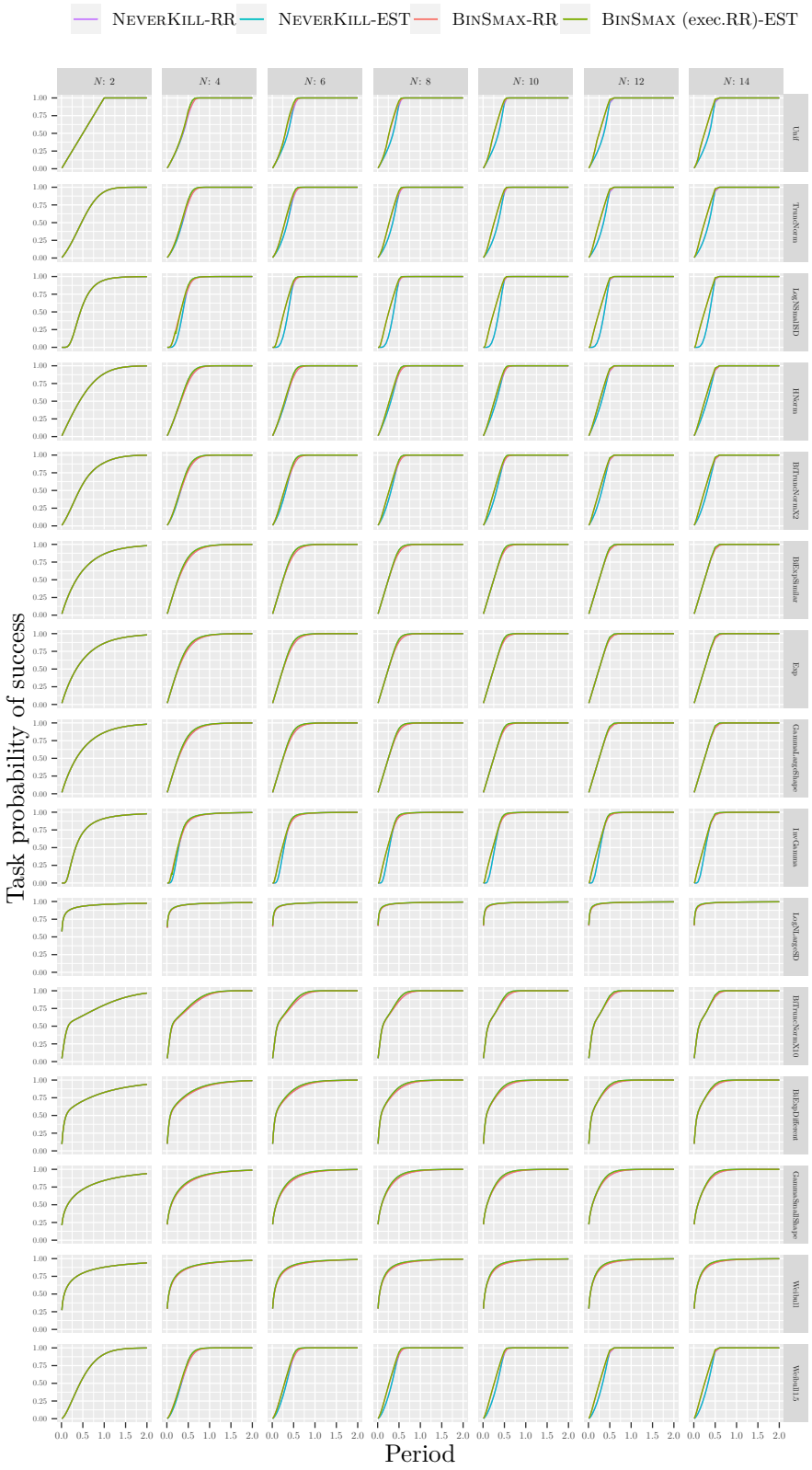


Figure 4.11: Absolute performance of task attribution heuristics RoundRobin and EarliestStartTime when the quantum length is 0.1 time unit, with **2** processors. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.

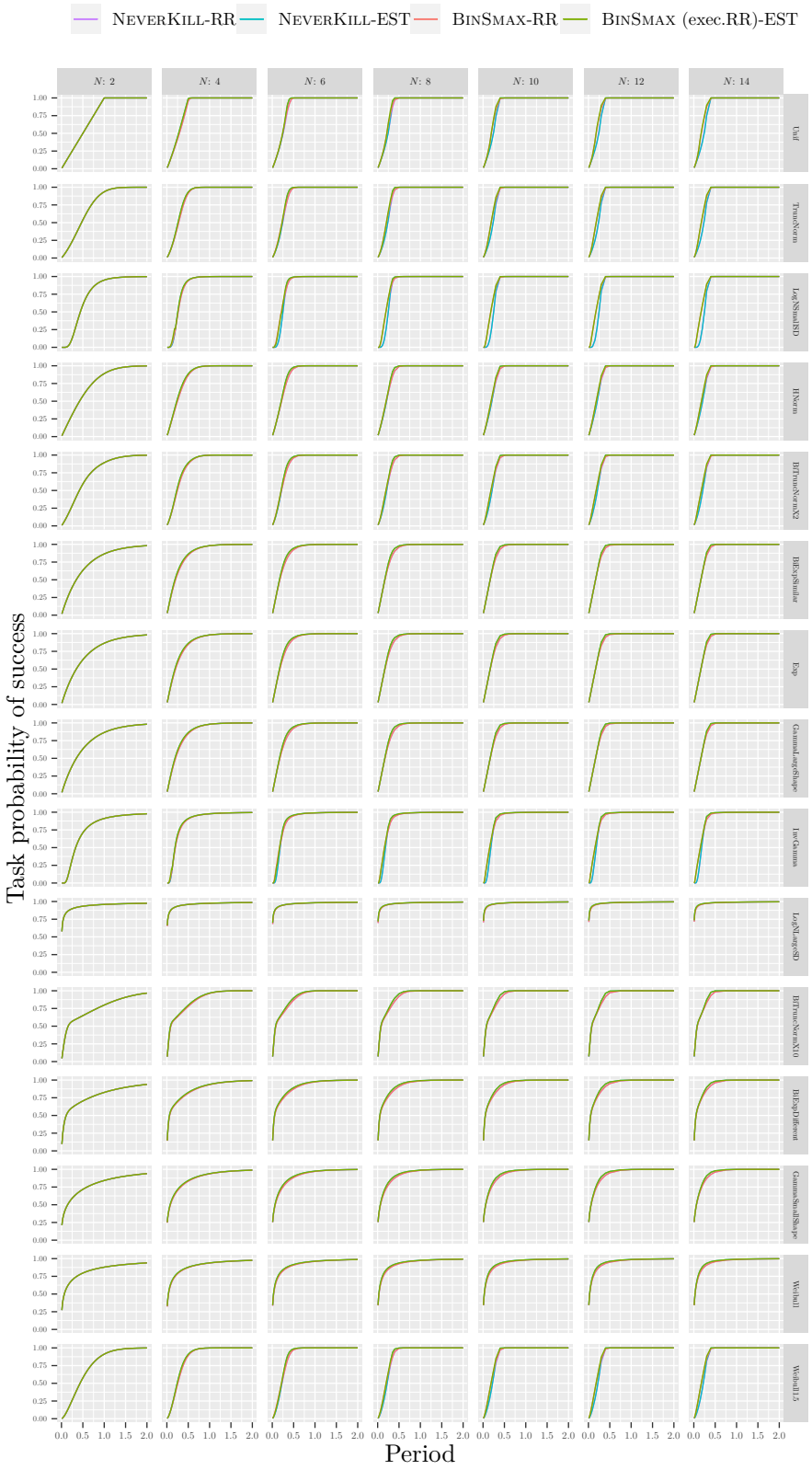


Figure 4.12: Absolute performance of task attribution heuristics RoundRobin and EarliestStartTime when the quantum length is 0.1 time unit, with **3** processors. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.

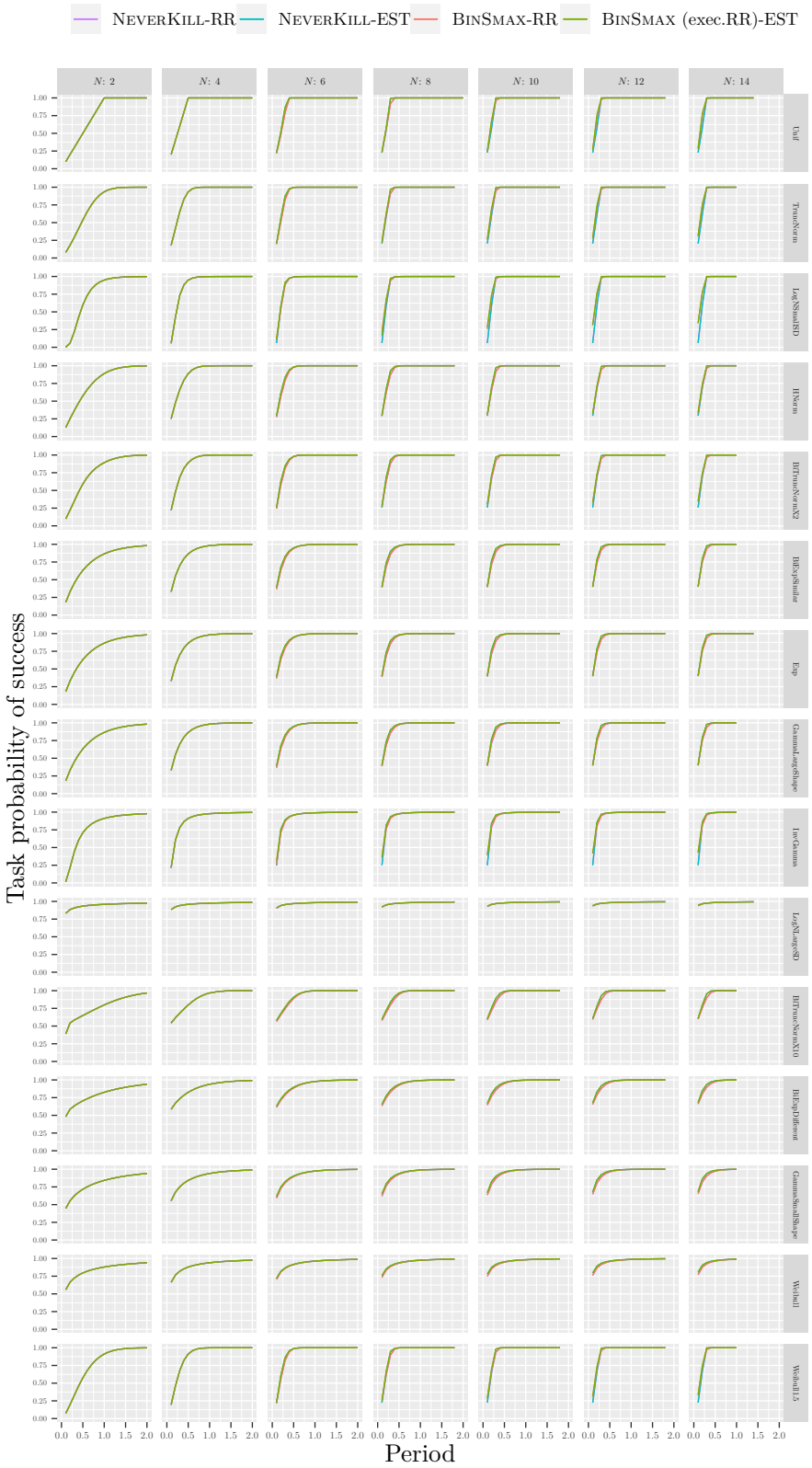


Figure 4.13: Absolute performance of task attribution heuristics RoundRobin and EarliestStartTime when the quantum length is 0.1 time unit, with **4** processors. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.



Figure 4.14: Absolute performance of task attribution heuristics RoundRobin and EarliestStartTime when the quantum length is 0.1 time unit, with **5** processors. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.



Figure 4.15: Absolute performance of task attribution heuristics RoundRobin and EarliestStartTime when the quantum length is 0.1 time unit, with **6** processors. The choice of the values of the parameter $s_{\max}$ is defined through the Markov chain approach, but the performance are assessed through simulations of 10^6 tasks.

BinSmax and NeverKill task admission heuristics. We can see difference when the performance is strictly smaller but very close to 1. For example, using exponential distribution, this difference can be found between $\tau = 0.5$ and 0.75 when the number of processors is equal to 2.

Therefore, we can conclude that, EarliestStartTime performs better than or similar to RoundRobin in all cases, and we decide to choose EarliestStartTime as task attribution criteria.

Summary

In this section, we did firstly a parametrization test which helps us to establish a framework which is accurate enough and costs less time: We use a theoretical model in which time is discretized into quantums of duration $q = 0.1$ time unit to calculate the best value of task admission criteria. We use a simulation of $E = 10^6$ to evaluate the performance while the values of task admission criteria are fixed. On the other hand, the simulation results confirm that Smax (or its variant BinSmax) and EarliestStartTime reaches better performance than the other task attribution and admission heuristics, up to 30% in performance.

4.6 Conclusion

In this work, we have studied the problem of allocating and scheduling firm real-time tasks on a platform composed of one or more identical processors. The difficulty is to decide: (i) on which processor to allocate each task, (ii) whether and when to interrupt a (long-lasting) task, (iii) which task in the waiting queue should be launched when a processor is idle. We designed a two-phase heuristic to solve this problem. At first, when a task is released, we map it to a processor either according to the Round robin strategy (RoundRobin) or the Earliest start time strategy (EarliestStartTime). Secondly, we can dynamically decide to interrupt and start tasks, based on several criteria: duration of time since release ($s_{\max}$), current length of execution time ($l_{\max}$) and remaining time until deadline ($d_{\max}$). We constructed a Markov chain based on a discretization of time and processor availability probabilities, in order to estimate the best values for these criteria, given the values of task period (τ), ratio of deadline to period (N), and distribution of task execution times ($\mathcal{D}$). We have conducted an extensive set of experiments, which showed that EarliestStartTime and Smax outperform other task attribution and task admission heuristics, and can achieve a gain of performance up to 30%. On the other hand, BinSmax is proved to have the same accuracy as Smax, but because of the binary search method, the execution speed is much faster than Smax.

Chapter 5

Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms

5.1 Introduction

Real-time systems are composed of periodic tasks that are regularly input to a parallel computing platform and must complete execution before their deadlines. In many applications, another requirement is reliability: the execution of each task is prone to transient faults, so that several replicas of the same task must be executed in order to guarantee a prescribed level of reliability [18, 104]. Recently, several strategies have been introduced with the objective to minimize the expected energy consumption of the system while matching all deadlines and reliability constraints [44, 45].

This work aims at extending these energy-aware strategies in the context of heterogeneous platforms. Heterogeneous platforms have been used for safety-critical real-time systems for many years [38]. With the advent of multiple hardware resources such as multi-cores, GPUs, and FPGAs, modern computing platforms exhibit a high level of heterogeneity, and the trend is increasing. The multiplicity of hardware resources with very different characteristics in terms of speed profile, reliability level and energy cost, raises an interesting but challenging problem: given several device types, which ones should we keep and which ones should we discard, in order to achieve the best possible tri-criteria trade-off (time, energy, reliability)? Needless to say, this optimization problem is NP-hard, even with two identical error-free processors, simply because of matching deadlines.

This work provides several mapping and scheduling heuristics to solve the tri-criteria problem on heterogeneous platforms. The design of these heuristics is much more technical than in the case of identical processors. Intuitively, this is because the reliability of a replica of one task depends upon the processor which executes it, and is different for each task instance (which we define in Section 5.2.1 below). More precisely, the reliability of a replica of the j-th instance of the i-th task mapped on processor m_k can be estimated by $R(\tau_{i,j}, m_k) = e^{-\lambda_k c_{i,k}}$, where $c_{i,k}$ is the worst case execution time (WCET, which is common for all instances of the same task) of task τ_i on m_k, and λ_k the failure rate of m_k. The total reliability of a task instance can be estimated by a function of the reliability of all its replicas (which we explicit in Equation 5.1 below); hence, it is not known until the end of the mapping process, unless we pre-compute an exponential number of reliability values. Then there are many processors to choose from,

and those providing a high reliability, thereby minimizing the number of replicas needed to match the reliability threshold, may also require a high energy cost per replica: in the end, it might be better to use less reliable but also less energy-intensive processors. Furthermore, the reliability is not enough to decide for the mapping: if two processors offer similar reliabilities for a task, it might be better to select the one with smaller execution time, in order to increase the possibility of mapping other tasks without exceeding any deadline. Altogether, we face a complicated decision, and we provide several criteria to guide the mapping process.

Overall, the objective is to minimize the expected energy consumption while matching all deadlines and reliability constraints. The expected energy consumption is the average energy consumed over all failure scenarios. Consider a sample execution: whenever the execution of a task replica succeeds, all the other replicas are instantaneously deleted; therefore, the actual amount of energy consumed depends both upon the error scenario (which replica is the first successful) and upon the overlap between replicas (some replicas are partially executed and interrupted when the successful one completes). Given a mapping, the scheduling phase aims at reducing overlap between any two replicas of the same task. Note that having an overlap-free scheduling is not always possible, because of utilization constraints. Also, deciding whether an overlap-free scheduling exists for a given mapping is NP-hard [43], even for deterministic tasks.

Finally, in actual real-time systems, tasks often complete before their worst-case execution time, or WCET, so that execution times are routinely modeled as stochastic. For instance, one typically assumes that the execution time of every instance of task τ_i on m_k follows a common uniform probability distribution in the range $[\beta_{\mathrm{b/w}} c_{i,k}, c_{i,k}]$ for some constant $\beta_{\mathrm{b/w}} < 1$ (ratio of best case over worst case).

In the end, the expected energy consumption must also be averaged over all possible values for execution times in addition to over all failure scenarios. To assess the performance of our heuristics, we use a comprehensive set of execution scenarios, with a wide range of processor speed profiles and failure rates. When the failure rate is low, most heuristics are equivalent, but when the failure rate is higher, only a few heuristics achieve good performance. Because we have no guarantee on the performance of the global mapping and scheduling process, we analytically derive a lower-bound for the expected energy consumption of any mapping. This bound cannot always be met. Nevertheless, we show that the performance of our best heuristics remains quite close to this bound in the vast majority of simulation scenarios.

The main contributions of this chapter are the following:

- The formulation of the tri-criteria optimization problem;
- The design of several mapping and scheduling heuristics;
- The characterization of a lower-bound for energy consumption;
- An experimental evaluation based on a comprehensive set of simulations scenarios, showing that our best heuristics achieve a significant gain over the whole spectrum of application and platform parameters.

The rest of the chapter is organized as follows. Section 5.2 provides a detailed description of the optimization problem under study, including a few notes on its complexity. The mapping and scheduling heuristics are described in Section 5.3 and 5.4 respectively. The lower-bound of energy consumption is introduced in Section 5.5. Section 5.6 is devoted to a comprehensive experimental comparison of the heuristics. Finally, Section 5.7 gives concluding remarks of this work.

Table I: Key Notations

Notation	Explanation
N and M	number of tasks and of processors
τ_i	the i-th task
$\tau_{i,j}$	the j-th instance of the i-th task
m_k	the k-th processor
p_i	period (deadline) for each instance of task τ_i
$L = \text{lcm}_{1 \leq i \leq n} p_i$	hyperperiod of the system
$w_{i,j,k}$	actual execution time for task instance $\tau_{i,j}$ on processor m_k
$c_{i,k}$	WCET for task τ_i on processor m_k
$u_{i,k} = \frac{c_{i,k}}{p_i}$	utilization of task τ_i executing on processor m_k
u_k	utilization of m_k (sum of utilization of replicas assigned to m_k)
$\mathcal{R}_i$	target reliability threshold for task τ_i
λ_k	failure rate of processor m_k
$R(\tau_{i,j}, m_k)$	reliability of task $\tau_{i,j}$ on processor m_k
$P_{k,d}$	dynamic power consumed per time unit on processor m_k
$P_{k,s}$	static power consumed per time unit on processor m_k
E_s	total static energy consumption of the system
E_d	total dynamic energy consumption of the system
$E_d(\tau_{i,j}, m_k)$	dynamic energy cost of task instance $\tau_{i,j}$ on processor m_k

5.2 Model

The inputs to the optimization problem are a set of real-time independent tasks, a set of non-identical processors and a reliability target. Key notations are summarized in Table I.

5.2.1 Platform and tasks

The platform consists of M heterogeneous processors $m_1, m_2, \ldots, m_M$ and a set of N periodic atomic tasks $\tau_1, \tau_2, \ldots, \tau_N$. Each task τ_i has WCET $c_{i,k}$ on processor m_k. The WCETs among different processors are not necessarily related. In the experiments, we generate the $c_{i,k}$ values with the method proposed in [11], where we have two parameters to control the correlation among task execution times and processors (see Section 5.6.1 for details). Each periodic task τ_i generates a sequence of *instances* with period p_i, which is equal to its deadline. The j-th instance of task τ_i is released at date $(j-1)p_i$ and its deadline is jp_i. The whole input pattern repeats every hyperperiod of length $L = \text{lcm}_{1 \leq i \leq n} p_i$. Each task τ_i has $\frac{L}{p_i}$ instances within the hyperperiod.

As already mentioned, real-time tasks usually complete execution earlier than their estimated WCET: actual execution times are assumed to be data-dependent and non-deterministic, randomly sampled from some probability distribution whose support is upper bounded by the WCET. See Section 5.6.1 for details on the generation of actual execution times from WCET values. The *utilization* $u_{i,k}$ of task τ_i executing on processor m_k is defined as $u_{i,k} = \frac{c_{i,k}}{p_i}$. The utilization of a processor is the sum of the utilizations of all task instances that are assigned to it.

The execution of any task can be preempted, that is, to be temporarily stopped (for instance to allow for the execution of another task) and later resumed without any induced penalty.

5.2.2 Reliability

We consider transient faults, modeled by an exponential probability distribution of rate λ_k on processor m_k. Thus, fault rates differ from one processor to another. This is a very natural assumption for a heterogeneous platform made of different-type processors. At the end of the execution of each task, there is an *acceptance test* to check the occurrence of soft errors induced by the transient faults. It is assumed that acceptance tests are 100% accurate, and that the duration of the test is included within the task WCET [45].

The *reliability* of a task instance is the probability of executing it successfully without software faults. It is related to its execution time. But in our problem, only the tasks WCETs are known, but not the actual execution times. Thus, in our heuristics, we use the WCETs to estimate the reliability of task instances and to make mapping decisions. The reliability of task instance $\tau_{i,j}$ on processor m_k with WCET $c_{i,k}$ can be estimated by $R(\tau_{i,j}, m_k) = e^{-\lambda_k \times c_{i,k}}$. Two instances $\tau_{i,j}$ and $\tau_{i,j'}$ of the same task τ_i have the same estimated reliability if they are executed on the same processor, because their WCETs are the same. But the actual reliability during execution is not the same ($e^{-\lambda_k \times w_{i,j,k}}$ and $e^{-\lambda_k \times w_{i,j',k}}$ respectively) because of their different actual execution times.

Note that a task instance cannot be replicated twice on the same processor. Thus, in the final schedule, task instance $\tau_{i,j}$ may have several replicas executing on different processors, in order to match its reliability threshold. Let $alloc(i,j)$ denote the index set of the processors executing a replica of $\tau_{i,j}$. The mapping achieves the following reliability $R(\tau_{i,j})$ for task instance $\tau_{i,j}$:

$$R(\tau_{i,j}) = 1 - \Pi_{k \in alloc(i,j)}(1 - R(\tau_{i,j}, m_k)) \tag{5.1}$$

Indeed, the task will succeed if at least one of its replicas does. The success probability is thus equal to 1 minus the probability of all replicas failing, which is the expression given in Equation (5.1). It also means that all other replicas (executing or not started) can be canceled when the first one is successfully finished.

Each instance $\tau_{i,j}$ of task τ_i has a reliability threshold $\mathcal{R}_i$ which is an input of the problem and that must be met by the mapping. In other words, the constraint writes $R(\tau_{i,j}) \geq \mathcal{R}_i$ for $1 \leq i \leq N$ and $1 \leq j \leq \frac{L}{p_i}$. This threshold is always satisfied during the actual execution, although we use the estimated reliability for mapping; this is because the actual execution time can never be greater than the WCET and, thus, the actual reliability is never smaller than the estimated one.

Because the tasks are independent, it is natural to assume that they might have different reliability thresholds: a higher threshold means that more resources should be assigned for the task to complete successfully with a higher probability. In the experiments we use $\mathcal{R}_i = \mathcal{R}$ for all tasks, but our heuristics are designed to accommodate different thresholds per task.

5.2.3 Power and energy

The power consumed per time unit on processor m_k is composed of two parts, static power ($P_{k,s}$) and dynamic power ($P_{k,d}$). The static power is consumed permanently if the processor m_k is used in the schedule. In contrast, the dynamic power is consumed only if the processor

is actually executing a task. To summarize, we have $2M$ input values, $\{P_{1,s}, P_{2,s} \dots P_{M,s}\}$ for static powers and $\{P_{1,d}, P_{2,d} \dots P_{M,d}\}$ for dynamic powers.

The total static energy consumption of the system during one hyperperiod is simply given by

$$E_s = \sum_{k \in Used} P_{k,s} \times L \tag{5.2}$$

where *Used* denotes the index set of the processors used by the schedule.

Similarly to Section 5.2.2, the dynamic energy consumption of one replica of task instance $\tau_{i,j}$ on processor m_k is estimated using the WCET $c_{i,k}$ of task τ_i:

$$E_d(\tau_{i,j}, m_k)^{estimated} = P_{k,d} \times c_{i,k} \tag{5.3}$$

in which we use the same value $c_{i,k}$ for all instances $\tau_{i,j}$ of the same task τ_i on m_k because their WCET is the same. Thus, the estimated dynamic energy consumption during one hyperperiod, which is used to compute the schedule, is expressed as

$$E_d^{estimated} = \sum_{(i,j,k)|k \in alloc(i,j)} E_d(\tau_{i,j}, m_k)^{estimated} \tag{5.4}$$

But the actual energy consumption of $\tau_{i,j}$ will likely be lower than stated in Equation (5.3). First the actual execution time $w_{i,j,k}$ is typically smaller than the WCET $c_{i,k}$. In addition, recall that we cancel all other replicas of a task instance as soon as one of its replicas has successfully completed. Thus, the execution time of a replica can be reduced or even zeroed out by the success of another replica of the same task instance.

We define a scenario $s = \{W, F\}$ as the random drawing of the execution times $w_{i,j,k}$ and of the failure booleans $f_{i,j,k}$ for all (i, j, k) where $k \in alloc(i, j)$. $f_{i,j,k} = True$ if the replica of task instance $\tau_{i,j}$ executed on processor m_k can be successfully completed, and $f_{i,j,k} = False$ if it will be victim of a failure. Again, some replicas will be interrupted or never launched, depending upon the scenario. Given a scenario sc, we let $r_{i,j,k}$ denote the actual execution times of each replica of each task instance. These values are dynamically computed as the execution of the schedule progresses. We compute the actual dynamic energy as

$$E_d(\tau_{i,j}, m_k)_{sc}^{actual} = P_{k,d} \times r_{i,j,k} \tag{5.5}$$

$$E_{d,sc}^{actual} = \sum_{(i,j,k)|k \in alloc(i,j)} E_d(\tau_{i,j}, m_k)_{sc}^{actual} \tag{5.6}$$

Finally, we weight each scenario $sc \in S$ by its probability p_{sc}, where S is the set of all scenarios, and compute the expected dynamic energy consumption during one hyperperiod:

$$E_d^{expected} = \sum_{sc \in S} (p_{sc} * E_{d,sc}^{actual}) \tag{5.7}$$

Altogether, the expected energy consumption of the schedule during one hyperperiod becomes:

$$E^{expected} = E_s + E_d^{expected} \tag{5.8}$$

The optimization objective is to find the schedule which minimizes this value. Clearly, there is no closed-form formula for the expected energy consumption, whose value is approximated by Monte-Carlo simulations.

5.2.4 Optimization Objective

The objective is to determine a set of replicas for each task, a set of processors to execute them, and to build a schedule of length at most L, so that the expected energy consumption is minimized, while matching the reliability threshold $\mathcal{R}_i$ for each task τ_i and the deadline $j \times p_i$ of each task instance $\tau_{i,j}$. As already mentioned in Section 5.1, the expected energy consumption is an average made over all possible execution times randomly drawn from their distributions, and over all failure scenarios (with every component weighted by its probability to occur). An analytical formula is out of reach, and we use Monte-Carlo sampling in the experiments. However, we stress the following two points:

- To guide the design of the heuristics, we use a simplified objective function; more precisely, we use WCETs instead of (yet unknown) actual execution times, and we conservatively estimate the dynamic energy of a task as the sum of the dynamic energy of all its replicas. Because mapping decisions are based upon WCETs, the number of enrolled processors does not depend upon actual execution times and the static energy is always the same for all scenarios, namely the length of the period times the sum of the static powers of the enrolled processors (see Equation (5.2)).
- To assess the absolute performance of the heuristics, we derive a lower-bound for the dynamic energy. This bound is based upon actual execution times but neglects scheduling constraints and assumes no overlap between any two task replicas; hence, it is not reachable in general. However, we show that our best heuristics achieve performance close to this bound.

5.2.5 Complexity

The global optimization problem is obviously NP-hard, since it is a generalization of the makespan minimization problem with a fixed number of parallel processors [35]. The optimization of the sole scheduling phase is also NP-hard for identical processors: if the number of replicas has already been decided for each task, and if the assigned processor of each replica has also been decided, the scheduling phase aims at minimizing the expected energy consumption by avoiding overlap between the replicas of a same task [43]. In fact, the heterogeneity of the processors, both in terms of power cost and fault rate, dramatically complicates the problem. Proposition 4 shows that the problem remains NP-complete when mapping identical and deterministic tasks with same period onto processors with different fault-rates, even without any energy constraint; in other words, choosing the optimal subset of processors to execute each task and its replicas to match their reliability threshold is an intrinsically combinatorial problem! We formally state the corresponding decision problem:

Definition 3 (Reliability)**.** *Consider n identical and deterministic tasks τ_i with same period and deadline $p = 1$, and same reliability threshold $\mathcal{R}$, to be executed on a platform composed of M heterogeneous processors. On processor m_k, $1 \leq k \leq M$, each task τ_i has execution time $c_{i,k} = 1$ and reliability $R(\tau_i, m_k) = R_k$. Is it possible to map all tasks so as to match the reliability threshold for each task?*

Because the execution time on each processor is always equal to the period/deadline, the hyperperiod of length $L = 1$ is composed of a single instance of each task, and all replicas will necessarily be executed in parallel in the time interval $[0, 1]$; hence each processor executes at most one replica.

Proposition 4. RELIABILITY *is NP-Complete.*

Proof. The problem is obviously in NP. For the completeness, we use a reduction from 3-PARTITION [35]. Consider an arbitrary instance $\mathcal{I}_1$ of 3-PARTITION: given $3m$ positive integers a_i, $1 \leq i \leq 3m$ with $\frac{B}{4} < a_i < \frac{B}{2}$ and $\sum_{i=1}^{3m} a_i = mB$, can we find a partition of $\{1, 2, \ldots, 3m\}$ into m subsets I_j such that $\sum_{i \in I_j} a_i = B$ for $1 \leq j \leq m$? We build the following instance $\mathcal{I}_2$ of RELIABILITY: we have $M = 3m$ processors and $n = m$ tasks. A replica on processor m_k, $1 \leq k \leq M$, has reliability $R_k = \frac{a_k}{A}$ where $A = 6B^2 + 2B^3$. Finally, the target reliability threshold for each task is $\mathcal{R} = \frac{B}{A}(1 - 3\frac{B}{A})$. The size of $\mathcal{I}_2$ is clearly polynomial (and even linear) in the size of $\mathcal{I}_1$.

We now show that $\mathcal{I}_2$ has a solution if and only if $\mathcal{I}_1$ does. Assume first that $\mathcal{I}_1$ has a solution and let I_j denote the m subsets of the partition. Note that each subset has exactly three elements because on the condition on the size of the a_i's. For $\mathcal{I}_2$, we map task τ_i, $1 \leq i \leq n = m$, on the three processors whose index belongs to I_i. This is a valid mapping onto the $M = 3m$ processors because the m subsets form a partition of $\{1, 2, \ldots, M\}$. Each task has three replicas. Writing $I_i = \{i_1, i_2, i_3\}$, the reliability of τ_i writes

$$\begin{aligned} \mathcal{R}_i &= 1 - \textstyle\prod_{q=1}^{3}(1 - R_{i_q}) \\ &= \frac{\sum_{q=1}^{3} a_{i_q}}{A} - \frac{\sum_{1 \leq q < r \leq 3} a_{i_q} a_{i_r}}{A^2} + \frac{a_{i_1} a_{i_2} a_{i_3}}{A^3} \\ &\geq \frac{\sum_{q=1}^{3} a_{i_q}}{A} - \frac{\sum_{1 \leq q < r \leq 3} a_{i_q} a_{i_r}}{A^2} \end{aligned} \tag{5.9}$$

Since $\sum_{q=1}^{3} a_{i_q} = B$ and $a_i < \frac{B}{2}$ for all $1 \leq i \leq 3m$, we obtain

$$\mathcal{R}_i \geq \frac{B}{A} - \frac{3B^2}{4A^2} \geq \frac{B}{A} - \frac{3B^2}{A^2} = \mathcal{R}$$

Hence, $\mathcal{I}_2$ has a solution.

Assume now that $\mathcal{I}_2$ has a solution, and let I_i denote the indices of the processors executing replicas of task τ_i for $1 \leq i \leq n$. The subsets I_i are pairwise disjoint, but we do not know whether they form a partition yet (some processors may have not been enrolled in the mapping). But assume that some I_i has only 1 element i_1: we get $\mathcal{R}_i = \frac{a_{i_1}}{A} \geq \mathcal{R}$. This leads to

$$a_{i_1} \geq B\left(1 - 3\frac{B}{A}\right) = B - \frac{3B^2}{A} \geq B - \frac{1}{2}$$

The last inequality comes from the definition of A which has been chosen large enough. But $a_{i_1} < \frac{B}{2} < B$ implies $a_{i_1} \leq B - 1$ since we have integers. Hence, a contradiction. Similarly, if some I_i has only 2 elements i_1 and i_2: we get

$$\mathcal{R}_i = \frac{a_{i_1} + a_{i_2}}{A} - \frac{a_{i_1} a_{i_2}}{A^2} \geq \mathcal{R}$$

hence $\frac{a_{i_1} + a_{i_2}}{A} \geq \mathcal{R}$. We conclude as before, because it implies that $a_{i_1} + a_{i_2} \geq B - \frac{1}{2}$, while $a_{i_1} + a_{i_2} < 2\frac{B}{2} = B$ implies $a_{i_1} + a_{i_2} \leq B - 1$. As a consequence, each subset I_i has at least 3 elements. We do have a partition of $\{1, 2, \ldots, 3m\}$, and each subset I_i has exactly 3 elements. Then, writing $I_i = \{i_1, i_2, i_3\}$ as before, we have $\mathcal{R}_i \geq \mathcal{R} = \frac{B}{A}(1 - 3\frac{B}{A})$ for all $1 \leq i \leq m$, where $\mathcal{R}_i$ is given by Equation (5.9). We derive that

$$\begin{aligned} \frac{\sum_{q=1}^{3} a_{i_q}}{A} &\geq \frac{B}{A}\left(1 - 3\frac{B}{A}\right) + \frac{\sum_{1 \leq q < r \leq 3} a_{i_q} a_{i_r}}{A^2} - \frac{a_{i_1} a_{i_2} a_{i_3}}{A^3} \\ &\geq \frac{B}{A}\left(1 - 3\frac{B}{A}\right) - \frac{a_{i_1} a_{i_2} a_{i_3}}{A^3} \end{aligned}$$

As $a_i < \frac{B}{2}$ for $1 \leq i \leq 3m$, we can deduce:

$$\frac{\sum_{q=1}^{3} a_{i_q}}{A} \geq \frac{B}{A}\left(1 - 3\frac{B}{A}\right) - \frac{B^3}{8A^3}$$

Recall that $A = 6B^2 + 2B^3$ and B is positive integer. Hence,

$$\sum_{q=1}^{3} a_{i_q} \geq B\left(1 - 3\frac{B}{A}\right) - \frac{B^3}{8A^2} \geq B\left(1 - 3\frac{B}{A}\right) - \frac{B^3}{A}$$
$$\geq B - \frac{3B^2 + B^3}{A} \geq B - \frac{1}{2}$$

Then $\sum_{q=1}^{3} a_{i_q} \geq B$ because we have integers. But the sum of the $3m$ integers a_i is mB. Hence, all the sums are indeed equal to B, and we have a solution to $\mathcal{I}_1$. This concludes the proof. □

5.3 Mapping

In the mapping phase, we need to define the number of replicas for each task instance, as well as the execution processor for each replica, aiming at meeting the reliability target and deadline, while minimizing the energy cost. As hyperperiods are repetitive, we aim at finding a feasible mapping within one hyperperiod (time length L). One difficulty introduced by platform heterogeneity is that we do not know the number of replicas needed for each task instance to reach its reliability threshold, before completing the mapping process, because different processors have different failure rates and speeds and, hence, they provide different reliabilities for each replica. Therefore, the simpler three-step method of [43, 45] cannot be applied.

Since the estimation of dynamic energy consumption ($E_d(\tau_{i,j}, m_k)^{estimated}$) and reliability ($R(\tau_{i,j}, m_k)$) of task instance $\tau_{i,j}$ on processor m_k do not depend on the index of instance j, these two values will be the same for all instances of the same task. Thus, in the following description, instead of making decision for each task instance $\tau_{i,j}$, we will use $E_d(\tau_i, m_k)$ and $R(\tau_i, m_k)$ as the estimated dynamic energy consumption and reliability on processor m_k, for all instances of task τ_i, and we will make the same mapping decision for all instances of the same task.

When mapping all the $\frac{L}{p_i}$ instances of a given task on a processor, we use the standard *Earliest Deadline First (EDF)* scheduling heuristic [61]. EDF tells us that a given processor is a fit for that replica if and only if the utilization of that processor does not exceed 1. Recall that the utilization of a processor is the sum of the utilizations of all task instances assigned to it.

As shown in Algorithm 3, given a set of tasks with their periods and reliability targets and a set of heterogeneous processors, we first order the tasks according to TaskMapCriterion, which can be either:

- *deW* (*inW*): decreasing (increasing) average work size $\bar{c_i} = \frac{c_{i,1}+c_{i,2}+\cdots+c_{i,M}}{M}$;
- *deMinW* (*inMinW*): decreasing (increasing) minimum work size $\bar{c_i} = \min_{1\leq k\leq M} c_{i,k}$;
- *deMaxW* (*inMaxW*): decreasing (increasing) maximum work size $\bar{c_i} = \max_{1\leq k\leq M} c_{i,k}$;
- *random*: random ordering.

Then, for each task in the ordered list, we order the processors for mapping its replicas according to ProcMapCriterion, which can be either:

- *inE*: increasing energy cost;

Table II: Example of processors with different characteristics and two ratios calculated

m_k	$E(\tau_i, m_k)$	$R(\tau_i, m_k)$	$\frac{R(\tau_i,m_k)}{E(\tau_i,m_k)}$	$-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$
1	1	0.9	0.9	1
2	1	0.9	0.9	1
3	2	0.99	0.495	1
4	1	0.99	0.99	2
5	2	0.9	0.45	0.5

- *deR*: decreasing reliability;
- *deP*: decreasing ratio of $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$ (explained below);
- *random*: random ordering.

We use the example shown in Table II to explain how to design a better criterion in ProcMapCriterion. Assume there are five processors with different energy and reliability configurations (the first two processors are identical). Considering only the reliability, we cannot distinguish between the third and fourth processors. Apparently, the fourth processor is better since it consumes less energy and provide the same level of reliability. The problem is the same when ordering processors only according to energy cost. This gives us a hint that we need to consider energy and reliability interactively. A first idea would be to use the ratio $\frac{R(\tau_i,m_k)}{E(\tau_i,m_k)}$, which expresses the reliability per energy unit of every instance of task τ_i executing on processor m_k. But consider a task instance with a reliability target $\mathcal{R}_i = 0.98$: it requires either the third processor alone or the first two processors together. Both solutions match the reliability goal with the same energy cost 2, but they have a different ratio. We aim at a formula that would give the same weight to both solutions. The ratio $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$ is a good candidate, because the total energy cost is the sum of all processors while the reliability is a product. This discussion explains how we have derived the third criterion *deP* in ProcMapCriterion, namely to order processors by decreasing ratio of $-\frac{\log_{10}(1-R(\tau_i,m_k))}{E(\tau_i,m_k)}$.

For the mapping phase, for task τ_i, we add replicas for all its instances $\tau_{i,j}$ in the order of the processor list until the reliability target $\mathcal{R}_i$ of each instance is reached. The algorithm uses the probability of failure $PoF = 1 - R(\tau_i) = \Pi_{k \in alloc(i)}(1 - R(\tau_i, m_k))$ (Equation (5.1)). The mapping process always ensures that: (i) no two replicas of the same task are assigned to the same processor; (ii) the utilization u_k of each processor does not exceed 1.

5.4 Scheduling

In the scheduling phase, we aim at ordering the tasks mapped on each processor, with the objective to further minimize the energy consumption during execution. Recall that the success of any replica leads to the immediate cancellation of all the remaining replicas, a crucial source of energy saving. Thus, our approaches identify a primary replica for each task instance, then all other replicas for that instance become secondaries. The goal of the proposed scheduling heuristics is to avoid overlaps between the execution of the primary and secondary replicas of each task instance: the primary must be terminated as soon as possible, while the secondaries must be delayed as much as possible.

Algorithm 3: Replication setting and mapping

Input: A set of tasks τ_i with reliability targets $\mathcal{R}_i$ and periods p_i; a set of heterogeneous processors m_k
Output: An allocation σ_m of all replicas on the processors

```
1  begin
2      order all the tasks with TaskMapCriterion and renumber them τ_1, ..., τ_N
       /* initialize the utilization of all processors to zero                    */
3      u ← [0, ..., 0]
       /* iterate through the ordered list of tasks                               */
4      for i ∈ [1, ..., N] do
           /* order processors for each task                                      */
5          order all processors for task τ_i with ProcMapCriterion and renumber them proc_1, ..., proc_M
           /* this ordered list may differ from task to task                      */
6          k = 1
7          PoF = 1
8          while 1 − PoF < R_i do
9              if k > m then
10                 return not feasible
               /* u_temp is the utilization of processor m_k after adding a replica of task τ_i
                  */
11             u_temp = u_k + u_{i,k}
12             if u_temp ≤ 1 then
13                 u_k = u_temp
14                 PoF = PoF × (1 − R(τ_i, m_k))
15                 for all instances of τ_i, add a replica on proc_k
16             k++
17         return σ_m
```

After the mapping phase, we have a static schedule on each processor, also called the *canonical* schedule, which is based upon the WCET of each task and EDF. The EDF schedule can use preemption, so that a given task instance may be split into several chunks. As a result, from the canonical schedule, each processor has an ordered list made up with all task instance chunks that it has to execute during the hyperperiod, together with their starting and finish times. As the actual execution time of a real-time task instance will be shorter than its WCET, the canonical schedule will never be executed exactly as such. Still, it can be used as the baseline to compute the maximum delay for secondary replicas without missing any deadline.

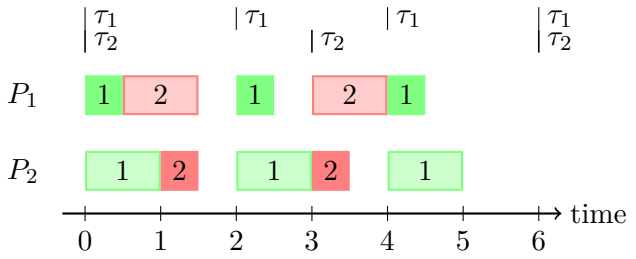
To determine the primary replica for each task instance, we could make the decision offline or online. The offline strategy means that we use pre-knowledge before real execution, where we consider the following criteria:

- EDF_WCET: choose the processor that can execute the replica the fastest;
- EDF_Energy: choose the processor that can execute the replica with the minimum dynamic energy consumption;
- EDF_Reliability: choose the processor that can execute the replica the most reliably.

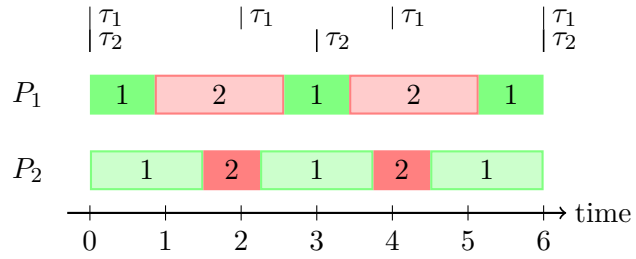
Note that for offline strategies, the primary replicas for different task instances of the same task are on the same processor. After determining the primary replicas, we deploy the following techniques based on the canonical schedule to differentiate the primary replica and the secondary replicas. The techniques are accompanied by figures, in which different colors represent different tasks, and primary replicas are marked with a darker color:

- Scale the WCETs of all tasks by $\frac{1}{\alpha}$, where α is the utilization, which also gives a valid canonical schedule, but with longer worst case expected execution times for all tasks, which we call the scaled canonical schedule as shown in Figure 5.1. This gives a better reference to further delay the start time of secondary replicas. Here is why: at the mapping phase, as long as the total utilization of replicas that are mapped onto the processor is less than or equal to one, then we are able to find a valid scheduling using EDF. Assume we have mapped three tasks t_i, t_j, t_k onto processor p, and that the utilization is $\alpha = \frac{c_{i,p}}{p_i} + \frac{c_{j,p}}{p_j} + \frac{c_{k,p}}{p_k}$. Either we keep the mapping and have a fraction $1 - \alpha$ of the interval where p is idle, or we artificially slow down the execution times of all three tasks by a factor α, then we will have a new utilization $\beta = \frac{c_{i,p}}{p_i\alpha} + \frac{c_{j,p}}{p_j\alpha} + \frac{c_{k,p}}{p_k\alpha} = 1$, which also gives us a feasible mapping without any idle time. Then we could delay the start time of secondary replicas without conflict timeliness as long as they will finish executions not later than their finish times in the scaled canonical schedule (Figure 5.2).
- Consider a *scheduling interval* defined by two consecutive deadlines in the canonical schedule. Inside the interval, task chunks to be executed are ordered by the EDF policy. We observe that we can freely reorder the chunks without missing any deadline, by definition of an interval. It means that in each interval, we could reorder to execute all primary replicas first, and then secondary replicas (Figure 5.3).
- Since we have delayed the start time of secondary replicas, there are some idle slots in the schedule. We could take advantage of these idle slots by pre-fetching other primary replica chunks in the availability list: those primaries that have been released but which were scheduled later because they have lower EDF priority than the current secondary replicas.

Based on the above ideas, considering one interval in the improved static schedule, it: 1) starts with the primary replica chunks; 2) fills in the idle slot by inserting other primary replica chunks that are available; 3) ends by the secondary replica chunks. Then, during the real execution, according to the failure case, the scheduler will dynamically cancel the execution of



(a) Canonical schedule



(b) Scaled canonical schedule

Figure 5.1: Example of scaling WCETs with two tasks τ_1 and τ_2 ($c_{1,1} = c_{2,2} = 0.5$, $c_{1,2} = c_{2,1} = 1$, $p_1 = 2$ and $p_2 = 3$) on each processor with total utilization $u_1 = \frac{7}{12}$ and $u_2 = \frac{4}{6}$.

some secondaries if the corresponding primary replica finished successfully. If the replica selected as primary is the least expensive energy-wise (like the one selected by EDF_Energy), and if no secondary replica has started executing yet, then the energy consumption will be minimal for that task. However, because the choice of the primary replica is highly dependent on the characteristics of processors, different tasks may have their primary replicas on the same processor, which would lead to load imbalance and heavy overlaps. To avoid this situation, we design another type of method to choose primary replicas on-the-fly:

- EDF_Start_Time: choose the processor that can start the execution of the replica the earliest (given already made scheduling decisions).

This online strategy dynamically chooses the replica starting the earliest as the primary replica for each task instance. This tends to spread primaries onto different processors, which gives more slack time on each processor to reduce overlaps. As a side note, it will possibly be the case that two different instances of the same task have not the same primary processor. Then all other replicas of the same task become secondaries and are delayed according to the scaled canonical schedule, as shown in Figure 5.2. We could not implement the "reordering inside intervals" shown in Figure 5.3 for EDF_Start_Time, which means that the end time of secondaries will not exceed the one planned in the scaled canonical schedule, as this is an offline improvement that requires the primary replica known in advance. But the online strategy has two major advantages compared to the offline strategies: (1) as long as we finish one replica successfully, we can safely cancel other replicas of the same task instance earlier than in offline schedules, which gives more flexibility to adjust the schedule afterwards; (2) the static schedules have to reserve time slots for the secondaries that correspond to their WCET (it is impossible to know their actual execution times before execution). Since their actual execution times are usually shorter, this dynamically frees some time slots that the online schedule uses to prefetch available primary replica chunks.

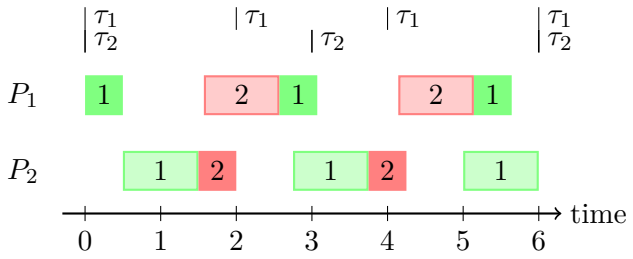


Figure 5.2: Static schedule when prioritizing primaries while delaying secondaries according to their finish times in the Scaled canonical schedule.

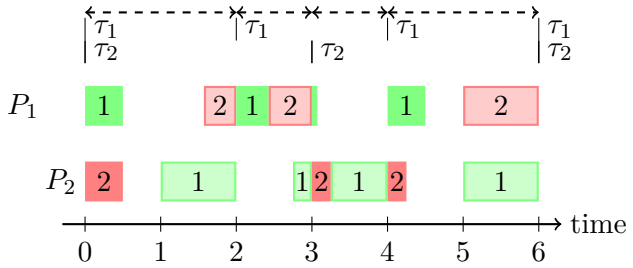


Figure 5.3: Reordering chunks freely inside intervals.

5.5 Lower bound

In this section, we explain how to derive a lower-bound for the expected energy consumption of a solution to the optimization problem, namely a mapping/scheduling heuristic that uses some of the selection criteria outlined in Sections 5.3 and 5.4.

For each problem input, namely N tasks τ_i with reliability thresholds $\mathcal{R}_i$, M processors m_k with failure rates λ_k, and with all WCET $c_{i,k}$, we compute a solution, i.e., a mapping and scheduling of all replicas. We first use Monte-Carlo simulations (see Section 5.6) and generate several sets of values for the actual execution time $w_{i,j,k}$ of task instance $\tau_{i,j}$ on processor m_k. The values $w_{i,j,k}$ are drawn *uniformly across processors* as some fraction of their WCET $c_{i,k}$ (refer to Section 5.6.1 for details).

Now, for each set of values $w_{i,j,k}$, we generate a set of failure scenarios, compute the actual energy consumed for each scenario, and report the average of all these values as the expected energy consumption. A failure scenario operates as follows. We call an event the end of the execution of a replica on some processor. At each event, we flip a biased coin (weighted with the probability of success of the replica on that processor) to decide whether the replica is successful or not. If it is, we delete all other replicas of the same task instance. At the end of the execution, we record all the dynamic energy that has been actually spent, accounting for all complete and partial executions of replicas, and we add the static energy given by Equation (5.2). This leads to the energy consumption of the failure scenario. We average the values over all failure scenarios and obtain the expectation, denoted as $E(\{w_{i,j,k}\})$.

In addition, we also compute a lower-bound $LB(\{w_{i,j,k}\})$ as follows. Our goal is to accurately estimate the energy consumption of an optimal solution. Since the static energy depends upon the subset of processors that are used in the solution (see Equation (5.2)), we need to try all possible subsets. Given a processor subset $\mathcal{S}$, we consider each task instance $\tau_{i,j}$ independently, and try all possible mappings of replicas of $\tau_{i,j}$ using only processors in $\mathcal{S}$. Thus we explore all

subsets $\mathcal{T}$ of $\mathcal{S}$. A subset $\mathcal{T}$ is *safe* if mapping a replica of $\tau_{i,j}$ on each processor of $\mathcal{T}$ meets the reliability criterion $\mathcal{R}_i$, and if no strict subset of $\mathcal{T}$ is safe. Note that safe sets are determined using the WCETs $c_{i,k}$, and not using the $w_{i,j,k}$, because of the problem specification. Now for each safe subset $\mathcal{T}$, we try all possible orderings (there are $k!$ of them if $|\mathcal{T}| = k$); for each ordering, we compute the expected value of the dynamic energy consumption as follows: if, say, $\mathcal{T} = \{m_1, m_3, m_4\}$ and the ordering is m_3, m_4, m_1, then we compute

$$P_{3,d}w_{i,j,3} + (1 - e^{-\lambda_3 w_{i,j,3}})P_{4,d}w_{i,j,4} + (1 - e^{-\lambda_3 w_{i,j,3}})(1 - e^{-\lambda_4 w_{i,j,4}})P_{1,d}w_{i,j,1}.$$

We see that we optimistically assume no overlap between the three replicas, and compute the dynamic energy cost as the energy of the first replica (always spent) plus the energy of the second replica (paid only if the first replica has failed) plus the energy of the third replica (paid only if both the first and second replicas have failed), and so on. Note that here we use execution times and failure probabilities based upon the actual execution times $w_{i,j,k}$ and not upon the WCETs $c_{i,k}$. The value of the sum depends upon the ordering of the processors in $\mathcal{T}$. Hence, we check the 6 orderings and retain the minimal value. We do this for all safe subsets and retain the minimal value. Finally we sum the results obtained for each task instance and get the lower-bound for the original processor subset $\mathcal{S}$. We stress that this bound is not necessarily tight, because our computation assumes no overlap for any replica pair, and does not check the utilization of each processor (which may exceed 1). The final lower-bound $LB(\{w_{i,j,k}\})$ is the minimum over all processor subsets. Although the computation has exponential cost, due to the exploration of all processor subsets $\mathcal{S}$, the computation of the expected energy for a given ordering in a subset $\mathcal{T}$ of $\mathcal{S}$ obeys a closed-form formula.

5.6 Performance evaluation

This section assesses the performance of our different strategies to map and schedule real-time tasks onto heterogeneous platforms. In Section 5.6.1, we describe the parameters and settings used during the experimental campaign. We present the results in Section 5.6.2. The algorithms are implemented in C++ and in R. The related computing code and experimental data are publicly available in [31].

5.6.1 Experimental methodology

In the experiments, we have $M = 10$ processors and $N = 20$ tasks. The hyperperiod L of the system is fixed at 300. The task instances $\tau_{i,j}$ are arrived every p_i, chosen between one of the following values: 20, 30, 50, 60, 100, 150. The set of WCETs is generated by the method proposed in [11], as mentioned in Section 5.2.1. The WCET values are controlled by the correlation factor between the different tasks (cor_{task}) and between the different processors (cor_{proc}). For example, $cor_{\text{task}} = 0$ (resp. $cor_{\text{proc}} = 0$) means that the WCET values between different tasks on one processor (resp. between different processors for one task) are completely randomly generated. Inversely, $cor_{\text{task}} = 1$ (resp. $cor_{\text{proc}} = 1$) means that the WCET values between different tasks on one processor (resp. between different processors for one task) are all the same. We vary these two parameters between 0.25 and 0.75 to visualize the result under different correlation conditions, while guaranteeing a certain degree of randomness. We also define a parameter *basicWork* as the estimated total utilization of the system with a single

replica per task instance, in order to study the impact of system workload pressure:

$$basicWork = \frac{\sum_{i,k} u_{i,k}}{M^2} \tag{5.10}$$

In Equation (5.10), we use the average utilization on the M processors ($\frac{\sum_k u_{i,k}}{M}$) to estimate the pressure that one replica of task τ_i can give on the system. We sum up the average utilization of all N tasks, and we divide this value by M because M processors are available. Hence, *basicWork* represents an estimation of the fraction of time that processors are used if each task has a single replica. In the experiments, we vary *basicWork* from 0.1 to 0.3.

To generate the actual execution times for each task instance from the task WCETs, we use two parameters. The first one, $\beta_{\mathrm{b/w}}$, is global to all tasks: $\beta_{\mathrm{b/w}}$ is the ratio between the best-case execution time and the worst-case execution time. It is the smallest possible ratio between the actual execution time of a task instance and the WCET of that task. Therefore, the actual execution time of all instances of task τ_i on processor m_k belongs to $[\beta_{\mathrm{b/w}} c_{i,k}, c_{i,k}]$. We consider five possible values of $\beta_{\mathrm{b/w}}$: 0.2, 0.4, 0.6, 0.8, and 1. The second parameter, $\beta_{i,j}$, is task instance dependent: $\beta_{i,j}$ describes whether the task instance $\tau_{i,j}$ is a small one or a large one. $\beta_{i,j}$ is randomly drawn in $[0, 1]$. $\beta_{i,j} = 0$ means that task instance $\tau_{i,j}$ has the shortest possible execution time, and $\beta_{i,j} = 1$ means that the actual execution is equal to its worst case execution time. Overall, the actual execution time of task instance $\tau_{i,j}$ on processor m_k is thus defined as: $w_{i,j,k} = (\beta_{\mathrm{b/w}} + (1 - \beta_{\mathrm{b/w}})\beta_{i,j})c_{i,k}$.

For a processor m_k in the platform, we fix its static power $P_{k,s}$ at 0.001 as in the literature [95, 97, 100]. For the dynamic power and the failure rate, we have two sets of parameters. The first parameter set also follows values from previous work [95, 97, 100]. For this set, we have a relatively high dynamic power and very low failure rate. Therefore, the replicas using this first set of parameters succeed in almost all cases. Then, to evaluate our heuristics in the context when failures occur more frequently, we introduce a second set of parameters where the replicas have lower dynamic power and relatively high failure rate. For the first set, we choose randomly the dynamic power $P_{k,d}$ between 0.8 and 1.2, and the failure rate λ_k between 0.0001 and 0.00023. And for the second set, we have $P_{k,d}$ 10 times smaller (between 0.08 and 0.12), and λ_k 100 times larger (between 0.01 and 0.023). With the second set of parameters, the actual reliability of one replica ranges from 0.1 to 0.99. To be more realistic, in our experiments processors with a larger dynamic power $P_{k,d}$ have a smaller failure rate λ_k. It means that a more reliable processor costs more energy than a less reliable one. We guarantee this by ordering inversely the $P_{k,d}$'s and the λ_k's after generating the random values.

We vary the local reliability target $\mathcal{R}_i$ between 0.8 and 0.95 for the big failure rate set and between 0.9 and 0.98 for the small failure rate set. This is to give the system a reasonable freedom while mapping and scheduling. The reliability target is relatively high, implying that tasks need several replicas to reach it. But it is chosen low enough so that feasible mappings can be found in the vast majority of scenarios.

5.6.2 Results

In this section, we compare the performance of the different criteria presented in Sections 5.3 and 5.4, and we choose the criterion which performs the best. Next, we analyze the impact of the different parameters on the performance of the chosen criterion.

Because of the large amount of experimental data, we used more than 300 figures to show the complete experimental results. As it is impossible to include all of them in this thesis,

we choose only to present a representative subset of the results. The comprehensive set of all results is available in the research report [34].

We choose as default values $\beta_{\mathrm{b/w}} = 1$, $basicWork = 0.3$, $\mathcal{R}_i = 0.95$ for big failure rate case, and $\mathcal{R}_i = 0.98$ for small failure rate case. This set of parameters is chosen to observe results when the platform is under maximum pressure. We fix cor_{task} and cor_{proc} both at 0.5 as default values.

Each experiment is an average of 25 WCETs sets generated as follows. We first generate 5 sets of periods. For each of these sets of periods, 5 WCET matrices are generated. For each WCET matrix, we generate 10 sets of random $P_{k,d}$ and λ_k values. For each $P_{k,d}$ and λ_k generated, the final result is the average over 10 executions. Overall, we run $2,500$ randomly generated experiments for each set of $\beta_{\mathrm{b/w}}$, $basicWork$, $\mathcal{R}_i$, cor_{task} and cor_{proc} values. The total number of experiments ran is 2,700,000 for each heuristic.

Each result point is represented as the percentage of energy saved over the random baseline method (defined below). Hence, on all the figures and in all the tables, the higher, the better. The random baseline method is defined as follows: for each task, we add replicas randomly on available processors until reaching the task reliability target during the mapping phase; for scheduling, we randomly order replicas mapped on each processor and execute them in sequence and as soon as possible. We compare our different strategies to the lower-bound proposed in Section 5.5. During the scheduling phase, we add several other references called EDF_Plain, EDF_Ref_Paper, and Smallest. They all use the same mapping as our heuristics, but have different scheduling strategies: EDF_Plain is simply the plain EDF heuristic. EDF_Ref_Paper is the heuristic proposed in [45]. Smallest considers a single replica for each task instance, the replica which costs the least energy (hence, Smallest is not always a valid heuristic as it may not satisfy the reliability threshold). In fact, Smallest shows an ideal case for the scheduling phase: every instance of every task successfully executes one replica, the replica which costs the least energy, while all other replicas are not yet started and are canceled after the first replica is completed. In contrast, LowerBound calculates the lowest expected energy consumption without considering the load of processors. Thus, we can find in the experimental results that Smallest can have a better result than LowerBound in some cases.

For the $2,500$ random trials for each setting, in the figures, we report the average number of replicas needed in total for the 20 tasks (on the left side), the average number of failures that occur per time unit (in the middle), and the average percentage of random trials in which we find feasible mapping (on the right side). These numbers are reported in black above the horizontal axis on each figure.

Task ordering criteria for the mapping phase

In Tables III and IV, we summarize the performance of different task ordering criteria during the mapping phase for different cor_{task} and cor_{proc} parameter sets. We see that, for each (cor_{task}, cor_{proc}) group, the difference between the different task ordering criteria, including random, is not obvious. The difference is at most 0.7% between the best and the worst criteria for any given set of (cor_{task}, cor_{proc}) values. We conclude that these criteria do not critically influence energy consumption. Therefore, in the following, for ordering tasks, we will only consider $deMinW$, which performs globally slightly better than others, especially in the big failure rate case. We now focus on how to select processors during the mapping phase, and on the scheduling strategies.

	random	*deW*	*inW*	*deMinW*	*deMaxW*	*inMinW*	*inMaxW*
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.25$	54.8%	54.7%	55.1%	55.0%	54.7%	54.8%	55.1%
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.50$	47.6%	47.5%	48.0%	47.6%	47.5%	48.0%	48.0%
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.75$	33.5%	33.8%	33.7%	33.8%	33.8%	33.7%	33.7%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.25$	55.1%	55.4%	55.1%	55.4%	55.4%	55.1%	55.1%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.50$	48.6%	48.7%	48.8%	48.7%	48.7%	48.8%	48.8%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.75$	33.4%	33.6%	33.6%	33.6%	33.6%	33.6%	33.6%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.25$	54.6%	54.9%	54.4%	54.9%	54.8%	54.4%	54.4%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.50$	44.2%	44.4%	44.3%	44.3%	44.4%	44.3%	44.3%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.75$	32.9%	33.2%	33.2%	33.2%	33.1%	33.2%	33.2%

Table III: Percentage of energy saved over the baseline when using different task ordering heuristics during the mapping phase under big failure rate, with $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.95$.

	random	*deW*	*inW*	*deMinW*	*deMaxW*	*inMinW*	*inMaxW*
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.25$	59.5%	59.3%	59.9%	59.4%	59.3%	59.4%	60.0%
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.50$	45.7%	45.8%	46.0%	45.7%	45.7%	45.9%	46.0%
$cor_{\text{task}} = 0.25, cor_{\text{proc}} = 0.75$	24.9%	25.0%	24.8%	25.0%	24.9%	24.9%	24.9%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.25$	57.6%	57.5%	57.7%	57.7%	57.6%	57.6%	57.7%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.50$	47.3%	47.3%	47.5%	47.3%	47.3%	47.4%	47.6%
$cor_{\text{task}} = 0.50, cor_{\text{proc}} = 0.75$	25.8%	25.7%	25.7%	25.7%	25.6%	25.7%	25.7%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.25$	55.9%	55.8%	56.0%	55.9%	55.8%	55.9%	56.0%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.50$	41.1%	40.9%	41.0%	40.9%	40.9%	41.0%	41.0%
$cor_{\text{task}} = 0.75, cor_{\text{proc}} = 0.75$	25.8%	25.9%	25.8%	25.8%	25.8%	25.8%	25.8%

Table IV: Percentage of energy saved over the baseline when using different task ordering heuristics during the mapping phase under small failure rate, with $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.98$.

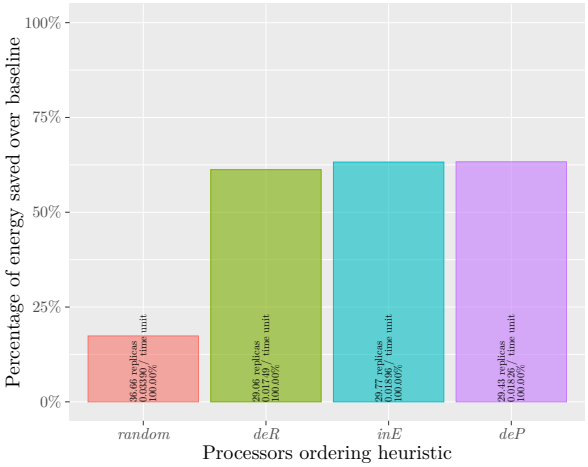


Figure 5.4: Percentage of energy saved over the baseline when using *deMinW* and different processor ordering heuristics during the mapping phase under big failure rate, with $cor_{\text{proc}} = 0.5$, $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.95$.

Processor ordering criteria for the mapping phase

Figures 5.4 and 5.5 show the performance of different processor ordering criteria. We see from this figure that our criteria perform globally well. Our heuristics, random excepted, can save around 60% of energy compared to the baseline scheme. Among the different criteria, *inE* and *deP* have similar performance, and this performance is better than that of *deR*. In the following, we will consider *deP* as the default criterion for ordering processors.

Scheduling heuristics

For the scheduling strategies, Figures 5.6 and 5.7 show that, after carefully selecting the mapping criterion, our heuristics can save up to 66.9% of the energy in the best case when compared to the random baseline. Among the different heuristics, we can find that, for small failure rate case, all heuristics perform similarly, and has a performance of around 63.8%. As for big failure rate case, the EDF_Start_Time heuristic, which chooses the earliest replica as primary, and which has an energy saving performance of 64.2%, is the worst of our original heuristics. On the contrary, EDF_Energy performs the best (with a performance of 66.9%). It saves 8.4% more energy than EDF_Plain, and the performance is only 2.7% worse than Smallest. Recall that Smallest sums up, for each task instance, only one replica which costs the least energy, without considering failures or replica overlapping. It is very likely that this bound is unreachable, especially when the system is under high pressure. EDF_WCET performs similarly to EDF_Energy, and EDF_Reliability performs slightly worse. In fact, as the energy consumption of a replica depends a lot on its WCET, it is reasonable that EDF_WCET and EDF_Energy tend to choose the same primary replica and have similar performance. This result shows that it is better to choose a replica which costs less energy as primary. In contrast, the earliest starting replica may not lead to minimal energy consumption, although we could have expected that it would induce less overlap among replicas.

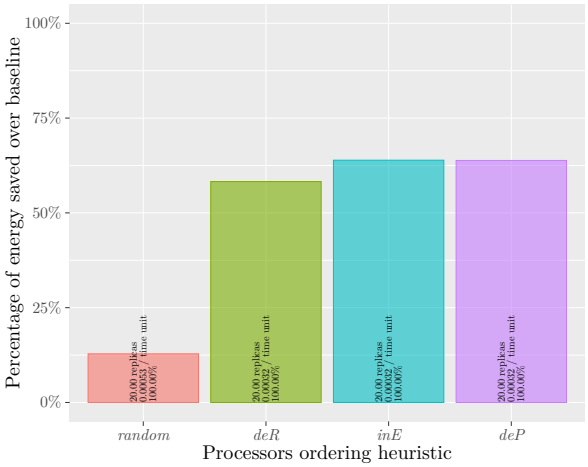


Figure 5.5: Percentage of energy saved over the baseline when using *deMinW* and different processor ordering heuristics during the mapping phase under small failure rate, with $cor_{\text{proc}} = 0.5$, $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.98$.

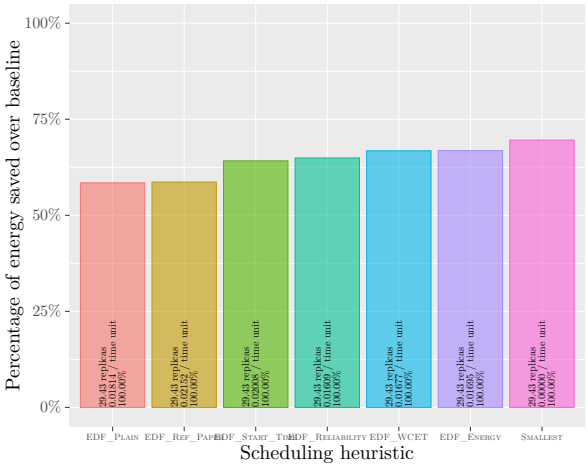


Figure 5.6: Percentage of energy saved over the baseline when using the *deMinW* and *deP* heuristics during the mapping phase and different criteria during the scheduling phase under big failure rate, with $cor_{\text{proc}} = 0.5$, $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.95$.

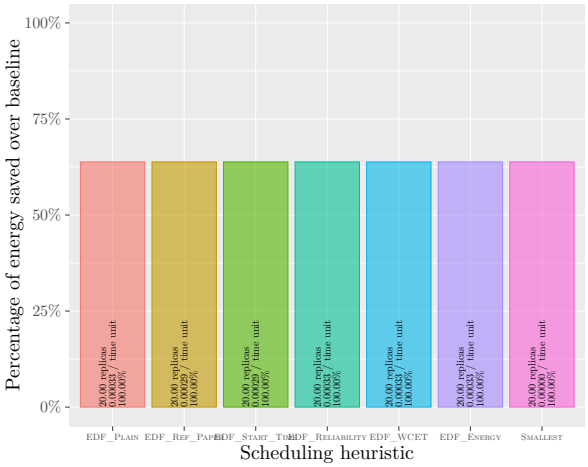


Figure 5.7: Percentage of energy saved over the baseline when using the *deMinW* and *deP* heuristics during the mapping phase and different criteria during the scheduling phase under big failure rate, with $cor_{\text{proc}} = 0.5$, $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.98$.

In the following paragraphs, we only consider EDF_Energy and we focus on the variation of its performance for the different parameters. We add EDF_Plain, EDF_Start_Time, and Smallest as references.

Task and processor correlation

Figures 5.8, 5.9, 5.10 and 5.11 show results when task correlation or processor correlation varies. We see that, when cor_{task} and cor_{proc} are not high (< 0.75), the performance remains relatively stable. The EDF_Energy strategy achieves an energy saving percentage over the random baseline of around 60%. This result is close to the Smallest and LowerBound references. When cor_{task} and cor_{proc} is high, we can observe that the performance decreases. In the case of $cor_{\text{task}} = 0.75$ or $cor_{\text{proc}} = 0.75$, the performance of EDF_Energy decreases respectively to 57.3% and 45.4% under big failure rate, and to 54.0% and 36.2% under small failure rate. But the difference with LowerBound remains small (3.8% and 7.2% respectively in big failure rate case, and under 1% in small failure rate case).

Figures 5.8 and 5.9 also show that, in the case of big failure rate, the performance of EDF_Start_Time is closer to that of EDF_Energy when cor_{task} or cor_{proc} is high. The reason is that, with a higher cor_{task} or cor_{proc}, the WCETs of different tasks on the same processor, or the WCETs of the same task on different processors, are more similar. In both cases, the orders of the processors for different tasks become more similar, and are more closely related to the power and/or the reliability parameters ($P_{k,d}$ and λ_k). Therefore, after the mapping phase, we have a few fully used processors, while the other processors have a low load. The overlapping increases, and the performance becomes worse. Inversely, when cor_{task} or cor_{proc} is not high, if we simply choose the replica which costs the least energy as the primary replica, the primary replicas are more randomly distributed on the different processors because of the high randomness of WCETs, and the overlapping is minimal. Then it is possible that we finish the primary early and with success, and delete all secondary replicas before starting the processing of any of them. This is why, when cor_{task} or cor_{proc} is high, the EDF_Energy strategy cannot save as much energy as in other cases.

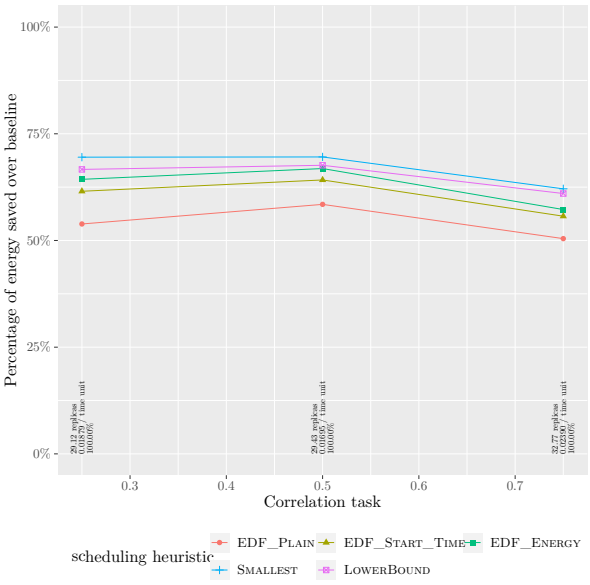


Figure 5.8: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under big failure rate, when varying cor_{task}, with $cor_{\text{proc}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.95$.

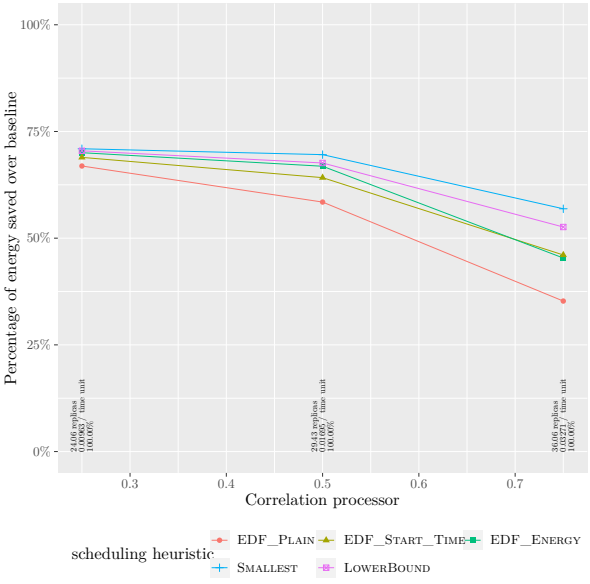


Figure 5.9: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under big failure rate, when varying cor_{proc}, with $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.95$.

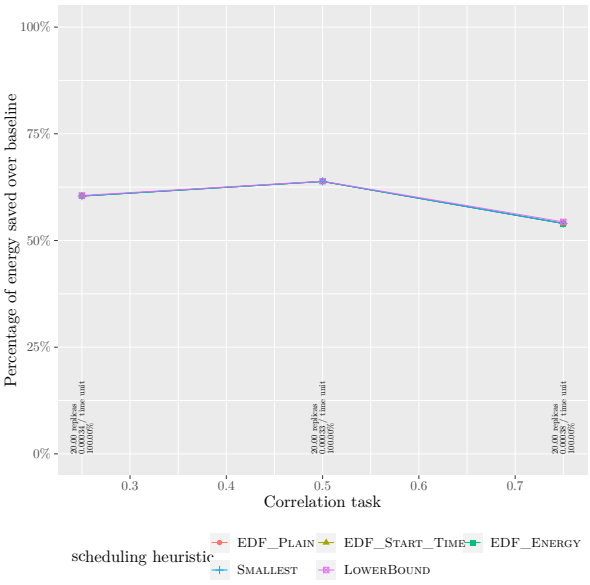


Figure 5.10: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under small failure rate, when varying cor_{task}, with $cor_{\text{proc}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.98$.

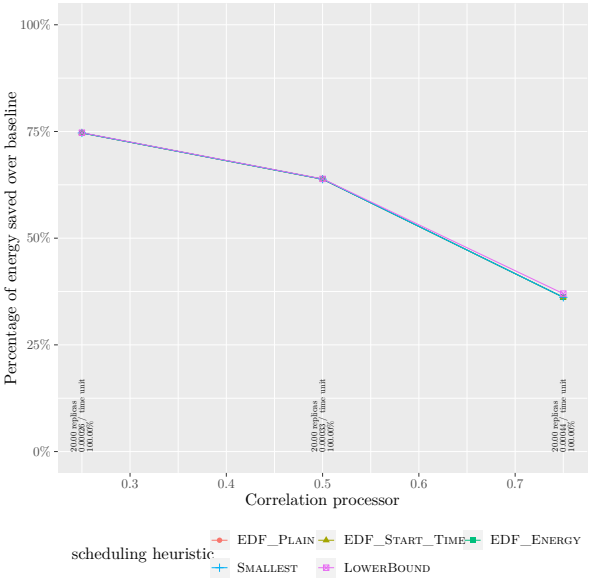


Figure 5.11: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under small failure rate, when varying cor_{proc}, with $cor_{\text{task}} = 0.5$, $basicWork = 0.3$, $\beta_{\text{b/w}} = 1$, and $\mathcal{R}_i = 0.98$.

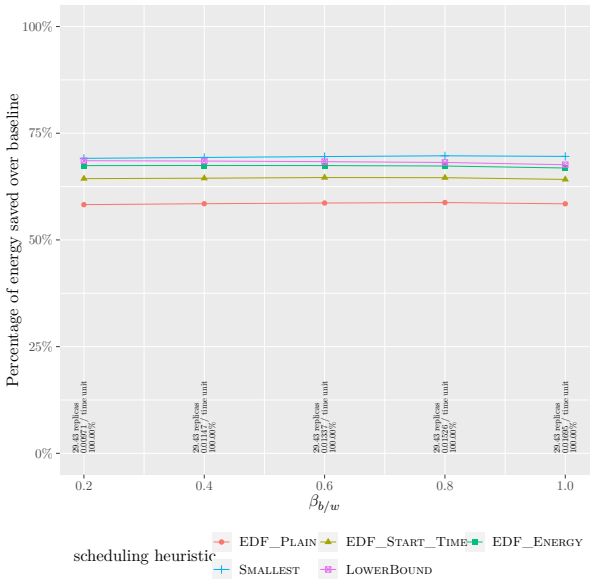


Figure 5.12: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under big failure rate, when varying $\beta_{\text{b/w}}$, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.95$.

Task variability

Figures 5.12 and 5.13 presents the results when $\beta_{\text{b/w}}$ varies. We observe that the result of each heuristic is (almost) independent of the value of $\beta_{\text{b/w}}$. This is because we map and schedule tasks based on their WCETs, so the mapping and scheduling results are independent of the value of $\beta_{\text{b/w}}$. Furthermore, each task τ_i has the same β_i on the different processors. Therefore the energy savings tend to be similar. More precisely, the performance of EDF_Energy varies under 0.5% when $\beta_{\text{b/w}}$ increases from 0.2 to 1. So we can conclude that the result is independent of $\beta_{\text{b/w}}$.

When $\beta_{\text{b/w}}$ is small, actual execution times can greatly differ from the WCETs which are used for mapping and scheduling. However, in this case, the heuristics perform as well as in the case $\beta_{\text{b/w}} = 1$. This shows that the heuristics are very robust.

Utilization and reliability threshold

On Figures 5.14 and 5.15, we observe the performance of the different heuristics when varying both *basicWork* and $\mathcal{R}_i$. We see that, in small failure rate case, all heuristics have similar performance while varying *basicWork* and $\mathcal{R}_i$. In contrast, in big failure rate case, when *basicWork* and $\mathcal{R}_i$ increase, the differences between heuristics become larger. EDF_Plain performs worse, while the performance of EDF_Energy remains stable. The difference between the Smallest and LowerBound references and our heuristics also increases with *basicWork* and $\mathcal{R}_i$. When $basicWork = 0.3$ and $\mathcal{R}_i = 0.95$, this difference of percentages is only of 1.5% for Smallest and 0.4% for LowerBound. In fact, with the increase of the system load, the WCETs increase, and each replica has a higher chance to be victim of a failure. But LowerBound does not consider the load of processors, and Smallest estimates the energy consumption without con-

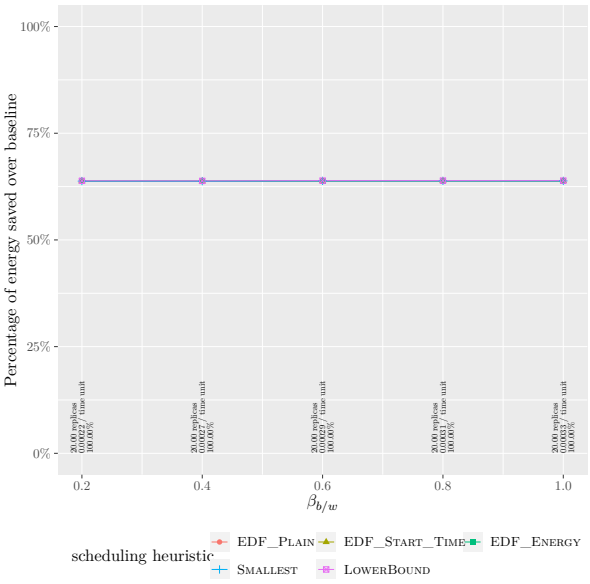


Figure 5.13: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under small failure rate, when varying $\beta_{\text{b/w}}$, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.98$.

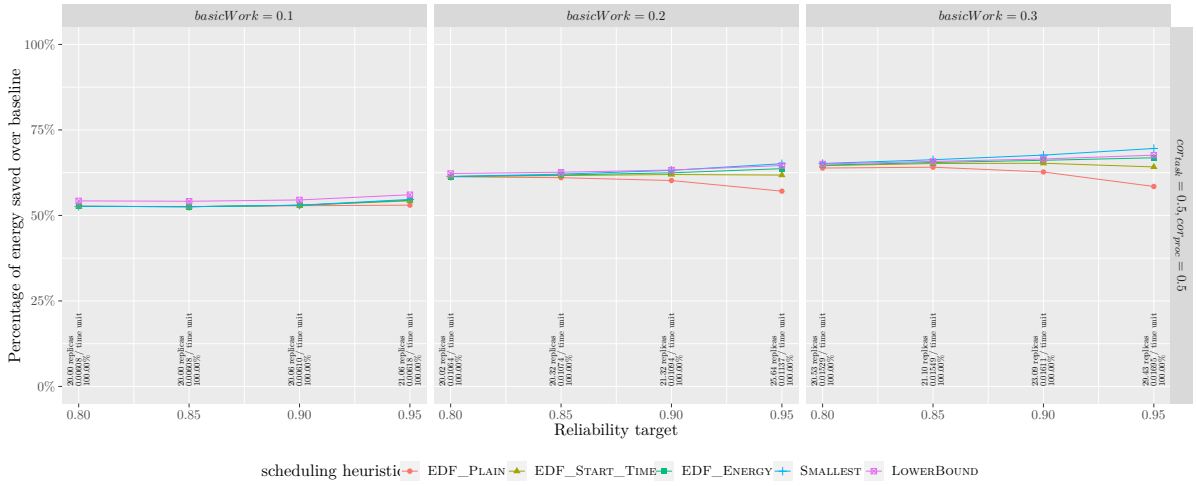


Figure 5.14: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under big failure rate, when varying *basicWork* and $\mathcal{R}_i$, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$ and $\beta_{\text{b/w}} = 1$.

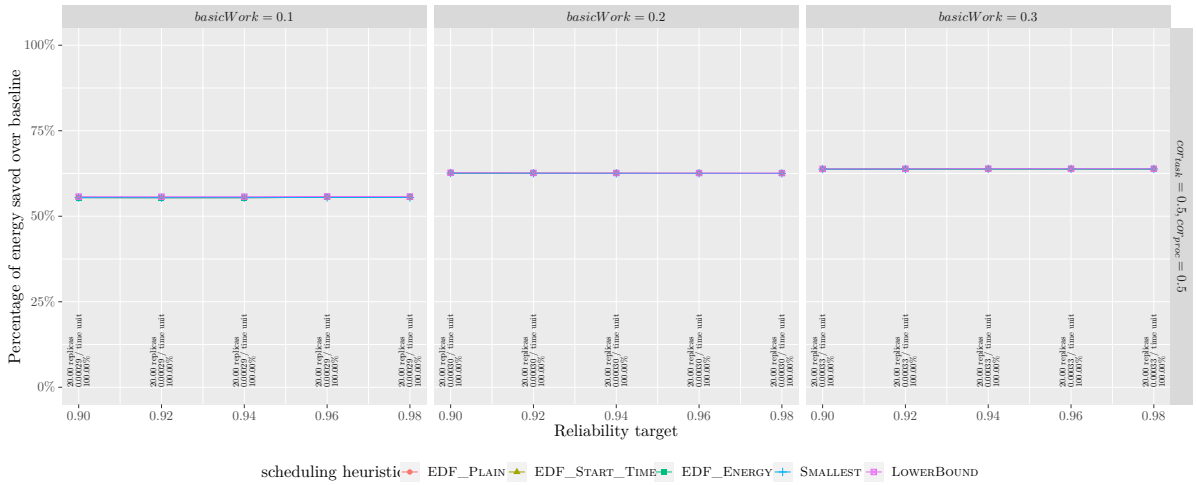


Figure 5.15: Percentage of energy saved over the baseline when using *deMinW* and *deP* during mapping for different scheduling criteria, under small failure rate, when varying *basicWork* and $\mathcal{R}_i$, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$ and $\beta_{\text{b/w}} = 1$.

sidering failures or overlapping. This explains the growing gap between these references and the heuristics when *basicWork* and $\mathcal{R}_i$ increase.

In summary, our strategy maintains a stable performance when *basicWork* and $\mathcal{R}_i$ increase. In contrast, EDF_Plain performs worse, especially in big failure rate case. The difference between EDF_Energy and the LowerBound reference is lower than 1% even in the worst case, which stresses the high performance of the designed strategy.

Number of replicas, number of failures and success rate

For each experiment, we counted the number of replicas used for different heuristics, number of replicas that failed during the execution, and the success rate which means the percentage of experiments for which we can find feasible mapping.

In the small failure rate case, there are low number of replicas (slightly higher than 20 replicas per execution), little failure (on average 0.0002 failed replicas per time unit) and high success rate (100% of experiments find feasible mapping), it is more difficult to find a variation through parameters using this experimental set. Hence, we will focus on big failure rate case in this section.

In the set with big failure rate, the average number of failed replicas per time unit increases to 0.0129. In Figure 5.4, we observe that *random* needs more replicas and is the victim of a larger number of failures. *deR* achieves the lowest total number of replicas and total number of failures encountered because it always uses the processors with highest reliability. *deP* achieves a slightly lower number of replicas and failures than *inE* because it considers both reliability and energy. In Figure 5.6, the total number of replicas is the same for all criteria because they all use the same mapping. For the number of failures, the difference is not obvious between strategies. Unsurprisingly, EDF_Reliability has the lowest failure rate. Then, EDF_WCET and EDF_Energy have slightly better failure rates than the other heuristics. Figures 5.8 and 5.9 show that the number of replicas and failures remain relatively stable when cor_{task} or cor_{proc} is not high. But there is a relative increase when cor_{task} or cor_{proc} reaches 0.75. As for $\beta_{\text{b/w}}$, we

	$cor_{\text{task}} = 0.25$	$cor_{\text{task}} = 0.50$	$cor_{\text{task}} = 0.75$
$cor_{\text{proc}} = 0.25$	100%	100%	100%
$cor_{\text{proc}} = 0.50$	99.971%	100%	100%
$cor_{\text{proc}} = 0.75$	99.111%	99.992%	100%

Table V: Success rate of random trials under big failure rate, when varying cor_{task} and cor_{proc}.

	$basicWork = 0.1$	$basicWork = 0.2$	$basicWork = 0.3$
$\mathcal{R} = 0.80$	100%	100%	100%
$\mathcal{R} = 0.85$	100%	100%	100%
$\mathcal{R} = 0.90$	100%	100%	99.940%
$\mathcal{R} = 0.95$	100%	100%	98.825%

Table VI: Success rate of random trials under big failure rate, when varying cor_{task} and cor_{proc}.

see on Figure 5.12 that, when $\beta_{\text{b/w}}$ increases, the number of replicas remains the same, while the number of failures increases. This is because we map replicas according to the task WCET which is independent on the value of $\beta_{\text{b/w}}$. However, when $\beta_{\text{b/w}}$ is larger, we have longer actual execution times and, thus, a higher probability that failures actually happen. While increasing $\mathcal{R}_i$ and *basicWork*, we observe in Figure 5.14 that the number of replicas increases, and also the number of failures. With higher $\mathcal{R}_i$, we need more replicas to satisfy the reliability target, and more replicas executed at the same time means a higher probability that failures actually occur. As for *basicWork*, a higher *basicWork* means larger average WCETs. Longer replicas lead to lower reliability. Thus failures happen more frequently, and we need also more replicas to reach the reliability threshold.

The success rate of big failure rate case can be found in Tables V and VI. We observe that the proportion of experiments for which we can find a feasible mapping decreases when the system pressure becomes large. The fraction of instances for which we can build a solution also decreases when cor_{task} is low and cor_{proc} is high. This is why we used $\mathcal{R}_i = 0.95$ and $basicWork = 0.3$ as the highest reliability target and basic work parameter, and we avoided to use extreme correlation values in our experiments. Within all platforms in the experiments, all the tested heuristics are able to find a valid solution in all tested configurations with small failure rate. And in the big failure rate cases, heuristics are able to build valid solutions for more than 99.8% of the instances. The very high success rate of our experiments means that we can be confident in our results and that they do not suffer of any bias.

Summary

In conclusion, in the experiments we compared different criteria for the mapping and scheduling phases, and we also studied the influence of the different parameters. Tables VII, VIII, IX and X present a synthetic view of the results, achieved by different heuristics, or for different correlations, under big or small failure rate, and when the system is under the highest pressure ($\beta_{\text{b/w}} = 1$, $basicWork = 0.3$, $\mathcal{R}_i = 0.95$ for big failure rate and $\mathcal{R}_i = 0.98$ for small failure rate).

Among the different criteria used in the heuristics, we observe from Tables VII and IX that the *deP* method is the best processor ordering for the mapping phase, and *inE* performs similar to *deP* in most of the cases. As for scheduling phase, all heuristics perform similarly in

	random	*deR*	*inE*	*deP*
EDF_Plain	6.9%	57.2%	58.2%	58.5%
EDF_Ref_Paper	8.8%	57.3%	58.1%	58.7%
EDF_Start_Time	17.0%	62.1%	64.0%	64.2%
EDF_Reliability	23.6%	62.1%	65.3%	64.9%
EDF_WCET	23.8%	64.2%	66.8%	66.8%
EDF_Energy	24.2%	64.4%	66.8%	66.9%
LowerBound	67.6%			
Smallest	51.8%	67.4%	69.7%	69.6%

Table VII: Percentage of energy saved over the baseline when using different processor ordering criteria during mapping and using different scheduling criteria under big failure rate, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$, $\beta_{\text{b/w}} = 1$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.95$.

	$cor_{\text{task}} = 0.25$	$cor_{\text{task}} = 0.50$	$cor_{\text{task}} = 0.75$
$cor_{\text{proc}} = 0.25$	72.4%	70.0%	67.1%
$cor_{\text{proc}} = 0.50$	64.3%	66.9%	57.3%
$cor_{\text{proc}} = 0.75$	42.6%	45.4%	46.4%

Table VIII: Percentage of energy saved over the baseline when using the *deMinW* task ordering criterion and the *deP* processor ordering criterion during mapping, and using the EDF_Energy scheduling criterion, under big failure rate, when varying cor_{task} and cor_{proc}, with $\beta_{\text{b/w}} = 1$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.95$.

	random	*deR*	*inE*	*deP*
EDF_Plain	12.8%	58.3%	63.9%	63.8%
EDF_Ref_Paper	12.8%	58.3%	63.9%	63.8%
EDF_Start_Time	12.8%	58.3%	63.9%	63.8%
EDF_Reliability	12.8%	58.3%	63.9%	63.8%
EDF_WCET	12.8%	58.3%	63.9%	63.8%
EDF_Energy	12.8%	58.3%	63.9%	63.8%
LowerBound	63.9%			
Smallest	12.8%	58.3%	63.9%	63.8%

Table IX: Percentage of energy saved over the baseline when using different processor ordering criteria during mapping and using different scheduling criteria under small failure rate, with $cor_{\text{task}} = 0.5$, $cor_{\text{proc}} = 0.5$, $\beta_{\text{b/w}} = 1$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.98$.

	$cor_{\text{task}} = 0.25$	$cor_{\text{task}} = 0.50$	$cor_{\text{task}} = 0.75$
$cor_{\text{proc}} = 0.25$	78.4%	74.7%	71.4%
$cor_{\text{proc}} = 0.50$	60.4%	63.8%	54.0%
$cor_{\text{proc}} = 0.75$	35.4%	36.2%	36.0%

Table X: Percentage of energy saved over the baseline when using the *deMinW* task ordering criterion and the *deP* processor ordering criterion during mapping, and using the EDF_ENERGY scheduling criterion, under small failure rate, when varying cor_{task} and cor_{proc}, with $\beta_{\text{b/w}} = 1$, $basicWork = 0.3$, and $\mathcal{R}_i = 0.98$.

small failure rate case, and EDF_ENERGY performs the best among the scheduling criteria in big failure rate case. Respectively under big or small failure rate, in the case of high system pressure, our best criterion can still achieve 66.9% or 63.9% of energy saving compared to the random baseline. While using the best criterion, the difference of performance with *random* is 42.7% and 51.0% respectively under big and small failure rate. The difference is only 2.7% compared to the SMALLEST reference in big failure rate case. And in the case of small failure rate, this difference with SMALLEST is very close to 0. To study the efficiency of our selected heuristic under different cor_{task} and cor_{proc}, Tables VIII and X present the performance while using *deMinW*, *deP* and EDF_ENERGY as criteria. Our strategies can save more than 35% of the energy consumed by the baseline in all cases, and this percentage can be higher than 70% in the best case. Furthermore, in big failure rate case, we report a median energy saving percentage of only 1.6% less than that of the LOWERBOUND; and an average of only 2.0% less. In small failure rate case, these two values can both be reduced to 0.3%.

We can confidently conclude that our best strategies perform remarkably well over the whole experimental settings.

5.7 Conclusion

In this work, we have studied the problem of executing periodic real-time tasks on an heterogeneous platform, with several objectives: minimizing energy consumption, guaranteeing reliability thresholds, and meeting all deadlines. For each task, we decide how many replicas should be launched, and on which processors to map them. We tagged one replica per task as "primary" replica and the other ones as "secondary" replicas. To obtain an absolute measure for the evaluation of our heuristics, we have computed a theoretical lower-bound on energy consumption.

Extensive simulations show that our best heuristics always achieve very good performance, very close to the LOWERBOUND (on average the percentage of energy saving of our best heuristic is only 2% less than that of LOWERBOUND). This performance was reached by considering processors in the *deP* order when mapping the replicas of a task (roughly speaking, *deP* is the ratio of a task failure rate by its energy cost), and by tagging as "primary" the replicas which cost the smallest dynamic energy.

Furthermore, while all decisions are taken with worst-case execution times (WCETs) of tasks as only input, the simulations used actual execution times; the best heuristic always achieved excellent performance even when the actual execution times were far smaller than the WCETs, showing the robustness of our approach.

Conclusion

Summary of results

In this thesis, we have designed scheduling heuristics for independent tasks under budget and time constraints, in order to satisfy different requirements. The first three chapters of this thesis have performance as objective, while the fourth chapter focuses on energy efficiency.

The first three chapters have a common framework: We have a set of tasks whose execution times follow the same probability distribution. We can decide at any instant to interrupt the execution of a long running task and to launch a new one instead. The main questions are how many (or which) processors to enroll, and whether and when to interrupt tasks if they have been executing for a long time. In a previous work, the problem has been dealt with on a homogeneous platform and with the same release time and deadline for all tasks. Our work extends the state-of-the-art in three directions: In the first work, we consider an heterogeneous platform. In the second work, we assume that the distribution of task execution times is unknown. In the third work, we consider real-time tasks which are released periodically and have their own deadlines.

The fourth chapter considers a more complicated framework: We have periodic tasks and an heterogeneous platform. We consider transient faults. Tasks are replicated to guarantee a pre-defined reliability threshold. We aim at finding a heuristic which minimizes the expected energy consumption, while matching the deadline and reliability constraints of all tasks.

Our contributions in each chapter are summarized in the following paragraphs:

Task scheduling on heterogeneous platform This chapter dealt with the problem of scheduling stochastic tasks on an heterogeneous platform under deadline and budget constraints. The main difficulty was to select the best subset of processors in order to maximize the expected number of tasks successfully executed. We proved that this problem is NP-hard, and designed Greedy, a greedy algorithm which is proved to be a 2-approximation. In the Greedy algorithm, we sorted the processors by non-increasing yield and calculated which ones to enroll, based on the budget and deadline. Through the experiments with a variety of parameter sets, we found that Greedy significantly outperforms the other heuristics, and its performance is very close to the upper bound established in this work.

Non-clairvoyant task scheduling In this chapter, we considered a similar framework as in the previous work and in Chapter 2. Previous results showed that long-lasting tasks must be interrupted at some optimal cutting threshold, calculated according to the distribution of task execution times. But in this work, we assumed that this distribution is unknown. Therefore, we designed a set of scheduling heuristics to estimate the cutting threshold, some of which making use of the Kaplan-Meier estimator. We also introduced several methods to recompute

the threshold value while new data are observed. We found in the experiments that our best heuristic AutoPerSurvival (40%, 0.01) can execute 79% of tasks that an omniscient oracle which knows the distribution of task execution times would be able to complete.

Firm real-time task scheduling In this chapter, we considered firm real-time tasks, which arrive periodically and have their proper deadlines. We continue to study the problem of maximizing the number of tasks successfully completed. In this work, we designed several heuristics which decide, on which processor to allocate each task, whether and when to interrupt a (long-lasting) task, and which task in the waiting list should be launched when a processor is idle. As for theoretical results, we discretized the time and constructed a Markov Chain. We showed that the chain is aperiodic and irreducible, and computed the asymptotic expected performance via the limit distribution of the chain. On the practical side, we launched an extensive set of experiments, which showed that EarliestStartTime and Smax respectively outperform other task attribution and task admission heuristics, and can achieve a gain up to 30% compared to the RoundRobin and NeverKill heuristic.

Multi-criteria real-time task scheduling In this work, we continued to consider real-time tasks. But differently from Chapter 4, we assumed that the task execution times are limited by the Worst Case Execution Time (WCET). We studied the problem of executing real-time tasks on heterogeneous platforms with three objectives: minimizing energy consumption, guaranteeing reliability thresholds, and meeting deadlines. We designed a mapping-scheduling heuristic and several criteria for each phase. We also characterized the lower-bound of energy consumption in our problem. We conducted a comprehensive set of experiments and found that our best heuristic (*deP* for processor ordering and EDF_Energy for scheduling) achieves a performance very close to the lower-bound. On average, the percentage of energy saving by our best heuristic is only 2% less than that of LowerBound.

Future work and perspectives

We plan to continue our research along the following directions:

Firstly, we can extend the target platform to a more complex study. In Chapters 3 and 4, we consider a homogeneous platform in the current work. Future work will be dedicated to considering heterogeneous platforms, as what we do in Chapters 2 and 5. Typically, cloud providers provide different categories of processors with different power and cost. This heterogeneity will dramatically complicate the selection of a good processor subset and the estimation of the cutting threshold for the work in Chapter 3. As for the work in Chapter 4, the analysis of processor states and the allocation of task to processor will also become more complicated. Chapters 2 and 3 assume that the unit cost of a processor is proportional to the running time. We can extend the pricing model to take more complex scenarios into account, for example considering start-up costs (which would limit the number of enrolled processors), or non-constant costs that depend on the time and day, or on the load of the cloud platform. In Chapter 5, we assume that processors have only one available frequency. We can extend our heuristics to processors with multiple frequency levels.

Secondly, we can also extend our heuristics with a more general task model. Chapters 2, 3 and 4 consider tasks whose execution time follows a common probability distribution. We start the trial of multimodal distributions in Chapters 3 and 4. We can continue to study the

problem while different types of tasks need to be launched. Our tasks in Chapters 4 and 5 have a period completely fixed. We can add randomness to their period or we can consider tasks with random release times. This randomness is challenging for the estimation of processor states. In this thesis, we studied different problems with independent tasks. Future work can aim at extending the algorithms to graphs of tasks instead of independent task sets. The dependences between tasks will dramatically complicate the mapping and scheduling problems, because we need to enforce all graph dependences. In addition, we can extend the model to include communication cost, data storage cost, response time of tasks, etc., in order to make it closer to a real application.

Bibliography

[1] O. Aalen, O. Borgan, and H. Gjessing. *Survival and event history analysis: a process point of view*. Springer Science & Business Media, 2008.

[2] S. Abrishami, M. Naghibzadeh, and D. H. Epema. "Deadline-constrained workflow scheduling algorithms for Infrastructure as a Service Clouds". In: *Future Generation Computer Systems* 29.1 (2013). Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures, pp. 158–169. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2012.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0167739X12001008.

[3] M. Amirijoo, J. Hansson, and S. H. Son. "Specification and management of QoS in real-time databases supporting imprecise computations". In: *IEEE Trans. Computers* 55.3 (2006), pp. 304–319.

[4] V. Arabnejad, K. Bubendorfer, and B. Ng. "Budget distribution strategies for scientific workflow scheduling in commercial clouds". In: *12th IEEE International Conference on e-Science*. Oct. 2016, pp. 137–146. DOI: 10.1109/eScience.2016.7870894.

[5] G. Bernat, A. Burns, and A. Liamosi. "Weakly hard real-time systems". In: *IEEE Trans. Computers* 50.4 (2001), pp. 308–321.

[6] G. Buttazzo. "Handling Overload Conditions in Real-Time Systems". In: *Real-Time Systems, Architecture, Scheduling, and Application*. Ed. by S. M. Babamir. Rijeka: InTech, 2012. Chap. 7. DOI: 10.5772/37265. URL: https://doi.org/10.5772/37265.

[7] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng. "Cost optimized provisioning of elastic resources for application workflows". In: *Future Generation Computer Systems* 27.8 (2011), pp. 1011–1026. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2011.05.001. URL: http://www.sciencedirect.com/science/article/pii/S0167739X11000744.

[8] Z. Cai, X. Li, R. Ruiz, and Q. Li. "A delay-based dynamic scheduling algorithm for bag-of-task workflows with stochastic task execution times in clouds". In: *Future Generation Computer Systems* 71 (2017), pp. 57–72. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.01.020. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17300870.

[9] R. N. Calheiros and R. Buyya. "Meeting Deadlines of Scientific Workflows in Public Clouds with Tasks Replication". In: *IEEE Transactions on Parallel and Distributed Systems* 25.7 (July 2014), pp. 1787–1796. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.238.

[10] Y. Caniou, E. Caron, A. Kong Win Chang, and Y. Robert. "Budget-aware scheduling algorithms for scientific workflows with stochastic task weights on heterogeneous IaaS cloud platforms". In: *27th International Heterogeneity in Computing Workshop HCW 2013*. IEEE Computer Society Press, 2018.

[11] L.-C. Canon, M. El Sayah, and P.-C. Héam. "A markov chain monte carlo approach to cost matrix generation for scheduling performance evaluation". In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2018, pp. 460–467.

[12] L.-C. Canon, A. Kong Win Chang, Y. Robert, and F. Vivien. "Scheduling independent stochastic tasks under deadline and budget constraints". In: *SBAC-PAD*. IEEE, 2018.

[13] L.-C. Canon, A. Kong Win Chang, Y. Robert, and F. Vivien. "Scheduling independent stochastic tasks under deadline and budget constraints". In: *Int. J. High Performance Computing Applications* 34.2 (2019), pp. 246–264.

[14] L.-C. Canon, A. Kong Win Chang, F. Vivien, and Y. Robert. *Code for Scheduling independent stochastic tasks under deadline and budget constraints*. https://doi.org/10.6084/m9.figshare.6463223.v2. June 2018. DOI: 10.6084/m9.figshare.6463223.v2.

[15] L.-C. Canon and L. Philippe. "On the heterogeneity bias of cost matrices for assessing scheduling algorithms". In: *IEEE Trans. Par. Dist. Syst.* 28.6 (2017), pp. 1675–1688.

[16] H. Casanova, M. Gallet, and F. Vivien. "Non-clairvoyant Scheduling of Multiple Bag-of-Tasks Applications". In: *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*. 2010, pp. 168–179. DOI: 10.1007/978-3-642-15277-1_17. URL: https://doi.org/10.1007/978-3-642-15277-1_17.

[17] L. E. Chambless and G. Diao. "Estimation of time-dependent area under the ROC curve for long-term risk prediction". In: *Statistics in Medicine* 25.20 (2006), pp. 3474–3486. DOI: 10.1002/sim.2299. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim.2299. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.2299.

[18] H. Chen, J. Wen, W. Pedrycz, and G. Wu. "Big Data Processing Workflows Oriented Real-Time Scheduling Algorithm using Task-Duplication in Geo-Distributed Clouds". In: *IEEE Trans. Big Data* 6.1 (2020), pp. 131–144.

[19] J. Y. Chung, J. W. S. Liu, and K. J. Lin. "Scheduling periodic jobs that allow imprecise results". In: *IEEE Trans. Computers* 39.9 (1990), pp. 1156–1174.

[20] A. Chydzinski. "Queues with the dropping function and non-Poisson arrivals". In: *IEEE Access* 8 (2020), pp. 39819–39829.

[21] R. I. Davis and A. Burns. "A Survey of Hard Real-time Scheduling for Multiprocessor Systems". In: *ACM Comput. Surv.* 43.4 (Oct. 2011), 35:1–35:44. ISSN: 0360-0300. DOI: 10.1145/1978802.1978814. URL: http://doi.acm.org/10.1145/1978802.1978814.

[22] C. Denninnart, J. Gentry, A. Mokhtari, and M. A. Salehi. "Efficient task pruning mechanism to improve robustness of heterogeneous computing systems". In: *Journal of Parallel and Distributed Computing* (2020).

[23] C. Denninnart, J. Gentry, and M. A. Salehi. "Improving robustness of heterogeneous serverless computing systems via probabilistic task pruning". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 6–15.

[24] H. M. Fard, R. Prodan, and T. Fahringer. "A Truthful Dynamic Workflow Scheduling Mechanism for Commercial Multicloud Environments". In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (June 2013), pp. 1203–1212. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.257.

[25] D. Feitelson. "Workload modeling for computer systems performance evaluation". In: *Version 1.0.3* (2014), pp. 1–607.

[26] C.-W. Feng, L.-F. Huang, C. Xu, and Y.-C. Chang. "Congestion control scheme performance analysis based on nonlinear RED". In: *IEEE Systems Journal* 11.4 (2015), pp. 2247–2254.

[27] W. Feng and J. W. S. Liu. "An extended imprecise computation model for time-constrained speech processing and generation". In: *Proc. IEEE Workshop on Real-Time Applications*. May 1993, pp. 76–80. DOI: 10.1109/RTA.1993.263112.

[28] T. S. Ferguson. *Optimal stopping and applications*. UCLA Press, 2008.

[29] S. Floyd and V. Jacobson. "Random early detection gateways for congestion avoidance". In: *IEEE/ACM Transactions on networking* 1.4 (1993), pp. 397–413.

[30] Y. Gao, Y. Wang, S. K. Gupta, and M. Pedram. "An energy and deadline aware resource provisioning, scheduling and optimization framework for cloud systems". In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Sept. 2013. DOI: 10.1109/CODES-ISSS.2013.6659018.

[31] Y. Gao. *Heterogeneous real-time systems*. https://figshare.com/articles/software/Minimizing_energy_consumption_for_real-time_tasks_on_heterogeneous_platforms_under_deadline_and_reliability_constraints/14450538. Apr. 2021.

[32] Y. Gao. *Resource-Constrained Scheduling of Stochastic Tasks With Unknown Probability Distribution*. Nov. 2020. DOI: 10.6084/m9.figshare.13187135.v1. URL: https://figshare.com/articles/software/Resource-Constrained_Scheduling_of_Stochastic_Tasks_With_Unknown_Probability_Distribution/13187135/1.

[33] Y. Gao, L.-C. Canon, Y. Robert, and F. Vivien. *Code to schedule stochastic tasks on heterogeneous platforms*. https://figshare.com/articles/Code_to_schedule_stochastic_tasks_on_heterogeneous_platforms/7777046/3. Feb. 2019. DOI: 10.6084/m9.figshare.7777046.v3.

[34] Y. Gao, L. Han, J. Liu, Y. Robert, and F. Vivien. *Minimizing energy consumption for real-time tasks on heterogeneous platforms under deadline and reliability constraints*. Research report 9403. INRIA, Apr. 2021.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[36] J. Gentry, C. Denninnart, and M. A. Salehi. "Robust dynamic resource allocation via probabilistic task pruning in heterogeneous computing systems". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 375–384.

[37] A. Grekioti and N. V. Shakhlevich. "Scheduling Bag-of-Tasks Applications to Optimize Computation Time and Cost". In: *PPAM*. Vol. 8385. LNCS. Springer, 2014.

[38] T. Guo, J. Liu, W. Hu, and M. Wei. "Energy-Aware Fault-Tolerant Scheduling Under Reliability and Time Constraints in Heterogeneous Systems". In: *Intelligent Computing Methodologies*. Ed. by D.-S. Huang, M. M. Gromiha, K. Han, and A. Hussain. Springer, 2018, pp. 36–46.

[39] Y. Guo, D. Zhu, H. Aydin, J.-J. Han, and L. T. Yang. "Exploiting primary/backup mechanism for energy efficiency in dependable real-time systems". In: *Journal of Systems Architecture* 78 (2017), pp. 68–80. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.06.008. URL: http://www.sciencedirect.com/science/article/pii/S1383762116302624.

[40] P. Guyot, A. E. Ades, M. J. Ouwens, and N. J. Welton. "Enhanced secondary analysis of survival data: reconstructing the data from published Kaplan-Meier survival curves". In: *BMC Medical Research Methodology* 12.1 (Feb. 2012), p. 9. ISSN: 1471-2288. DOI: 10.1186/1471-2288-12-9. URL: https://doi.org/10.1186/1471-2288-12-9.

[41] O. Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, 2002.

[42] J.-J. Han, W. Cai, and D. Zhu. "Resource-aware Partitioned Scheduling for Heterogeneous Multicore Real-time Systems". In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. San Francisco, California: ACM, 2018, 124:1–124:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3196103. URL: http://doi.acm.org/10.1145/3195970.3196103.

[43] L. Han, L.-C. Canon, J. Liu, Y. Robert, and F. Vivien. "Improved energy-aware strategies for periodic real-time tasks under reliability constraints". In: *40th IEEE Real-Time Systems Symposium (RTSS)*. 2020.

[44] Q. Han. "Energy-aware Fault-tolerant Scheduling for Hard Real-time Systems". PhD thesis. Florida International University, 2015. DOI: 10.25148/etd.FIDC000077.

[45] M. A. Haque, H. Aydin, and D. Zhu. "On reliability management of energy-aware real-time systems through task replication". In: *IEEE Transactions on Parallel and Distributed Systems* 28.3 (2017), pp. 813–825.

[46] M. Harchol-Balter. "Open problems in queueing theory inspired by datacenter computing". In: *Queueing Systems* 97.1 (2021), pp. 3–37.

[47] I. A. T. Hashem, N. B. Anuar, M. Marjani, E. Ahmed, H. Chiroma, A. Firdaus, M. T. Abdullah, F. Alotaibi, W. K. M. Ali, I. Yaqoob, and A. Gani. "MapReduce scheduling algorithms: a review". In: *The Journal of Supercomputing* 76.7 (July 2020), pp. 4915–4945. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2719-5. URL: https://doi.org/10.1007/s11227-018-2719-5.

[48] H. Hassan, J. Simó, and A. Crespo. "Flexible real-time mobile robotic architecture based on behavioural models". In: *Engineering Applications of Artificial Intelligence* 14.5 (2001), pp. 685–702. ISSN: 0952-1976. DOI: https://doi.org/10.1016/S0952-1976(01)00029-X. URL: http://www.sciencedirect.com/science/article/pii/S095219760100029X.

[49] K. Huang, X. Jiang, X. Zhang, R. Yan, K. Wang, D. Xiong, and X. Yan. "Energy-Efficient Fault-Tolerant Mapping and Scheduling on Heterogeneous Multiprocessor Real-Time Systems". In: *IEEE Access* 6 (2018), pp. 57614–57630. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2873641.

[50] E. Hwang and K. H. Kim. "Minimizing Cost of Virtual Machines for Deadline-Constrained MapReduce Applications in the Cloud". In: *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*. GRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 130–138. ISBN: 978-0-7695-4815-9. DOI: 10.1109/Grid.2012.19. URL: http://dx.doi.org/10.1109/Grid.2012.19.

[51] S. Im, J. Kulkarni, and K. Munagala. "Competitive Algorithms from Competitive Equilibria: Non-Clairvoyant Scheduling under Polyhedral Constraints". In: *J. ACM* 65.1 (Dec. 2017). ISSN: 0004-5411. DOI: 10.1145/3136754. URL: https://doi.org/10.1145/3136754.

[52] F. Jumel and F. Simonot-Lion. "Management of anytime tasks in real time applications". In: *XIV Workshop on Supervising and Diagnostics of Machining Systems*. Karpacz/Pologne, 2003. URL: https://hal.inria.fr/inria-00099612.

[53] T. Kaldewey, C. Lin, and S. Brandt. "Firm Real-Time Processing in an Integrated Real-Time System". In: *12th IEEE Real-Time and EmbeddedTechnology and Applications Symposium*. 2006.

[54] E. L. Kaplan and P. Meier. "Nonparametric Estimation from Incomplete Observations". In: *Journal of the American Statistical Association* 53.282 (1958), pp. 457–481. DOI: 10.1080/01621459.1958.10501452.

[55] H. Kobayashi and N. Yamasaki. "RT-Frontier: a real-time operating system for practical imprecise computation". In: *10th IEEE Real-Time and Embedded Tech. Appl. Symp.* May 2004, pp. 255–264. DOI: 10.1109/RTTAS.2004.1317271.

[56] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd. USA: Kluwer, 2011.

[57] G. Koren and D. Shasha. "Skip-Over: algorithms and complexity for overloaded systems that allow skips". In: *Proc. 16th IEEE Real-Time Systems Symposium*. IEEE, 1995, pp. 110–117.

[58] K. Li. "Non-clairvoyant scheduling of independent parallel tasks on single and multiple multicore processors". In: *Journal of Parallel and Distributed Computing* 133 (2019), pp. 210–220. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2018.06.001. URL: http://www.sciencedirect.com/science/article/pii/S0743731518303988.

[59] C. Lin, T. Kaldewey, A. Povzner, and S. Brandt. "Diverse Soft Real-Time Processing in an Integrated System". In: *27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 2006, pp. 369–378.

[60] *List of Green500 November 2020*. https://www.top500.org/lists/green500/2020/11/.

[61] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[62] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. Yu, J. Y. Chung, and W. Zhao. "Algorithms for Scheduling Imprecise Computations". In: *Foundations of Real-Time Computing: Scheduling and Resource Management*. Ed. by A. M. van Tilborg and G. M. Koob. Springer, 1991, pp. 203–249. ISBN: 978-1-4615-3956-8. DOI: 10.1007/978-1-4615-3956-8_8. URL: https://doi.org/10.1007/978-1-4615-3956-8_8.

[63] K. Liu, H. Jin, J. Chen, X. Liu, D. Yuan, and Y. Yang. "A Compromised-Time-Cost Scheduling Algorithm in SwinDeW-C for Instance-Intensive Cost-Constrained Workflows on a Cloud Computing Platform". In: *Int. J. High Performance Computing Applications* 24.4 (2010), pp. 445–456. DOI: 10.1177/1094342010369114. eprint: https://doi.org/10.1177/1094342010369114. URL: https://doi.org/10.1177/1094342010369114.

[64] O. Lopez. "A generalization of the Kaplan–Meier estimator for analyzing bivariate mortality under right-censoring and left-truncation with applications in model-checking for survival copula models". In: *Insurance: Mathematics and Economics* 51.3 (2012), pp. 505–516. ISSN: 0167-6687. DOI: https://doi.org/10.1016/j.insmatheco.2012.07.009. URL: http://www.sciencedirect.com/science/article/pii/S0167668712000881.

[65] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. "Cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, Nov. 2012, pp. 1–11.

[66] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. "Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds". In: *Future Gen. Comp. Syst.* 48 (2015), pp. 1–18.

[67] M. Mao, J. Li, and M. Humphrey. "Cloud auto-scaling with deadline and budget constraints". In: *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE, Oct. 2010, pp. 41–48. DOI: 10.1109/GRID.2010.5697966.

[68] A. Marchand and M. Chetto. "Dynamic scheduling of periodic skippable tasks in an overloaded real-time system". In: *2008 IEEE/ACS International Conference on Computer Systems and Applications*. 2008, pp. 456–464.

[69] J. Meng, S. Chakradhar, and A. Raghunathan. "Best-effort parallel execution framework for Recognition and mining applications". In: *IPDPS*. IEEE, 2009. DOI: 10.1109/IPDPS.2009.5160991.

[70] A. Mokhtari, C. Denninnart, and M. A. Salehi. "Autonomous Task Dropping Mechanism to Achieve Robustness in Heterogeneous Computing Systems". In: *arXiv preprint arXiv:2005.11050* (2020).

[71] S. Moulik, R. Chaudhary, and Z. Das. "HEARS: A heterogeneous energy-aware real-time scheduler". In: *Microprocessors and Microsystems* 72 (2020), p. 102939. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2019.102939. URL: http://www.sciencedirect.com/science/article/pii/S0141933119302017.

[72] A. M. Oprescu and T. Kielmann. "Bag-of-Tasks Scheduling under Budget Constraints". In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. Nov. 2010, pp. 351–359. DOI: 10.1109/CloudCom.2010.32.

[73] A. M. Oprescu, T. Kielmann, and H. Leahu. "Stochastic Tail-Phase Optimization for Bag-of-Tasks Execution in Clouds". In: *Fifth Int. Conf.s on Utility and Cloud Computing*. IEEE, Nov. 2012, pp. 204–208. DOI: 10.1109/UCC.2012.23.

[74] A.-M. Oprescu, T. Kielmann, and H. Leahu. "BUDGET ESTIMATION AND CONTROL FOR BAG-OF-TASKS SCHEDULING IN CLOUDS". In: *Parallel Processing Letters* 21.02 (2011), pp. 219–243. DOI: 10.1142/S0129626411000175. eprint: https://www.worldscientific.com/doi/pdf/10.1142/S0129626411000175. URL: https://www.worldscientific.com/doi/abs/10.1142/S0129626411000175.

[75] D. Poola, S. K. Garg, R. Buyya, Y. Yang, and K. Ramamohanarao. "Robust Scheduling of Scientific Workflows with Deadline and Budget Constraints in Clouds". In: *AINA 2014*. May 2014, pp. 858–865. DOI: 10.1109/AINA.2014.105.

[76] X. Qin and H. Jiang. "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems". In: *Parallel Computing* 32.5-6 (2006), pp. 331–356.

[77] W. Qiu, Z. Zheng, X. Wang, and X. Yang. "An efficient fault-tolerant scheduling algorithm for periodic real-time tasks in heterogeneous platforms". In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. June 2013, pp. 1–7. DOI: 10.1109/ISORC.2013.6913213.

[78] V. Rosolen, O. Bonaventure, and G. Leduc. "A RED discard strategy for ATM networks and its performance evaluation with TCP/IP traffic". In: *ACM SIGCOMM Computer Communication Review* 29.3 (1999), pp. 23–43.

[79] M. Safari and R. Khorsand. "Energy-aware scheduling algorithm for time-constrained workflow tasks in DVFS-enabled cloud environment". In: *Simulation Modelling Practice and Theory* 87 (2018), pp. 311–326. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2018.07.006. URL: http://www.sciencedirect.com/science/article/pii/S1569190X18300984.

[80] M. A. Salehi, J. Smith, A. A. Maciejewski, H. J. Siegel, E. K. Chong, J. Apodaca, L. D. Briceno, T. Renner, V. Shestak, J. Ladd, et al. "Stochastic-based robust dynamic resource allocation for independent tasks in a heterogeneous computing system". In: *Journal of Parallel and Distributed Computing* 97 (2016), pp. 96–111.

[81] G. Scanniello. "Source code survival with the Kaplan Meier". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 524–527.

[82] S. Z. Sheikh and M. A. Pasha. "Energy-Efficient Multicore Scheduling for Hard Real-Time Systems: A Survey". In: *ACM Trans. Embed. Comput. Syst.* 17.6 (Dec. 2018), 94:1–94:26. ISSN: 1539-9087. DOI: 10.1145/3291387. URL: http://doi.acm.org/10.1145/3291387.

[83] P. Singh, B. Khan, A. Vidyarthi, H. Haes Alhelou, and P. Siano. "Energy-Aware Online Non-Clairvoyant Scheduling Using Speed Scaling with Arbitrary Power Function". In: *Applied Sciences* 9.7 (2019). ISSN: 2076-3417. DOI: 10.3390/app9071467. URL: https://www.mdpi.com/2076-3417/9/7/1467.

[84] S. Singh and I. Chana. "Cloud resource provisioning: survey, status and future research directions". In: *Knowledge and Information Systems* 49.3 (Dec. 2016), pp. 1005–1069. ISSN: 0219-3116. DOI: 10.1007/s10115-016-0922-3. URL: https://doi.org/10.1007/s10115-016-0922-3.

[85] J. Sosnowski. "Transient fault tolerance in digital systems". In: *IEEE Micro* 14.1 (1994), pp. 24–35. DOI: 10.1109/40.259897.

[86] R. Sridharan and R. Mahapatra. "Reliability Aware Power Management for Dual-processor Real-time Embedded Systems". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 819–824. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837480. URL: http://doi.acm.org/10.1145/1837274.1837480.

[87] L. Thai, B. Varghese, and A. Barker. "A survey and taxonomy of resource optimisation for executing bag-of-task applications on public clouds". In: *Future Generation Computer Systems* 82 (2018), pp. 1–11. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.11.038. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17305071.

[88] F. Tian and K. Chen. "Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds". In: *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, July 2011, pp. 155–162. DOI: 10.1109/CLOUD.2011.14.

[89] T. Tsai, Y. Chen, X. He, and C. Li. "STEM: A Thermal-Constrained Real-Time Scheduling for 3D Heterogeneous-ISA Multicore Processors". In: *IEEE Transactions on Computers* 67.6 (June 2018), pp. 874–889. ISSN: 2326-3814. DOI: 10.1109/TC.2017.2783941.

[90] V. der Vaart. *Asymptotic Statistics*. Cambridge University Press, 1998.

[91] E. B. Valentin. "Scheduling hard real-time tasks in heterogeneous multiprocessor platforms subject to energy and temperature constraints". PhD thesis. Universidade Federal do Amazonas, 2017.

[92] C. Vecchiola, R. N. Calheiros, D. Karunamoorthy, and R. Buyya. "Deadline-driven provisioning of resources for scientific applications in hybrid clouds with Aneka". In: *Future Generation Computer Systems* 28.1 (2012), pp. 58–65. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2011.05.008. URL: http://www.sciencedirect.com/science/article/pii/S0167739X11000896.

[93] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. "Techniques to reduce the soft error rate of a high-performance microprocessor". In: *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 2004, pp. 264–275. DOI: 10.1109/ISCA.2004.1310780.

[94] C. Q. Wu, X. Lin, D. Yu, W. Xu, and L. Li. "End-to-End Delay Minimization for Scientific Workflows in Clouds under Budget Constraint". In: *IEEE Transactions on Cloud Computing* 3.2 (Apr. 2015), pp. 169–181. ISSN: 2168-7161. DOI: 10.1109/TCC.2014.2358220.

[95] X. Xiao, G. Xie, C. Xu, C. Fan, R. Li, and K. Li. "Maximizing reliability of energy constrained parallel applications on heterogeneous distributed systems". In: *Journal of Computational Science* 26 (2018), pp. 344–353. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2017.05.002. URL: http://www.sciencedirect.com/science/article/pii/S1877750317304933.

[96] G. Xie, G. Zeng, X. Xiao, R. Li, and K. Li. "Energy-Efficient Scheduling Algorithms for Real-Time Parallel Applications on Heterogeneous Distributed Embedded Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.12 (Dec. 2017), pp. 3426–3442. ISSN: 2161-9883. DOI: 10.1109/TPDS.2017.2730876.

[97] G. Xie, Y. Chen, X. Xiao, C. Xu, R. Li, and K. Li. "Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems". In: *IEEE Transactions on Sustainable Computing* 3.3 (2018), pp. 167–181.

[98] G. Xie, G. Zeng, R. Li, and K. Li. *Scheduling Parallel Applications on Heterogeneous Distributed Systems*. Springer Singapore, 2019.

[99] J. Xie and C. Liu. "Adjusted Kaplan–Meier estimator and log-rank test with inverse probability of treatment weighting for survival data". In: *Statistics in Medicine* 24.20 (2005), pp. 3089–3110. DOI: 10.1002/sim.2174. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sim.2174. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.2174.

[100] H. Xu, R. Li, C. Pan, and K. Li. "Minimizing energy consumption with reliability goal on heterogeneous embedded systems". In: *Journal of Parallel and Distributed Computing* 127 (2019), pp. 44–57.

[101] C. Yang, J. Chen, T. Kuo, and L. Thiele. "An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems". In: *2009 Design, Automation Test in Europe Conference Exhibition*. Apr. 2009, pp. 694–699. DOI: 10.1109/DATE.2009.5090754.

[102] H.-E. Zahaf, A. E. H. Benyamina, R. Olejnik, and G. Lipari. "Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms". In: *Journal of Systems Architecture* 74 (2017), pp. 46–60. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.01.002. URL: http://www.sciencedirect.com/science/article/pii/S138376211730019X.

[103] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, M. Bertogna, and G. Lipari. "A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters". working paper or preprint. Jan. 2019. URL: https://hal.archives-ouvertes.fr/hal-01971594.

[104] X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu. "Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds". In: *IEEE Trans.Parallel and Distributed Systems* 27.12 (2016), pp. 3501–3517.

List of publications

Articles in International Refereed Conferences

[C1] Y. Gao, L.-C. Canon, Y. Robert, and F. Vivien. "Scheduling independent stochastic tasks on heterogeneous cloud platforms". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891048.

[C2] L. Han, Y. Gao, J. Liu, Y. Robert, and F. Vivien. "Energy-Aware Strategies for Reliability-Oriented Real-Time Task Allocation on Heterogeneous Platforms". In: *49th International Conference on Parallel Processing - ICPP*. ICPP '20. Edmonton, AB, Canada: Association for Computing Machinery, 2020. ISBN: 9781450388160. DOI: 10.1145/3404397.3404419. URL: https://doi.org/10.1145/3404397.3404419.

Research Reports

[R1] Y. Gao, L.-C. Canon, Y. Robert, and F. Vivien. *Scheduling independent stochastic tasks on heterogeneous cloud platforms*. Research Report RR-9275. Inria - Research Centre Grenoble – Rhône-Alpes, May 2019. URL: https://hal.inria.fr/hal-02141253.

[R2] Y. Gao, L. Han, J. Liu, Y. Robert, and F. Vivien. *Minimizing energy consumption for real-time tasks on heterogeneous platforms under deadline and reliability constraints*. Research Report RR-9403. Inria - Research Centre Grenoble – Rhône-Alpes, Apr. 2021, p. 417. URL: https://hal.inria.fr/hal-03202996.

[R3] Y. Gao, Y. Robert, and F. Vivien. *Resource-Constrained Scheduling of Stochastic Tasks With Unknown Probability Distribution*. Research Report RR-9373. Inria - Research Centre Grenoble – Rhône-Alpes, Nov. 2020. URL: https://hal.inria.fr/hal-02989801.

[R4] L. Han, Y. Gao, J. Liu, Y. Robert, and F. Vivien. *Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms*. Research Report RR-9324. Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, Mar. 2020. URL: https://hal.inria.fr/hal-02500381.

[1]

[1] [R3] and [R2] in submission to a journal.