Scheduling sequential or parallel hard real-time pre-emptive tasks upon identical multiprocessor platforms

Pierre Courbin

ÉCOLE DOCTORALE UNIVERSITÉ PARIS-EST

Mathématiques et STIC

THÈSE

en vue de l'obtention du titre de

Docteur
de l'Université Paris-Est

Spécialité : Informatique

Pierre Courbin

Scheduling Sequential or Parallel Hard Real-Time Pre-emptive Tasks upon Identical Multiprocessor platforms

---

soutenue le vendredi 13 Décembre 2013

Jury

Directeur :	**Laurent George**	–	Université Paris-Est (LIGM), France
Rapporteurs :	**Alan Burns**	–	University of York (ARTIST), United Kingdom
	Pascal Richard	–	Université de Poitiers, France
Examinateurs :	**Joël Goossens**	–	Université Libre de Bruxelles (PARTS), Belgium
	Yves Sorel	–	INRIA Rocquencourt (AOSTE), France

PhD prepared at
ECE Paris – LACSC
Laboratoire d'Analyse et Contrôle des Systèmes Complexe
37, Quai de Grenelle
CS 71520
75725 PARIS CEDEX 15

PhD in collaboration with
UPEC – LiSSi (EA 3956)
Laboratoire Images, Signaux et Systèmes Intelligents
Domaine Chérioux
122 rue Paul Armangot
94400 Vitry sur Seine

À mes parents,
À mes grands parents, encore là ou déjà partis,
À mes nombreux frères et sœurs,
À toute ma famille,
À tous mes amis,
À toutes les personnes que j'ai eu le privilège de croiser.

Parce que chaque personne rencontrée, même brièvement,
est une occasion exceptionnelle de se re-découvrir
et de s'émerveiller.

To my parents,
To my grandparents, still there or already gone,
To my many brothers and sisters,
To all my family,
To all my friends,
To all the people who have graced my life.

Because each encounter, even a brief one,
is an occasion to re-imagine oneself
and to marvel.

Fortunately,
I do not suffer from
friggatriskaidekaphobia

Acknowledgments

Et je n'ai point d'espoir de sortir par moi de ma solitude.
La pierre n'a point d'espoir d'être autre chose que pierre.
Mais, de collaborer, elle s'assemble
et devient Temple.

I have no hope of getting out of my solitude by myself.
Stones have no hope of being anything but stones.
However, through collaboration they get themselves together
and become a Temple.

Antoine de Saint-Exupéry [SE48]

On m'avait dit que c'était la dernière chose à faire. Je l'écris donc en dernier, moins d'un mois avant la soutenance publique.
Et je concède que le conseil était exact : il faut remercier au dernier moment.

Ne serait-ce que pour n'oublier personne.

Il n'y a pas de hiérarchie dans mes remerciements. Les personnes qui m'ont entouré régulièrement, celles qui m'ont formé humainement et techniquement (si différence il y a) et que je n'ai pas toujours revu, même celles que j'ai croisé chaque jour et qui n'ont fait que me sourire, chacune a contribué à mon avancement et mérite certainement des remerciements.

Mais ce serait bien trop simple de généraliser ainsi et de remercier tout le monde. Faudra-t-il que je montre du doigt ? Que je lance des noms[1] ? Que je liste les méfaits de chacun ?

Je le fais donc sans rechigner. Saurais-je dire, avec plaisir.

Commençons.

Mais, non, attendez. Il me faut être organisé[2]. Je peux au moins, délibérément, commencer par les personnes qui ont œuvré dans ma vie professionnelle. Je pourrai ensuite évoquer ceux qui m'ont supporté[3] dans ma vie personnelle. Même si, finalement, la frontière n'est pas toujours si évidente quand on a le loisir et la chance de croiser les gens dont je vais vous parler.

[1] Si vous vous attendiez à voir votre nom et qu'il est introuvable, c'est peut être simplement que vous êtes si précieux que je n'ai pas voulu étaler ici votre importance. Ou alors c'est ma mémoire de poisson rouge. Dans tous les cas, j'espère que vous ne m'en tiendrez pas rigueur. Et peut être que le verre que je m'engage ici à vous offrir contribuera à mon absolution.

[2] Qui a dit psychorigide ? Passons.

[3] Dans tous les sens du terme.

Le travail, c'est la santé ?

Même si le terme de "travail" me semble être de plus en plus désuet tant j'espère que nous saurons un jour nous en affranchir, je fais ici référence tout autant aux activités de recherche et d'enseignement que j'ai pu mener, qu'à tout l'environnement, à l'ECE Paris, qui a contribué à modeler ce travail de thèse.

Jury Je me dois, par respect, convention mais aussi une profonde gratitude, de remercier en tout premier lieu les membres de mon jury.

Alan Burns qui, malgré un emploi du temps chargé et une réputation qui n'est plus à présenter, a accepté de relire et de rapporter mon travail. Ses retours clairs, précis et constructifs ont amené des reconsidération intéressantes et des corrections essentielles.

Pascal Richard a réalisé une relecture attentive et m'a apporté des perspectives et des ouvertures qui m'apparaissent aujourd'hui très prometteuses. Il me semble avoir relevé des points importants où mon travail aurait pu être approfondi, tout en pointant ses aspects positifs.

Yves Sorel m'a fait le plaisir d'accepter d'être examinateur de ma thèse. Sa présence prochaine à ma soutenances et ses conseils avisés sur la gestion du stress ainsi que ses mots d'encouragements et de confiance me touchent et contribuent à faire de cette fin de thèse un moment agréable.

Joël Goossens a aussi accepté de participer à mon jury de soutenance. Pour être honnête, j'aurai eu beaucoup de difficulté à envisager une soutenance sans sa présence. J'ai eu la chance et le plaisir de travailler en étroite collaboration avec lui, notamment durant un mois complet où il a accepté de me recevoir dans sa belle ville de Bruxelles. Si je n'oublierai jamais l'exemple de rigueur, tant scientifique que personnelle, et de professionnalisme qu'il m'a montré, je garderai aussi un souvenir clair des échanges plus culturels que nous avons pu avoir. C'est ici les remerciements, je ne vais donc pas m'étendre plus sur son fort apport scientifique à ma formation ; Vous le verrez dans le reste de ce manuscrit. Cependant, je veux partager avec vous un point important et, connaissant son intérêt pour les définitions claires, je vais le formuler ainsi :

Définition 1 (Chocolat)**.**
Produit obtenu par le mélange de pâte de cacao et de sucre. Son goût est caractéristique et existe sous sa forme la plus pure en Belgique (attention aux contrefaçons). Essentiel à la bonne réalisation des recherches scientifiques. ■

Propriété 1 (Être du chocolat)**.**
Notons $\mathcal{C}$ l'ensemble des éléments suivant la Définition 1. On a donc :

$$\textit{Galler} \in \mathcal{C}$$
$$\textit{Léonidas} \notin \mathcal{C}$$

■

Laurent George a été durant ces quatre années mon directeur de thèse. Je peux le dire ici, j'ai eu la chance d'être encadré par une personne qui connaît très bien son domaine et les différents acteurs et qui m'a fait confiance en m'intégrant à de nombreux travaux, me permettant ainsi de m'approprier rapidement des notions et un domaine complexe. Même si, comme dans toutes relations de quatre ans, il y a parfois des divergences, je garde un excellent souvenir de nos échanges au tableau blanc et j'espère sincèrement qu'ils seront encore nombreux. La confiance qu'il m'a témoigné tout au long de ce parcours et la rigueur qu'il m'a montré, notamment durant ma rédaction, forment des bases solides qui m'ont permis d'arriver où j'en suis aujourd'hui.

ECE Paris J'ajoute ici quelques mots pour les employés de l'ECE Paris. Même si les choses ont évolué, sont parfois floues et incertaines, l'environnement de l'ECE Paris a gardé un parfum de famille et d'entraide. Quelque soit le service, de celui des admissions où Laurence Léonard et toute l'équipe fait un travail impressionnant, au service informatique où notamment Franck Tesseyre et Olivier Roux sont toujours là pour répondre à nos attentes souvent spécifiques, en passant par le service des moyens généraux où Philippe Allard et Sri cherchent sans cesse à nous faciliter les choses, avec Anne-Paul Dauphin qui met chaque fois à l'accueil une bonne humeur et permet de débuter les journées avec le sourire, et sans oublier Philippe Pinto qui a été d'une très grande aide pour de nombreuses choses mais notamment pour l'impression spécifique de ce travail. Je devrais citer l'ensemble de mes collègues enseignants et de l'équipe pédagogique, mais j'aurai peur d'en oublier en étant trop spécifique. Pourtant, qu'ils sachent que, que ce soit en participant à ma formation ou en étant là régulièrement pour me faire découvrir de nouvelles choses, je leur en suis sincèrement reconnaissant. J'utilise tout de même quelques mots pour remercier deux personnes. Stefan Ataman qui a été mon premier collègue, m'a accompagné à mes premiers cours et est pour moi un exemple silencieux de rigueur, de perfectionnisme et de dévouement pour ses enseignements et ses élèves. Max Agueh, avec qui j'apprécie particulièrement de chahuter et quoi qu'il puisse en penser, est et reste pour moi un modèle d'écoute, d'attention et de respect. La sérénité avec laquelle il aborde les choses est une source d'inspiration considérable.

Je vais tout de même m'étendre un peu sur le cas de cinq personnes en particulier. Pascal Brouaye et Nelly Rouyres ont été des modèles en dirigeant l'école durant mes études et une grande partie de ma thèse. Florence Wastin, rencontrée en intégrant le corps enseignant de l'école, a ensuite complété ce tableau de trio qui m'a montré une chose essentielle : même si personne n'est parfait et que tout est discutable, j'ai eu grâce à eux le sentiment que l'image utopiste que je me faisais d'une entreprise n'est pas totalement impossible. En étant proche, à l'écoute, ils m'ont montré qu'il était possible de faire attention au bien être de chacun tout en créant un espace de travail dynamique et performant.

Y-aurait-il un lien étroit entre les deux ?

J'ajoute aujourd'hui les noms de Lamia Rouai et Christophe Baujault qui leur ont succédé et à qui je souhaite beaucoup de réussite. Un merci à Lamia pour avoir aussi été mon enseignante passionnée par son domaine et à Christophe pour m'avoir encadré durant mes dernières années en tant qu'étudiant, m'avoir permis de faire un parcours atypique la dernière année, m'avoir rappelé ensuite pour que je réalise ce travail de thèse et, finalement, pour m'avoir accompagné en me donnant des conditions de travail exceptionnelles.

Choisit-on vraiment ses amis ?

Il y a des personnes qui ne se retrouvent pas dans la partie précédente. Et ils s'en sont certainement étonné. C'est parce que plusieurs, plus que des collègues, ont pris au fil des années une place importante dans ma vie. Je veux tout d'abord évoquer Ikbal Benakila. Même si je ne l'ai pas revu depuis plusieurs années, il restera une personne importante qui m'a accueilli, m'a accompagné et m'a conseillé durant mes premiers temps en thèse. Il a aussi été un modèle et m'a ravi par des échanges culturels précieux. Ensuite, Rafik Zitouni, esprit avisé et discret, a repris ce flambeau de modèle et d'ouverture culturelle. Nos discussions et nos débats de société resteront des souvenirs impérissables que je chéris. Philippe Thierry et Ermis Papastefanakis ont contribué et contribuent toujours au plaisir d'échanger sur des sujets tant techniques que personnels. Frédéric Fauberteau et Olivier Cros, rencontrés plus récemment, m'ont déjà amené à me poser de nombreuses nouvelles questions et, en si peu de temps, nos échanges variés présagent déjà de très belles perspectives. Clément Duhart et Thomas Guillemot sont aussi pour moi des découvertes incomparables. Chacun avec son esprit si particulier et vif, chacun avec ses désirs si présents et refoulés, chacun avec sa passion pour toutes les formes de sciences et d'arts, chacun est ainsi, à sa façon, un puits de sagesse. Vincent Sciandra a été une rencontre aussi très singulière et génératrice d'épanouissement. Nos compétences complémentaires et nos intérêts convergents pour les sciences, la technique et les réflexions sur les sociétés humaines font de la rencontre de ce "bobo[4]" un des évènements les plus important de ces dernières années.

Je prends un peu de place ici pour parler de ceux qui m'ont incité, parfois contre leur gré, à poursuivre en doctorat. Je veux parler de ceux que j'ai rencontré lors de mon Master et que j'ai toujours plaisir à revoir : Adrien Bak, Antoine Pédron, Joël Falcou et bien évidemment Tarik Saidani. Malgré les déboires de thèse que chacun a connu, ils m'ont montré, peut être inconsciemment, tout l'intérêt en matières de rencontres techniques et humaines que peut amener la recherche scientifique.

De ces rencontres, je retiens les personnes qui ont réalisé leur stage avec moi, ont contribué à ce travail et m'ont fait découvrir des mondes et des personnalités

[4]Private joke...

singulières, citons notamment Meryem Sahlali, Bruno Cats, Hervé Launay, Benjamin Bado, Sara Morillon et Adrien Leroy.

Mais je n'oublie pas mes amis plus anciens qui, par leurs blagues sur les enseignants et leur compréhension fine de mon travail sur les "truc avec plein de petites pattes", ont été un havre de paix pour se ressourcer et s'évader. Ils se reconnaitrons et savent à quel point je tiens à eux. Je me permets ici de citer quelques noms de ceux qui ont été particulièrement présents dans des moments de doutes importants et qui ont été d'un soutien inestimable. Dans un désordre total, je veux parler de Marie Muret, Nicolas Morin, Laura Bernais, Laëtitia Bernais, Irina Lupu et Aude Gueudry ainsi que sa famille Diane Gueudry, Corentin Roussel et particulièrement ses parents, Isabelle et Claude Gueudry, sans oublier la famille Dutay, Maud et Isabelle (et ma chère Gwenaëlle !). Vous avez fait tant pour moi dans des moments si difficiles que je ne saurai comment vous remercier vraiment. Et même si je vais maintenant parler de ma famille au sens strict, vous savez quelle est votre place pour moi.

"On choisit pas sa famille. On choisit pas non plus les trottoirs de Manille, de Paris ou d'Alger pour apprendre à marcher." [For87]

Ainsi donc, ma famille. Je ne l'ai certainement pas choisi mais elle est à l'image des personnes que j'ai rencontré par la suite : grande, diverse et donc enrichissante. Que ce soit mes nombreux frères et sœurs, Elodie, Paul, Jean-Baptiste, Nathalie, Julien, Cécile et Valérie, ou même leurs conjoints et enfants, tout comme mes seconds parents, Liliane et Daniel et mes grands parents Renée, Guy, Michelle, Yves et Éliane, ils ont été des exemples si divers qu'ils m'ont permis de me construire en gravant en moi cette chose essentielle : chacun est unique, chacun est nécessaire et chacun contribue à enrichir les autres par le simple fait d'exister.

Pour finir, mes parents. Je ne vais dire que quelques mots, une seconde thèse serait nécessaire pour parler vraiment d'eux. Ma mère Véronique, solide, attentionnée et délicate, ainsi que mon père René, aimant, entier et brillant, sont à eux deux des exemples indescriptibles qui ont forgé ma personnalité. Si je ne dois citer qu'une chose liée à mon travail de recherche, c'est leur caractéristique commune et primordiale qu'est l'ouverture d'esprit. Ils m'ont montré des voies différentes et complémentaires pour cheminer vers un fondamental : ne pas juger. Le jugement réduit les choses et les personnes à un état dans lequel ils ne sont déjà plus et limite ainsi les possibilités de découvertes et d'évolutions.

Voir les exemples débordant d'amour et de bienveillance juste que vous avez toujours été est pour moi un rappel de la chance que j'ai d'être, par fortune, votre enfant.

Bref, merci.

On dit parfois que les gens se définissent par la somme de leurs expériences. Je pense que c'est incomplet, sinon incorrect. Les gens se définissent par la somme de leurs rencontres et par leur capacité à extraire des apprentissages de ces coïncidences. Nous nous découvrons et nous révélons au contact des autres. C'est pourquoi ce travail et cette exploration personnelle de quatre années peuvent tout à fait être dédiés à toutes les personnes que j'ai déjà cité, et à toutes celles qui restent dans l'ombre.

Être homme, c'est précisément être responsable.
C'est connaître la honte face à une misère qui ne semblait pas dépendre de soi.
C'est être fier d'une victoire que les camarades ont remportée.
C'est sentir, en posant sa pierre,
que l'on contribue à bâtir le monde.

To be a man is, precisely, to be responsible.
It is to feel shame at the sight of what seems to be unmerited misery.
It is to take pride in a victory won by our comrades.
It is to feel, when setting our stone,
that we are contributing to the building of the world.

Antoine de Saint-Exupéry [SE39]

Abstract

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to", said the Cat.
"I don't much care where", said Alice.
"Then it doesn't matter which way you go", said the Cat.

Lewis Caroll, Alice's Adventures in Wonderland

In this work, we are interested in the problem of scheduling independent tasks on a hard **R**eal-**T**ime (RT) system composed of multiple processors. A RT system is a system that has time constraints (or timeliness constraints) such that the correctness of these systems depends on the correctness of results it provides, but also on the time instant the results are available. In order to constrain the availability of results, we generally use the concept of "deadline". In a "hard" RT system, the respect of temporal constraints is essential since a missed deadline may cause catastrophic consequences. For example, in the management of train traffic, if a train must use a railroad switch it is important to properly position it before the train arrives or a collision may occur. Thus, the problem of scheduling tasks on a hard RT system consists in finding a way to choose, at each time instant, which tasks should be executed on the processors so that each task succeeds to complete its work before its deadline. We are interested in the scheduling of **S**equential **Tasks** (S-Tasks) (tasks use one processor at a time) and **P**arallel **Tasks** (P-Tasks) (tasks may use multiple processors at a time) in hard RT systems composed of identical multiprocessor platforms (the processors in the platform are strictly identical). In the literature of the state-of-the-art, there are various approaches to schedule these systems.

Regarding S-Tasks scheduling, results have been proposed using the so-called **P**artitioned **Scheduling** (P-Scheduling) approach which has the advantage of reducing the problem containing multiple processors to multiple problems containing a single processor. This approach has received much attention, but it poses a problem: it can give poor results for task sets with a high utilization of the processors. For example, it can be shown that in some pathological task configurations, we can only ensure the schedulability of a system which uses less than 50% of the processors capacities. Notice that we compute the task utilization of processor capacity according to the execution time required by the task and its recurrence: if a task needs 2 milliseconds on a processor to complete its execution and it has to be executed again each 4 milliseconds, then this task requires $2/4 \times 100 = 50\%$ of the processor capacity. As a consequence of this poor results, the **G**lobal **Scheduling** (G-Scheduling) approach has been proposed and allows,

in theory, to fully use the processors capacities. However, this approach poses another problem: it induces many migrations of tasks between processors which can lead to additional costs that are still poorly mastered in the state-of-the-art of RT scheduling. Thus, a hybrid solution has been proposed, the **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approach, which aims at minimizing the number of tasks that can migrate between processors.

Regarding P-Tasks scheduling, recent research are very diverse because, in addition to several approaches, there are also several models to represent P-Tasks. The Gang model considers that there are many communications between concurrent threads of a P-Task and therefore requires scheduling them simultaneously. In contrast, the Multi-Thread model assumes that threads are independent. The synchronization between threads is generally defined by successive phases. Each phase is activated when all threads of the previous phase have been completed. This is particularly the case of the Fork-Join model.

In this thesis, we first study S-Tasks scheduling problem. For the P-Scheduling approach, we study different partitioning algorithms proposed in the literature of the state-of-the-art in order to elaborate a generic partitioning algorithm. Especially, we investigate four main placement heuristics (First-Fit, Best-Fit, Next-Fit and Worst-Fit), eight criteria for sorting tasks and seven schedulability tests for **E**arliest **D**eadline **F**irst (EDF), **D**eadline **M**onotonic (DM) and **R**ate **M**onotonic (RM) schedulers. It is equivalent to 224 potential P-Scheduling algorithms. Then, we analyse each of the parameters of this algorithm to extract the best choices according to various objectives. Afterwards, we study the SP-Scheduling approach for which we propose a solution for each of the two sub-categories: with **Rest**ricted **Migrations** (Rest-Migrations) where migrations are only allowed between two successive activations of the task (in other words, between two jobs of the task, thus only *task migration* is allowed), and with **UnRest**ricted **Migrations** (UnRest-Migrations) where migrations are not restricted to job boundaries (*job migration* is allowed). We provide schedulability tests and an evaluation for EDF scheduler in order to find advantages and disadvantages of each sub-category. In particular, we observe that the approach with UnRest-Migration gives the best results in terms of number of task sets successfully scheduled. However, we observe a limit on the ability of this approach to split tasks between many processors: if the execution time of the task is too small compared to the time granularity of processor execution, it will be impossible to split the execution time. Thus, the Rest-Migration approach is still interesting, especially as its implementation seems to be easier to achieve on real systems.

Regarding P-Tasks scheduling problem, we propose the **M**ulti-**P**hase **M**ulti-**T**hread (MPMT) task model which is a new model for Multi-Thread tasks to facilitate scheduling and analysis. We also provide schedulability tests and

a method for transcribing Fork-Join tasks to our new task model. An exact computation of the **W**orst **C**ase **R**esponse **T**ime (WCRT) of a periodic MPMT task is given as well as a WCRT bound for the sporadic case. Finally, we propose an evaluation to compare Gang and Multi-Thread approaches in order to analyse the advantages and disadvantages. In particular, even if we show that both approaches are incomparable (there are task sets which are schedulable using Gang approach and not by using Multi-Thread approach, and conversely), the Multi-Thread model allows us to schedule a larger number of task sets and it reduces the WCRT of tasks. Thus, if the tasks are not too complex and do not require too much communication between concurrent threads, it seems interesting to model them with a Multi-Thread approach.

Finally, we have developed a framework called **F**ramework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation (FORTAS) to facilitate evaluations and tests of multiprocessor scheduling algorithms. Its particularity is to provide a programming library to accelerate the development and testing of RT scheduling algorithms. This framework will be proposed as an open source library for the research community.

Résumé

"Voulez-vous me dire, s'il vous plaît, quel chemin je dois prendre à partir d'ici ?"
"Cela dépend grandement de où vous voulez aller", dit le Chat.
"Peu m'importe où", dit Alice.
"Alors le chemin que vous prenez n'a pas d'importance.", dit le Chat.

Lewis Caroll, Alice au Pays des Merveilles

Dans ce travail, nous nous intéressons au problème d'ordonnancement de tâches indépendantes sur des systèmes **T**emps-**R**éel (TR) durs composés de plusieurs processeurs. Les systèmes TR sont des systèmes qui ont des contraintes temporelles (ou contraintes de ponctualité associées aux tâches exécutées) qui font que la conformité de ces systèmes repose sur l'exactitude des résultats qu'ils fournissent mais aussi sur le moment où ces résultats sont disponibles. Pour contraindre la date de disponibilité des résultats, on utilise généralement le concept "d'échéance". Dans un système TR "dur", le respect des contraintes de ponctualité est essentiel car une échéance manquée peut entraîner des conséquences catastrophiques. Par exemple, dans le domaine de la gestion de trafic ferroviaire, si un train doit passer par un aiguillage, il est primordial de bien le positionner avant que le train n'arrive ou une collision pourrait se produire. Ainsi, le problème d'ordonnancer des tâches sur un système TR dur consiste à trouver une façon de choisir, à chaque instant, quelles tâches doivent s'exécuter sur les processeurs de façon à ce qu'elles puissent toutes s'exécuter complètement avant leur échéance. Nous nous intéressons ici à l'ordonnancement de **Tâches S**équentielles (S-Tasks) (les tâches utilisent un seul processeur à la fois) et de **Tâches P**arallèles (P-Tasks) (les tâches peuvent utiliser plusieurs processeurs à la fois) sur des systèmes TR durs composés de plate-formes multiprocesseurs identiques (tous les processeurs de la plate-forme sont strictement identiques). Dans la littérature, plusieurs approches permettant d'ordonnancer ces systèmes ont été proposées.

Concernant les S-Tasks, des résultats ont été proposés en utilisant l'approche dite par **Ordonnancement P**artitionné (P-Scheduling) qui a l'avantage de réduire le problème composé de plusieurs processeurs à plusieurs problèmes composés chacun d'un seul processeur. Cette approche a été largement étudiée mais elle pose un problème : elle donne des résultats médiocres pour des jeux de tâches nécessitant une forte utilisation des processeurs. Par exemple, on peut montrer que dans des configurations pathologiques de jeux de tâches, il n'est pas possible de garantir l'ordonnançabilité de jeux qui utilisent plus de 50% de la capacité des processeurs. Notez que la capacité d'un processeur utilisée par une tâche est calculée en fonction du temps d'exécution de la tâche et de sa récurrence : si une tâche a besoin de 2 millisecondes sur un processeur pour s'exécuter complètement

et qu'elle doit être exécutée à nouveau toutes les 4 millisecondes, alors cette tâche a besoin de $2/4 \times 100 = 50\%$ de la capacité d'un processeur. En conséquence de ces mauvais résultats, une autre approche nommée **Ordonnancement Global** (G-Scheduling) a vu le jour et permet, théoriquement, d'utiliser totalement la capacité de la plate-forme. Cependant, celle-ci pose un autre problème : elle induit de nombreuses migrations des tâches entre les processeurs, ce qui peut produire des coûts supplémentaires qui sont encore mal maîtrisés dans l'état de l'art de l'ordonnancement TR. Finalement, une solution hybride a été proposée, l'approche par **Ordonnancement Semi-Partitionné** (SP-Scheduling), qui cherche à minimiser le nombre de tâches pouvant migrer entre les processeurs.

Concernant les P-Tasks, les recherches récentes sont très variées car, en plus de plusieurs approches d'ordonnancement, il y a aussi divers modèles pour représenter ces P-Tasks. Le modèle Gang considère par exemple que les fils d'exécution concurrents (threads) d'une P-Task doivent souvent communiquer entre eux et qu'il est donc préférable de les ordonnancer ensemble. A l'inverse, le modèle Multi-Thread considère que les threads sont totalement indépendants. Les synchronisations entre les threads sont généralement représentées par des phases successives dans les P-Tasks. Chaque phase est activée uniquement quand tous les threads de la phase précédente ont terminé, ce qui correspond à une barrière en programmation parallèle. Fork-Join est un exemple d'un modèle Multi-Thread.

Dans cette thèse nous étudions tout d'abord le problème d'ordonnancement des S-Tasks. Pour l'approche P-Scheduling, nous étudions différents algorithmes proposés dans la littérature afin de pouvoir proposer un algorithme générique. Nous examinons notamment les quatre principales heuristiques de placement (First-Fit, Best-Fit, Next-Fit et Worst-Fit), huit critères de tri de tâches et sept tests d'ordonnançabilité pour les ordonnanceurs **E**arliest **D**eadline **F**irst (EDF), **D**eadline **M**onotonic (DM) et **R**ate **M**onotonic (RM). Ceci nous permet de tester l'équivalent de 224 algorithmes potentiels de P-Scheduling. Nous analysons ensuite chaque paramètre de cet algorithme pour en extraire les meilleurs choix à faire en fonction de divers objectifs. Puis nous étudions l'approche SP-Scheduling pour laquelle nous proposons une solution pour chacune de deux sous-catégories : avec des **Migrations Restreintes** (Rest-Migrations) où les migrations sont autorisées mais uniquement entre deux activations de la tâche (en d'autres termes, entre deux jobs de la tâche, donc seulement la *migration de tâche* est autorisées) et avec des **Migrations Non-Restreintes** (UnRest-Migrations) où les migrations ne sont pas limitées aux frontières des jobs (la *migration de job* est autorisée). Nous donnons un test d'ordonnançabilité et une évaluation pour l'ordonnanceur EDF afin de trouver les avantages et les inconvénients de chaque sous-catégorie. En particulier, nous observons que l'approche UnRest-Migrations donne de meilleurs résultats en matière de nombre de jeux de tâches ordonnancés avec succès. Néanmoins, nous observons que cette approche peut parfois être limitée quand elle cherche à

découper des tâches : si le temps d'exécution de la tâche est trop faible comparé à la granularité d'exécution du processeur, ce temps ne pourra pas être découpé. Ainsi, l'approche Rest-Migrations peut s'avérer intéressante, notamment parce que son implémentation sur des systèmes réels semble plus facilement envisageable.

Concernant l'ordonnancement des P-Tasks, nous proposons un nouveau modèle de tâches nommé **M**ulti-**P**hase **M**ulti-**T**hread (MPMT). Il permet notamment de faciliter l'ordonnancement et l'analyse des tâches Multi-Thread. Nous proposons aussi un test d'ordonnançabilité et une méthode pour traduire une tâche Fork-Join vers notre nouveau modèle de tâche. Un calcul exact du **P**ire **T**emps de **R**éponse (WCRT) d'une tâche MPMT périodique est aussi donné ainsi qu'une borne pour le calcul du WCRT d'une tâche MPMT sporadique. Enfin, nous menons une évaluation pour comparer les modèles Gang et Multi-Thread afin d'en extraire les avantages et les inconvénients respectifs. En particulier, même si nous montrons que les deux modèles sont incomparables (dans le sens où des jeux de tâches sont ordonnançable avec un modèle Gang mais pas avec un modèle Multi-Thread et inversement), le modèle Multi-Thread permet d'ordonnancer un plus grand nombre de jeux de tâches et il réduit aussi le WCRT des tâches. Ainsi, si les tâches ne sont pas excessivement complexes et qu'elles ne nécessitent pas beaucoup de communication entre leurs threads, il peut être intéressant de les modéliser avec une approche Multi-Thread.

Finalement, nous avons développé un *framework* nommé **F**ramework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation (FORTAS) pour faciliter l'évaluation et le test des algorithmes d'ordonnancement multiprocesseur. Sa particularité est de proposer une bibliothèque de programmation pour accélérer le développement et le test des théories sur les systèmes TR. Cet outil sera proposé à la communauté des chercheurs sous forme d'une bibliothèque au code source ouvert.

Author's publication list

Refereed Book Chapter Paper

IGI'2010 Laurent George and **Pierre Courbin**. "IGI Global". In: edited by Mohamed Khalgui and Hans-Michael Hanisch. IGI Global, 2011. Chapter Reconfiguration of Uniprocessor Sporadic Real-Time Systems: The Sensitivity Approach, pages 167–189. isbn: 978-1-5990-4988-5. doi: 10.4018/978-1-60960-086-0.ch007

Refereed Journal Papers

JSA'2011 Laurent George, **Pierre Courbin**, and Yves Sorel. "Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling". In: *Journal of Systems Architecture* 57.5 (May 2011), pages 518–535. issn: 1383-7621. doi: 10.1016/j.sysarc.2011.02.008

RTS'2013 **Pierre Courbin**, Irina Lupu, and Joël Goossens. "Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks". In: *Real-Time Systems* 49.2 (2013), pages 239–266. issn: 0922-6443. doi: 10.1007/s11241-012-9173-x

Refereed Conference Papers

ETFA'2010 Irina Lupu, **Pierre Courbin**, Laurent George, and Joël Goossens. "Multi-criteria evaluation of partitioning schemes for real-time systems". In: *Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*. Emerging Technologies and Factory Automation (ETFA). Bilbao, Spain: IEEE Computer Society, Sept. 2010, pages 1–8. isbn: 978-1-4244-6848-5. doi: 10.1109/ETFA.2010.564121

RTNS'2010 Robert I. Davis, Laurent George, and **Pierre Courbin**. "Quantifying the Sub-optimality of Uniprocessor Fixed Priority Non-Pre-emptive Scheduling". In: *Proceedings of the 18th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Toulouse, France, Nov. 2010, pages 1–10

RTNS'2012 Benjamin Bado, Laurent George, **Pierre Courbin**, and Joël Goossens. "A semi-partitioned approach for parallel real-time scheduling".

In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Pont à Mousson, France: ACM, Nov. 2012, pages 151–160. ISBN: 978-1-4503-1409-1. DOI: 10.1145/2392987.2393006

Refereed Workshop and WIP[5] Papers

WATERS'2011 **Pierre Courbin** and Laurent George. "FORTAS : Framework fOr Real-Time Analysis and Simulation". In: *Proceedings of 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). Porto, Portugal, July 2011

JRWRTC'2011 Vandy Berten, **Pierre Courbin**, and Joël Goossens. "Gang fixed priority scheduling of periodic moldable real-time tasks". In: *Proceedings of the Junior Researcher Workshop Session of the 19th International Conference on Real-Time and Network Systems*. Edited by Alan Burns and Laurent George. Real-Time and Network Systems (RTNS). Nantes, France, Sept. 2011, pages 9–12

RTSS-WiP'2012 Vincent Sciandra, **Pierre Courbin**, and Laurent George. "Application of mixed-criticality scheduling model to intelligent transportation systems architectures". In: *Proceedings of the WIP Session of the 33th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). San Juan, Puerto Rico: ACM, Dec. 2012, pages 22–22. DOI: 10.1145/2518148.2518160

[5] Work In Progress

Contents

List of Figures

List of Tables

List of Algorithms

Part I

General concepts and notations

Chapter 1

General introduction

Finalement, le raisonnement a priori est si satisfaisant pour moi que si les faits ne correspondent pas, mon sentiment est : tant pis pour les faits.

In fact the a priori reasoning is so entirely satisfactory to me that if the facts won't fit in, why so much the worse for the facts is my feeling.

Charles Darwin's brother Erasmus [Dar58]

Contents

1.1 Real-Time Systems

The term "**R**eal-**T**ime (RT)" is used in very different ways, particularly in France. It is often used in an unclear meaning and away from the one considered in this work. For example, the SNCF (France's national state-owned railway company) offers a mobile application that provides "real-time" travel time. However, when we are accustomed to use this application and when we know the relevance of information, we are entitled to ask what "real-time" means!

In fact, a mistake that is often made is to make a connection between "real-time" and "speed". If the speed can be useful for a RT system, it is not a defining characteristic. A RT system is only a system that has time constraints. The correctness of these systems depends on the correctness of results it provides, but also at which time instant the results are available. Generally, we referred to a "deadline" to constrain the availability of results. For example, in an augmented reality application, if a virtual information should be superimposed on a real environment it is important that it appears before the environment changes, otherwise it is no longer relevant. Similarly, in the management of train traffic, if a train must use a railroad switch it is important to properly position it before

the train arrives. Notice that here, there is no concept of speed, the railroad switch can have an hour between two train to make the change, the important thing is that the change must be made before the train arrives.

With these two examples, we can define two types of RT systems:

Soft RT refers to systems where the respect of temporal constraints is desired but not necessary. In such systems, it is acceptable to miss some deadlines without causing catastrophic consequences. This is particularly the case for our example of augmented reality: if the information does not appear at the right time and the right place, it will simply be wrong but the user will not be in danger.

Hard RT refers to systems where the respect of temporal constraints is essential. In such systems, a missed deadline may cause catastrophic consequences. This is particularly the case for our example with railroad switch: if the change is not made before the train arrives, a collision may occur.

Thus, we find RT systems in many areas, for example:

- Banking systems (stock exchange etc.),
- Aerospace,
- Processing and routing the information (video, data, etc.),
- Industry production (control engines etc.),
- Traffic Management (road, air, railway etc.),
- Military Systems.

Finally, in RT systems research, we are interested in how we schedule the tasks to be made in order to ensure that all end before their deadline, but we are also interested in how theoretically predict that we will be able to meet all deadlines.

1.2 Motivations of the thesis

RT systems have been studied extensively in the context where a single processing unit was available (a single processor or a single network channel etc.). But today, in the field of computer science, we have wide access to several processing unit in a single platform. Indeed, let's talk a little history. In 1975, Gordon E. Moore prophesies that the number of transistors in microprocessors will double every two years. So far, this law has been respected and has become a target for the world of microprocessors, even if Gordon Moore announced its end

in "10 to 15 years" (Intel Developer Forum, 2007). However, what is commonly called "MOORE's Law" is that "the performance of microprocessors doubles every 18 months". This declaration, actually from DAVID HOUSE [Moo03], has been respected from the Intel 4004 (1971) until today. Semiconductor manufacturers used mainly the increase in processor frequency to meet this law. Consequently, a program could benefit from increased performance without any effort on the part of software developers. In 2004, semiconductor manufacturers were blocked by physical limitations (miniaturization, thermal dissipation problem, etc.) which prevent a continuous increase in processor frequency. To follow the "HOUSE's Law", they then have begun to double the number of "cores" in processors. Today we are witnessing the development of processors with 2, 4, 8 up to 1000 cores.

It therefore becomes important to see how RT systems can benefit from these new multiprocessor platforms. We begin by taking the same tasks that we used with a single processor, which we call **S**equential **Tasks** (S-Tasks). We study various existing approaches to bring new solutions or new ways of solving the scheduling of S-Tasks on a multiprocessor platform.

We continue this work by exploring new types of tasks, called **P**arallel **Tasks** (P-Tasks). Indeed, these multi-core (or many-core) processors raise a major issue: programs cannot anymore benefit from increased performance for nothing as written by HERB SUTTER in "The Free Lunch is Over" [Sut05]. Actually, programmers should launch concurrent treatments (parallel treatments) to take advantage of these new architectures. Therefore a process is divided in "threads" that could run concurrently in order to reduce the total processing time. To facilitate programming on multi-core, many tools are available such as **O**pen **M**ulti-**P**rocessing (OpenMP) [Cha+00] that can turn a sequential code to a parallel one. This tool has spread rapidly because it is easy to use. It enables parallel code generation by means of library of functions as well as preprocessor directives. These directives allow changes in the sequential operation of a program without being destructive to the original code. Intel also maintains a library, **T**hreading **B**uilding **B**locks (TBB) [Rei07] which is a C++ runtime library. Among the many other possibilities, **M**essage **P**assing **I**nterface (MPI) [GB98; GLS00] has a slight different use since it allows us to distribute the computation not only on multi-core who share a memory, but also on processors with a separate memory. For example, MPI can distribute the computations on machines on a network by sending messages with the data required for calculations.

We study the scheduling of such tasks by providing a new model for representing and different solutions and ways to schedule them.

1.3 Content of this thesis

Chapter 2 presents the essential concepts for the understanding of this work. We describe the processor model, the S-Task model and various P-Task model used in this work in Section 2.2. We then clarify the concept of *scheduler* and propose some example of priority assignment in Section 2.3. Since models and schedulers are known, Section 2.4 expounds the concepts of schedulability and feasibility with some examples of results for the uniprocessor case. Finally, we summarize some results for the multiprocessor case in Section 2.5.

Chapter 3 presents our results for the scheduling of S-Tasks. We provide results for two different approaches: **P**artitioned **Scheduling** (P-Scheduling) in Section 3.2 and **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) in Section 3.3. For the P-Scheduling approach, we expound our generalized P-Scheduling algorithm and give an evaluation of its parameters. For the SP-Scheduling approach, we propose a solution for the **Rest**ricted **Migration** (Rest-Migration) case (migration are allowed but only *between* the activations of the tasks) in Subsection 3.3.2 and a solution for the **UnRest**ricted **Migration** (UnRest-Migration) case (migration are allowed anytime) in Section 3.3.3.

Chapter 4 presents our results for the scheduling of P-Tasks. We present two task models used including our new model in Section 4.3. Section 4.4 defines and summarize the schedulers used with P-Tasks. We put forward our results on the schedulability of our new task model in Section 4.5. Sections 4.6 and 4.7 end this chapter with an evaluation to compare the advantages and disadvantages of different types of P-Tasks.

Chapter 5 presents the tool **F**ramework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation (FORTAS), developed as part of this thesis. The existing tools presented in Section 5.2 provide a valuable aid for the analysis of RT systems. However, it seemed that almost all of them focus on the analysis or design of a given scheduling: given my platform, or even my task set, what will be the performance or how do I have to change my system to ensure its schedulability? FORTAS implements some of these elements but often remains less advanced than existing tools. However, it focuses on the possibility to automate the comparison and the evaluation of scheduling algorithms, whether based on a theoretical analysis of feasibility or on the simulation of scheduling, without necessarily focusing on a given platform or a specific task set.

Chapter 6 provides a conclusion of this work and suggests some perspectives.

Chapter 2

Introduction to RT Scheduling

N'a de convictions que celui qui n'a rien approfondi.

We have convictions only if we have studied nothing thoroughly.

Cioran [Cio86]

Contents

2.1 Introduction

We dedicated this chapter to the presentation of the main concepts and definitions used in RT systems. First, in Section 2.2, we propose to formally define the two main elements of a RT system: processors and tasks. Then, in Section 2.3, we explain how to execute the tasks on the processors by using schedulers. Section 2.4 shows some results which allow us to verify that the scheduler previously introduced is able to execute all the tasks on the processors while meeting all time constraints. Finally, we introduce some existing results for the multiprocessor case in Section 2.5.

2.2 System models

The main part of a RT system consists of *tasks*, i.e. of computer *processes*, and of *processors*, i.e. of computer *central processing unit*. We present in Subsection 2.2.1 and Subsection 2.2.2 various models and constraints related to these two main components of RT systems.

2.2.1 Processor model

In this work, we use the simple processor model given by Definition 2.1. We do not consider specific processor architectures, caches, pipelines etc.

Definition 2.1 (Processor set)**.**
Let $\pi = \{\pi_1, \ldots, \pi_m\}$ be a processor set composed of m processors. A processor π_k is characterized by the 2-tuple (ν_k, Δ_k) where:

- ν_k is the relative *speed* of π_k. The speed is said to be relative since ν_k represents the factor by which the time unit has to be multiplied to be consistent on processor π_k. For example, if a process has an execution time equal to C on a reference processor of speed 1, its execution time will be equal to $C \times \nu_k$ on processor π_k. Hence, $\forall k, l \in [\![1; m]\!]$, $\nu_k < \nu_l$ indicates that processor π_k is faster than processor π_l.

- Δ_k is the granularity of time on π_k. The granularity of time of a processor represents the minimum time unit which can be executed on this processor. E.g. if $\Delta_k = 0.1$ for processor π_k, a process which needs exactly 2.32 unit of time will take $\lceil {}^{2.32}/_{\Delta_k} \rceil \times \Delta_k = 2.4$ unit of time on processor π_k since 2.3 is not sufficient and the minimum time unit that can be executed on this processor is 0.1.

■

We now define the three main types of processor set:

Heterogeneous refers to a processor set composed of processors with **different speeds** and **different architectures** (for example, a processor $i7$ from Intel with Nehalem architecture and a processor Snapdragon S2 from Qualcomm with a ARMv7 architecture).

Homogeneous refers to a processor set composed of processors with possibly **different speeds** but **identical architectures**.

Identical refers to a processor set composed of processors with **identical speeds** and **identical architectures**.

In this work, we consider an identical processor set $\pi = \{\pi_1, \ldots, \pi_m\}$ with $\forall k \in [\![1; m]\!]$, $\nu_k = 1$ and $\Delta_k = 1$. Then, each time value presented in this document has to be read as a multiple of $\Delta_k = 1$. If you consider that Δ_k is expressed in millisecond, then you can read this document considering each time value as milliseconds. We do not define any specific time unit so that our results can be adapted to any time unit.

2.2.2 Task models

In this work, we consider various task models depending on the use of parallelism. In Subsection 2.2.2.2 we present the task model for S-Tasks, that is tasks which use at most one processor at each time instant. In Subsection 2.2.2.3 we present some task models for P-Tasks, that is tasks which may use more than one processor at a given time instant. In this work, we assume that all tasks are *independent*, that is, there is no shared resource, no precedence constraint and no communication between tasks. We now define the main time constraints used by each task model.

2.2.2.1 Task parameters and definitions

Some time constraints are often applied to tasks, whatever the task model. We present in this section the two main parameters of a task: the periodicity and the deadline. Then, we define what is a pre-emption, a migration of a task and the various states of a task during the system life.

Periodicity We consider that a task can be activated more than once during the system life. Each activation creates an instance of the task called *job*. We then have Definition 2.2 and Definition 2.3.

Definition 2.2 (Task)**.**
A *task* is defined as the set of common *off-line* properties of a set of works that need to be done. By analogy with object-oriented programming, a task can be seen as a *class*. ■

Definition 2.3 (Job)**.**
A *job*, or *instance* of task, is the *runtime* occurrence of a task. By analogy with object-oriented programming, a job can be seen as the *object* created after instantiation of the corresponding task class. In computer science, a job can be seen as a *computer process*. ■

To represent the recurrence of tasks, we use the principle of periodicity. We distinguish two types of periodicity:

Periodic task refers to a task which is always reactivated after a fixed duration called *inter-arrival time*. Thus, after the first activation of a task, it will be indefinitely reactivated and two successive activations will be always separated by the same duration.

Sporadic task refers to a task which can be reactivated at any instant after a specific duration called *minimum inter-arrival time*. Thus, if a task is activated, the next activation cannot occur during a fixed duration, but after this minimum inter-arrival time, it can occur at any instant.

Moreover, we distinguish two types of task sets accordingly to the first arrival instant of their tasks:

Synchronous refers to a task set composed of tasks which are all activated for the first time at the *same* instant.

Asynchronous refers to a task set composed of tasks which are activated for the first time at *different* instants.

Deadline As expressed in the introduction Section 1.1, a RT system is a specific system where we need to respect some time constraints. The typical one is the *deadline*. When a task is activated, it has to be executed *before* a specific time instant. We distinguish the *relative deadline* which is the *duration* available to execute the task starting from its activation, and the *absolute deadline* which is the latest *time instant* at which the task has to be fully executed. For example, if the relative deadline of a task is equal to 4 and this task is activated at time instant 23, the absolute deadline will be equal to $24 + 5 = 27$. We identify three types of relative deadlines:

I-Deadline refers to a task with an **I**mplicit **Deadline** (I-Deadline) i.e. a task with a relative deadline **equal to** its inter-arrival time. In other word, the task must, implicitly, be completed at its reactivation.

C-Deadline refers to a task with a **C**onstrained **Deadline** (C-Deadline) i.e. a task with a relative deadline **equal to or less than** its inter-arrival time. In other words, the task must be completed before its reactivation.

A-Deadline refers to a task with a **A**rbitraty **Deadline** (A-Deadline) i.e. a task with a relative deadline **equal to, less than or greater than** its inter-arrival time. In other word, the completion of the task does not depend on its reactivation.

Pre-emption A pre-emption occurs when *a job* executed on *one processor* is interrupted to execute *a job of an other task* on *the same processor*. In a real system, pre-emption is usually composed of four phases:

1. interrupt the executed job on the processor,
2. save the execution context of the previously executed job (program counter, registers values etc.),
3. load the execution context of the new job,
4. execute the new job on the processor.

These phases can take some time and a pre-emption is not "free". However, we relax this constraint in this work and we neglect the additional time produced by a pre-emption: we suppose it is included in the execution time of the task or it is equal to zero.

Migration A migration occurs when a job executed on *one processor* is interrupted (or interrupts itself) to execute *the same job* on *another processor*. In a real system, a migration is usually composed of four phases:

1. interrupt the executed job on the first processor,
2. save the execution context of the job (program counter, registers values etc.),
3. load the execution context of the job on the new processor,
4. the job is ready to be executed on the new processor.

Just as pre-emption, these phases can take some time but we relax this constraint in this work and we neglect the additional time produced by a migration. However, we will study the number of migrations of some algorithms to compare their relative deviation from an actual implementation.

Tasks and jobs states A task, during the system life, can be in four different states. Figure 2.1 gives these states and the transitions.

Ready refers to the state in which the task has been activated but the corresponding job is not currently executing on a processor.

Running refers to the state in which a job of the task is executing on a processor. We also say that the task is *scheduled* on the processor.

Blocked refers to the state in which a job of the task was executed but it has been stopped by the system. For example, when a job needs to take a mutex, it goes to the *Blocked* state while the mutex is unavailable.

Inactive refers to the state in which a task is not active in the system. For example, if the previous job of the task has completed its execution, before its reactivation the task is inactive.

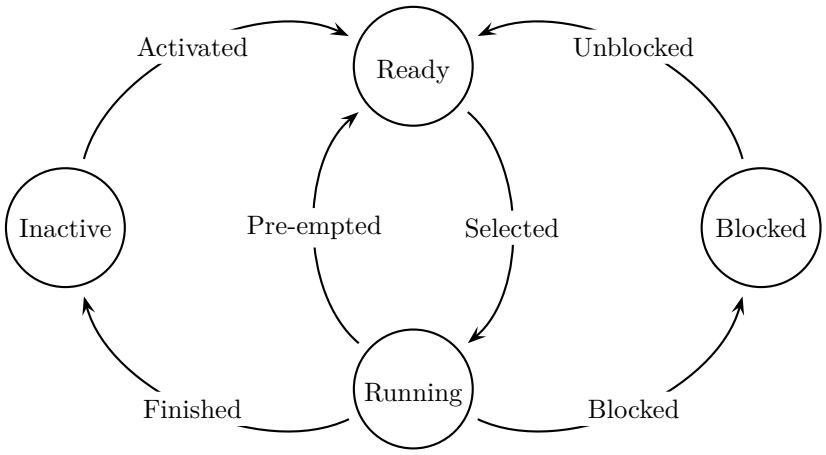


Figure 2.1 – States and transitions of a task during the system life

2.2.2.2 Sequential Task (S-Task) model

A task is said to be *sequential* if it can use one and only one processor at each time instant of its execution. We now give the definitions of a periodic (Definition 2.4 and Figure 2.2) and a sporadic (Definition 2.5) sequential task set which are based on the work of Liu and Layland [LL73]. For notations we were inspired by those used in the work of Cucu-Grosjean and Goossens [CGG11].

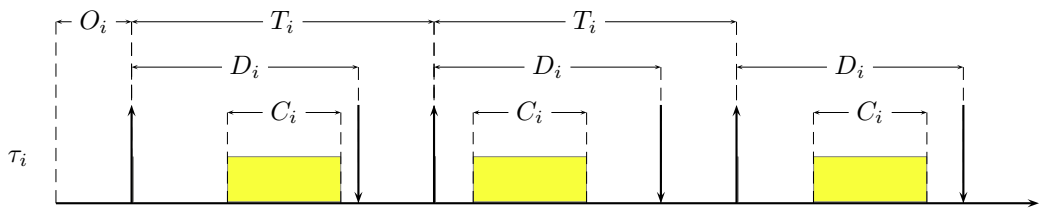


Figure 2.2 – Representation of a periodic sequential task, from Definition 2.4

Definition 2.4 (Periodic sequential task set)**.**
Let $\tau_{(O,C,T,D)} = \{\tau_1(O_1, C_1, T_1, D_1), \ldots, \tau_n(O_n, C_n, T_n, D_n)\}$ be a periodic sequential task set composed of n periodic sequential tasks. The task set $\tau_{(O,C,T,D)}$ can be abbreviated as τ. A periodic sequential task $\tau_i(O_i, C_i, T_i, D_i)$, abbreviated as τ_i (Figure 2.2), is characterized by the 4-tuple (O_i, C_i, T_i, D_i) where:

- O_i is the *first arrival instant* of τ_i, i.e., the instant of the first activation of the task since the system initialization.
- C_i is the ***W**orst **C**ase **E**xecution **T**ime (WCET)* of τ_i, i.e., the maximum execution time required by the task to complete.
- T_i is the *period* of τ_i, i.e., the exact inter-arrival time between two successive activations of τ_i.
- D_i is the *relative deadline* of τ_i, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant.

■

Definition 2.5 (Sporadic sequential task set)**.**
Let $\tau_{(C,T,D)} = \{\tau_1(C_1, T_1, D_1), \ldots, \tau_n(C_n, T_n, D_n)\}$ be a sporadic sequential task set composed of n sporadic sequential tasks. The task set $\tau_{(C,T,D)}$ can be abbreviated as τ. A sporadic sequential task $\tau_i(C_i, T_i, D_i)$, abbreviated as τ_i, is characterized by the 3-tuple (C_i, T_i, D_i) where:

- C_i is the *WCET* of τ_i, i.e., the maximum execution time required by the task to complete.
- T_i is the *minimum inter-arrival time* of τ_i, i.e., the minimum time between two successive activations of τ_i.
- D_i is the *relative deadline* of τ_i, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant.

■

Each task τ_i generates an infinite sequence of jobs.

Notice that for any sequential task set $\tau_{(O,C,T,D)}$ (periodic) or $\tau_{(C,T,D)}$ (sporadic), (O, C, T, D) are respectively the vectors of first arrival instants (O), WCETs (C), inter-arrival times (T) and deadlines (D) of tasks in τ. For instance, C_2, the second value in vector C is the WCET of task τ_2 in τ. In this way, we give the following examples:

- $\tau_{(C,T,D)}$ is a task set of n sporadic sequential tasks where $C = (C_1, \ldots, C_n)$, $T = (T_1, \ldots, T_n)$, $D = (D_1, \ldots, D_n)$ are respectively the sets of WCETs, periods (or minimum inter-arrival times) and deadlines of the tasks in τ. A task $\tau_i \in \tau$ is defined by the i^{th} element of the three sets C, T and D.

- $\tau_{(O,C,T,D)}$ is a task set of n periodic sequential tasks where $O = (O_1, \ldots, O_n)$, $C = (C_1, \ldots, C_n)$, $T = (T_1, \ldots, T_n)$, $D = (D_1, \ldots, D_n)$ are respectively the sets of first arrival instant, WCETs, periods (or exact inter-arrival times) and deadlines of the tasks in τ. A task $\tau_i \in \tau$ is defined by the i^{th} element of the four sets O, C, T and D.
- τ is an abbreviation of $\tau_{(O,C,T,D)}$ in a periodic context or $\tau_{(C,T,D)}$ in a sporadic context.
- $\tau_{(X,T,D)}$ is a sporadic sequential task set where $X = (x_1, \ldots, x_n)$ is a set of WCETs variables, T and D are sets of fixed periods and deadlines.
- $\tau_{(C,T,D/p)}$ denotes a set of sporadic sequential tasks with a set of fixed WCETs C, a set of fixed periods T and a set of fixed deadlines where all deadlines in D are divided by $p \geqslant 1$ according to an original task set $\tau_{(C,T,D)}$.
- $\tau_{(C,pT,D)}$ denotes a set of sporadic sequential tasks with a set of fixed WCETs C, a set of fixed deadlines D and a set of fixed periods where all periods in T are multiplied by $p \geqslant 1$ according to an original task set $\tau_{(C,T,D)}$.

2.2.2.2.1 Metrics for S-Task sets A task set composed of S-Tasks is also characterized by some metrics. We define in this section the most commonly used and especially the metrics used in this work.

A S-Task is characterized by the following metrics:

Utilization The utilization of a S-Task τ_i is given by Equation 2.1.

$$U_{\tau_i} \stackrel{\text{def}}{=} \frac{C_i}{T_i} \tag{2.1}$$

Density The density of a S-Task τ_i is given by Equation 2.2.

$$\Lambda_{\tau_i} \stackrel{\text{def}}{=} \frac{C_i}{\min(D_i, T_i)} \tag{2.2}$$

A S-Task set is characterized by the following metrics:

Utilization The utilization of a task set τ composed of n S-Tasks is given by Equation 2.3.

$$U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_{\tau_i} \tag{2.3}$$

Density The density of a task set τ composed of n S-Tasks is given by Equation 2.4.

$$\Lambda_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \Lambda_{\tau_i} \tag{2.4}$$

RBF The **R**equest **B**ound **F**unction (RBF) of a task set τ composed of n preemptive synchronous S-Tasks represents the upper bound of the work load generated by all tasks with activation instants included within the interval $[0; t)$. Lehoczky, Sha, and Ding [LSD89] gave Equation 2.5 which allows us to compute the RBF.

$$RBF(\tau, t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil \times C_i \tag{2.5}$$

DBF The **D**emand **B**ound **F**unction (DBF) of a task set τ composed of n preemptive synchronous S-Tasks represents the upper bound of the work load generated by all tasks with activation instants and absolute deadlines within the interval $[0; t]$. Baruah, Rosier, and Howell [BRH90] gave Equation 2.6 which allows us to compute the DBF.

$$DBF(\tau, t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) \times C_i \tag{2.6}$$

WCRT The **W**orst **C**ase **R**esponse **T**ime (WCRT) of a task $\tau_i \in \tau$ is the maximum duration between the activation of the task and the instant it completes its execution.

2.2.2.3 Parallel Task (P-Task) model

A task is said to be *parallel* if it is allowed to use more than one processor during its execution. As presented in the introduction Section 1.2, multiple approaches have been proposed to distribute the computation. Whatever the approach, considering distributed processors or multi-core processors, a crucial point remains the *communication between parallel threads*. If two parallel threads need to exchange information, they may have to wait for each other.

In the field of RT scheduling, researchers tackled the issue of parallel treatments few years ago. As a result, they offered a variety of models to describe the so-called P-Tasks. These task models are based on a different view of synchronization points (or communication points) between parallel threads. We define two classes of parallel task models:

- *Gang class* where parallel threads are considered and scheduled in unison.
- *Multi-Thread class* where parallel threads can be considered and scheduled independently.

We present in the following the main model of each class:

- Gang is a task model of the eponymous class. It is derived from the scheduler used on supercomputer — especially in the “Connection Machine”

CM-5 created in 1991 [Cor92; Fei96]. This scheduler considers that threads of a process must communicate very often with each other. The easiest way to reduce their waiting time is then to schedule all threads of each process together. A Gang task is defined by an execution requirement which corresponds to a "$C_i \times V_i$" rectangle, with the interpretation that a process requires *exactly* V_i processors *simultaneously* for a duration of C_i time units. This model is detailed in Subsection 2.2.2.3.1. Schedulers using this class of task model are called *Gang schedulers*.

- Fork-Join is a task model of the Multi-Thread class. It is derived from parallel programming paradigm such as **POSIX thread** (Pthread) and OpenMP. This model considers each task as a sequence of segments (or phases), alternating between sequential and parallel phases. During a parallel phase, threads are completely independent and only wait for each other for starting the next sequential phase. There are alternating "fork" (separation into independent threads) and "join" (waiting for thread completion). This model is detailed in Subsection 2.2.2.3.2. Schedulers using this class of task model are called *Multi-Thread schedulers*.

2.2.2.3.1 Gang task model The following model is based on the work of Kato and Ishikawa [KI09]. We define the periodic parallel Gang task model in Definition 2.6 and give an example in Figure 2.3.

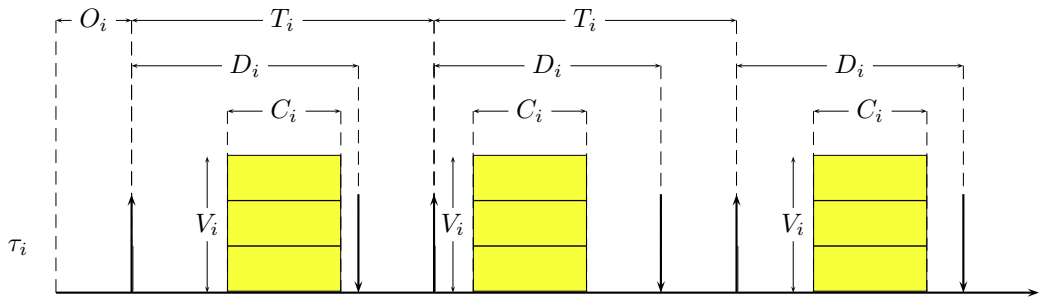


Figure 2.3 – Representation of a periodic parallel Gang task, from Definition 2.6

Definition 2.6 (Periodic parallel Gang task set)**.**
Let $\tau_{(O,C,T,D,V)} = \{\tau_1(O_1, C_1, T_1, D_1, V_1), \ldots, \tau_n(O_n, C_n, T_n, D_n, V_n)\}$ be a periodic parallel Gang task set composed of n periodic parallel Gang tasks. The task set $\tau_{(O,C,T,D,V)}$ can be abbreviated as τ. A periodic parallel Gang task $\tau_i(O_i, C_i, T_i, D_i, V_i)$, abbreviated as τ_i (Figure 2.3), is characterized by the 5-tuple $(O_i, C_i, T_i, D_i, V_i)$ where:

- O_i is the *first arrival instant* of τ_i, i.e., the instant of the first activation of the task since the system initialization.

- C_i is the *WCET* of τ_i when executed in parallel on V_i processors, i.e., the maximum execution time required simultaneously on V_i processors by the task to complete.
- T_i is the *period* of τ_i, i.e., the exact inter-arrival time between two successive activations of τ_i.
- D_i is the *relative deadline* of τ_i, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant.
- V_i is the number of processors used *simultaneously* to schedule τ_i.

■

Each task τ_i generates an infinite sequence of jobs. Each job of τ_i is executed in parallel on V_i processors (by V_i threads) during C_i time units. Kato and Ishikawa [KI09] assumed that all threads within the job consume C_i time units including idle and waiting times to synchronize with each other, even if in fact perfect parallelism may not be possible. In conclusion, the execution of a job of τ_i is represented as a "$C_i \times V_i$" rectangle in "time $\times$ processor" space.

Remark 2.1. All threads of a Gang task have to execute simultaneously, so V_i processors need to be available at the same time instant to schedule a Gang task. ◇

2.2.2.3.2 Fork-Join task model The following model is based on the work of Lakshmanan, Kato, and Rajkumar [LKR10]. They propose this model referring from an existing paradigm used in various parallel programming models such as OpenMP [Cha+00] or Pthread. We define the periodic parallel Fork-Join task model in Definition 2.7 and give an example in Figure 2.4.

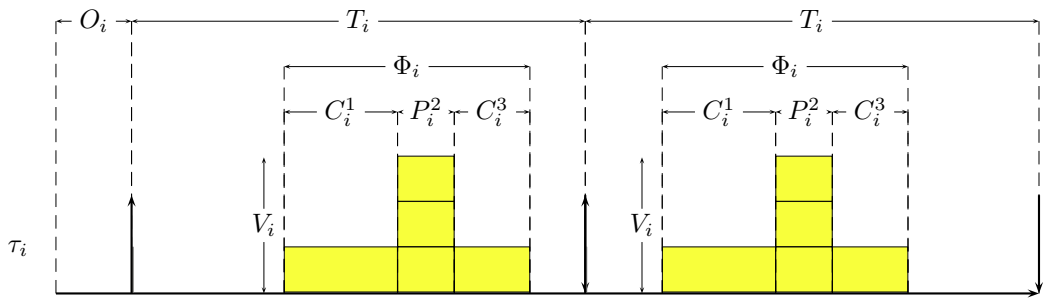


Figure 2.4 – Representation of a periodic parallel Fork-Join task, from Definition 2.7

Definition 2.7 (Periodic parallel Fork-Join task set)**.**
Let $\tau_{(O,\Phi,T,V)} = \{\tau_1(O_1, \Phi_1, T_1, V_1), \ldots, \tau_n(O_n, \Phi_n, T_n, V_n)\}$ be a periodic parallel Fork-Join task set composed of n periodic parallel Fork-Join tasks. The task set $\tau_{(O,\Phi,T,V)}$ can be abbreviated as τ. A periodic parallel Fork-Join task $\tau_i(O_i, \Phi, T_i, V_i)$ (Figure 2.4), abbreviated as τ_i (Figure 2.4), is characterized by the 4-tuple (O_i, Φ_i, T_i, V_i) where:

- O_i is the *first arrival instant* of τ_i, i.e., the instant of the first activation of the task since the system initialization.
- $\Phi_i = \left\{C_i^1, P_i^2, C_i^3, P_i^4, \ldots, P_i^{s_i-1}, C_i^{s_i}\right\}$ is the set of s_i *computation segments* of τ_i. Thus, the total computation of the task is divided in s_i successive parts called computation segments.
- T_i is the *period* of τ_i, i.e., the exact inter-arrival time between two successive activations of τ_i. The relative deadline is equal to the period, $D_i = T_i$.
- V_i is the number of parallel threads used in each parallel segment of τ_i. If $V_i = 1$, the task is sequential. Lakshmanan, Kato, and Rajkumar [LKR10] assume that $V_i \leqslant m$ with m the total number of identical processors (or cores).

■

If we focus on Φ_i, the set of computation segments, we can identify sequential and parallel segments:

- C_i^j *with* j *is an odd number* is the WCET of the j^{th} segment which refers to a sequential segment. This sequential segment is also referred to as $\tau_i^{j,1}$.
- P_i^j *with* j *is an even number* is the WCET of the V_i threads of the j^{th} segment which refers to a parallel segment. Each thread in this segment is also referred to as $\tau_i^{j,k}$ with $k \in [\![1; V_i]\!]$. All threads are assumed independent from each other but all threads of segment i need to complete before the execution of the next segment $(i + 1)$.

Remark 2.2. The set of computation segments is composed of alternating between sequential and parallel segments. A task always start with a sequential segment and finishes with a sequential segment. Then, s_i is an odd number. ◇

Remark 2.3. Since the number of parallel threads V_i is defined at the task level, all parallel segments of one task have the same number of threads. ◇

2.3 Schedulers

After Subsections 2.2.1 and 2.2.2, tasks and processors have no more secrets for you. Now, we have to execute and so, *schedule* the tasks on the processors! "Schedule" literally means "set a timetable" and in most cases of everyday life, when you set a timetable, you know roughly the things you will need to do and when you will have to begin. Notwithstanding, in the RT research field, we distinguish between two cases:

Clairvoyant refers to the case that we know the future of the system and especially when tasks will be activated.

Non-clairvoyant refers to the case that we do not know the future of the system.

In the vast majority of research results on RT systems, we consider to be in the *non-clairvoyant* case. This corresponds fairly well to the reality of industrial systems. Thus, without any clear specification, a *scheduler* refers to a *non-clairvoyant scheduler*. But, first of all, what is a scheduler?

Definition 2.8 (Scheduler)**.**
A (non-clairvoyant) scheduler is an algorithm which has to select, at each time instant, in the list of *Ready* and *Running* tasks, which jobs should be executed on the processors. To this end, it assigns priorities and selects the jobs of the tasks with the highest priorities. Hence, if the list of ready tasks is composed of tasks τ_i and τ_j with $\tau_i \succ \tau_j$ (τ_i has a higher priority than τ_j) and we have one processor, then the scheduler will choose τ_i to be executed on this processor. ■

Definition 2.9 (Optimal scheduler)**.**
A (non-clairvoyant) scheduler is *optimal* if this scheduler meets all deadlines when a task set can be scheduled by at least one (non-clairvoyant) scheduler without missing any deadline. In other words, if it exists a scheduler which succeeds in meeting all deadlines of a task set, then the optimal scheduler will also meet all deadlines. The contrapositive is also true: if, for a given task set, an optimal scheduler misses at least one deadline then there is not any existing (non-clairvoyant) scheduler which can successfully meet all deadlines. ■

In this section we present the main uniprocessor schedulers used in the state-of-the-art. We consider two types of schedulers:

FTP refers to **F**ixed **T**ask **P**riority (FTP), a scheduler which defines fixed priority to each task. The task priorities are defined before starting the system then they never change. Subsection 2.3.1 presents some of these schedulers.

DTP refers to **D**ynamic **T**ask **P**riority (DTP), a scheduler which defines dynamic priority to each task. The task priorities are not known when the system starts and a task can have different priorities during the system life. Subsection 2.3.2 presents some of these schedulers.

Remark 2.4. A scheduler can be pre-emptive (it allows pre-empting tasks) or non-pre-emptive (it forbids pre-empting tasks). In this work, we only consider pre-emptive schedulers. Thus, without clear specification, a *scheduler* refers to a *non-clairvoyant pre-emptive scheduler*. ◇

2.3.1 Fixed Task Priority (FTP) schedulers

A FTP scheduler is the classical way to handle RT systems. In this case, all decisions on priorities are taken before starting the system, thus the scheduler considers only this fixed value to select the new running task at each time instant. These schedulers are the main used uniprocessor schedulers in the state-of-the-art and they are mainly used in the industry.

FTP schedulers differ in how they assign fixed priorities to tasks. Here are the most studied FTP schedulers:

RM refers to **R**ate **M**onotonic (RM) scheduler studied by Liu and Layland [LL73]. It assigns priorities to tasks according to their period: more often a task is reactivated, the higher its priority. For example, if $T_i < T_j$ then $\tau_i \succ \tau_j$. For pre-emptive synchronous sporadic S-Tasks with I-Deadlines on uniprocessor platforms, RM is an optimal FTP scheduler [LL73]. It is optimal in the sense that if a task set can be scheduled by a FTP scheduler without missing any deadline, then RM will also meet all deadlines.

DM refers to **D**eadline **M**onotonic (DM) scheduler studied by Audsley et al. [Aud+91]. It assigns priorities to tasks according to their relative deadline: the shorter the relative deadline, the higher its priority. For example, if $D_i < D_j$ then $\tau_i \succ \tau_j$. For pre-emptive synchronous sporadic S-Tasks with C-Deadlines on uniprocessor platforms, DM is an optimal FTP scheduler [LM80].

OPA refers to **O**ptimal **P**riority **A**ssignment (OPA) scheduler proposed by Audsley [Aud01; Aud91]. It is an optimal algorithm to assign fixed priority to synchronous pre-emptive sporadic S-Tasks with A-Deadline [Aud91] and to synchronous non-pre-emptive sporadic S-Tasks with A-Deadlines [GRS96]. It is optimal in the sense that if it does not find a priority assignment which can meet all deadlines, then it does not exist any fixed priority assignment which can meet all deadlines. OPA has been proposed in order to improve the solution which consists in listing all possible priority orderings. Indeed, for a set of n tasks, $n!$ combinations have to be considered for the exhaustive enumeration whereas there are n^2 tests with the OPA algorithm.

2.3.2 Dynamic Task Priority (DTP) schedulers

A DTP does not assign fixed priority to tasks but recomputes the priorities during the system life. These schedulers typically allow scheduling more task sets but they often require additional time while the system is running to compute the new priorities.

DTP schedulers differ in how they assign the dynamic priorities to tasks. Again, we distinguish two sub-types of DTP schedulers:

FJP refers to **F**ixed **J**ob **P**riority (FJP), a scheduler which defines fixed priority to each job. The job priorities are defined when they are activated then they never change, but two jobs of the same task can have different priorities.

DJP refers to **D**ynamic **J**ob **P**riority (DJP), a scheduler which defines dynamic priority to each job. A job can have different priorities during its execution.

Here are the most studied DTP schedulers:

EDF refers to **E**arliest **D**eadline **F**irst (EDF) scheduler which is a FJP scheduler. It has been studied by Liu and Layland [LL73] and proven as optimal uniprocessor scheduler for pre-emptive sporadic S-Tasks with A-Deadlines by Dertouzos [Der74] and for synchronous non-pre-emptive sporadic S-Tasks with A-Deadlines by Jeffay, Stanat, and Martel [JSM91]. It assigns priorities to jobs according to their absolute deadline: the shorter the absolute deadline, the higher its priority. It is optimal in the sense that if a task set can be scheduled without missing any deadline, then EDF will also meet all deadlines.

LLF refers to **L**east **L**axity **F**irst (LLF) scheduler which is a DJP scheduler. It was proposed by Mok [Mok83; Leu89] and proven as optimal uniprocessor scheduler for pre-emptive sporadic S-Tasks with A-Deadlines. It assigns priorities to jobs according to the remaining slack of time before their absolute deadline. The slack, or laxity, of a task at any time instant is defined as remaining time to deadline minus the amount of remaining execution.

2.4 Feasibility and schedulability analysis

We have presented various schedulers which assign priorities and select tasks in different ways in order to execute them on the processors. But the main objective of the study of RT systems is not only to execute tasks, it has to verify and even guarantee that all these executed tasks will meet their deadline. In this section we present some theories, applied to various schedulers, which allow us to verify

before starting the system if a task set is schedulable with a given scheduler, or even if a task set is feasible.

Wait. A task set could be "feasible" but not "schedulable"? What is the difference?

2.4.1 Feasibility or schedulability?

These two notions have been defined differently according to researchers. Some of us use "schedulable" to describe the property of a task set that other researchers call "feasible". Here, we give the two definitions considered in this work. Notice that the definitions given in this section are based on the proposals presented in the work of Davis and Burns [DB11].

Definition 2.10 (Feasible)**.**
A task set is *feasible* if it exists, at least, *one solution* to schedule this task set which meet all deadlines. This solution may require a non-clairvoyant scheduler or a clairvoyant scheduler. ■

Definition 2.11 (Schedulable)**.**
A task set is *schedulable* according to *one scheduler* if *this* scheduler can meet all deadlines. ■

Notice that, if a task set is schedulable, it is necessarily feasible. The other way round is not always true: a task set can be feasible but not necessarily schedulable with a given scheduler, even if this scheduler is *optimal*. Indeed, an optimal (non-clairvoyant) scheduler can fail to schedule a task set which is feasible. This can happen especially when only a clairvoyant scheduler would be able to successfully schedule the task set.

2.4.1.1 Necessary, sufficient or necessary and sufficient?

To verify the schedulability or the feasibility of task sets, we generally propose schedulability or feasibility tests. These tests can be classified in different ways as given in the Definitions 2.12, 2.13 and 2.14.

Definition 2.12 (**N**ecessary **Test** (N-Test))**.**
A test is said to be a **N**ecessary **Test** (N-Test) if a negative result allows us to reject the proposition but a positive result does not allow us to accept the proposition. In other words, if this test is positive, we can continue to hope that the proposition is true, but if this test is negative, it is certain that the proposition is false. ■

Example for Definition 2.12 For a periodic sequential task set τ composed of n tasks with I-Deadlines scheduled with pre-emptive RM scheduler on uniprocessor platform, $U_\tau \leqslant 1$ is a schedulability N-Test. Thus, if $U_\tau \leqslant 1$, τ may or may not be schedulable with RM scheduler. However, if $U_\tau \leqslant 1$, τ is undoubtedly not schedulable with RM scheduler.

Definition 2.13 (**S**ufficient **Test** (S-Test))**.**
A test is said to be a **S**ufficient **Test** (S-Test) if a positive result allows us to accept the proposition but a negative result does not allow us to reject the proposition. In other words, if this test is negative, we have to continue to expect that the proposition is false, but if this test is positive, it is certain that the proposition is true. ■

Example for Definition 2.13 For a periodic sequential task set τ composed of n tasks with I-Deadlines scheduled with pre-emptive RM scheduler on uniprocessor platform, $U_\tau \leqslant n\left(\sqrt[n]{2}-1\right)$ is a schedulability S-Test proposed by Liu and Layland [LL73]. Thus, if $U_\tau > n\left(\sqrt[n]{2}-1\right)$, τ may be schedulable with RM scheduler, but it also may not be schedulable with RM scheduler. However, if $U_\tau \leqslant n\left(\sqrt[n]{2}-1\right)$, τ is undoubtedly schedulable with RM scheduler.

Definition 2.14 (**N**ecessary and **S**ufficient **Test** (NS-Test))**.**
A test is said to be a **N**ecessary and **S**ufficient **Test** (NS-Test) if a positive result allows us to accept the proposition and a negative result allows us to reject the proposition. In other words, if this test is positive, it is certain that the proposition is true and if this test is negative, it is certain that the proposition is false. Ultimately, a NS-Test always gives an undoubted response. ■

Example for Definition 2.14 For a periodic sequential task set τ composed of n tasks with I-Deadlines scheduled with pre-emptive EDF scheduler on uniprocessor platform, $U_\tau \leqslant 1$ is a schedulability NS-Test proposed by Liu and Layland [LL73]. Thus, if $U_\tau < 1$, τ is undoubtedly schedulable with EDF scheduler. Moreover, if $U_\tau > 1$, τ is undoubtedly not schedulable with EDF scheduler.

2.4.2 Schedulability analysis for FTP schedulers on uniprocessor platform

We present in this section some existing schedulability tests for FTP schedulers and uniprocessor platforms.

- For any FTP scheduler and any type of deadline, Equation 2.7 gives a schedulability N-Test for a task set τ.

$$U_\tau \leqslant 1 \tag{2.7}$$

- For pre-emptive RM scheduler and tasks with I-Deadlines, Liu and Layland [LL73] proposed the schedulability S-Test given by Equation 2.8.

$$U_\tau \leqslant n\left(\sqrt[n]{2}-1\right) \tag{2.8}$$

- For pre-emptive RM scheduler and tasks with I-Deadlines, Bini, Buttazzo, and Buttazzo [BBB03] proposed the schedulability S-Test given by Equation 2.9.

$$\prod_{i=1}^{n}\left(U_{\tau_i}+1\right) \leqslant 2 \tag{2.9}$$

- For any scheduler, Joseph and Pandya [JP86] showed that a schedulability NS-Test is to verify that the WCRT of each task is lower than its relative deadline, as expressed by Equation 2.10.

$$\forall \tau_i \in \tau, WCRT_i \leqslant D_i \tag{2.10}$$

For pre-emptive DM scheduler and tasks with C-Deadlines, Audsley et al. [Aud+93] proposed the schedulability NS-Test given by Equation 2.10 with the computation of WCRT given by Equation 2.11.

$$\forall \tau_i \in \tau,\ WCRT_i \text{ is the solution of}$$
$$\begin{cases} WCRT_i^0 & = C_i \\ WCRT_i^{k+1} & = C_i + RBF\left(\tau^{hp(\tau,\tau_i)}, WCRT_i^k\right) \\ \text{until} & WCRT_i^{k+1} = WCRT_i^k \text{ or } WCRT_i^k > D_i \end{cases}$$
$$\text{with } \tau_j \in \tau^{hp(\tau,\tau_i)} \text{ if } \tau_j \in \tau \text{ and } \tau_j \succ \tau_i \tag{2.11}$$

2.4.3 Schedulability analysis for DTP schedulers on uniprocessor platform

We present in this section some existing schedulability tests for DTP schedulers and uniprocessor platforms.

- For any DTP scheduler and any type of deadline, Equation 2.12 gives a schedulability N-Test for a task set τ.

$$U_\tau \leqslant 1 \tag{2.12}$$

- For pre-emptive EDF scheduler and tasks with C-Deadlines, Liu [Liu00] confirms the schedulability S-Test given by Equation 2.13.

$$\Lambda_\tau \leqslant 1 \tag{2.13}$$

- For pre-emptive EDF scheduler and tasks with A-Deadlines, Baruah, Rosier, and Howell [BRH90] proposed the schedulability NS-Test given by Equation 2.14. Equation 2.15 is another form of the same schedulability test.

$$\forall t > 0, DBF(\tau, t) \leqslant t \tag{2.14}$$

$$Load(\tau) \stackrel{\text{def}}{=} sup_{t>0} \frac{DBF(\tau, t)}{t} \leqslant 1 \tag{2.15}$$

2.4.3.1 EDF uniprocessor schedulability condition: reconsideration

In our work, we mainly apply our results to the EDF scheduler case, so we propose to go deeper into the schedulability analysis of this scheduler. The schedulability NS-Test considered is originally proposed by Baruah, Rosier, and Howell [BRH90]: the *Load* function given in Equation 2.16. We also use the result given by George and Hermant [GH09a] which allows us to reduce the number of time instants to consider during the *Load* computation.

2.4.3.1.1 The *Load* function Let τ be a sporadic sequential task set as presented in Subsection 2.2.2.2. *Load* is the cumulative execution requirement generated by jobs of the tasks in τ on any time interval divided by the length of the interval. The *Load* function is given by Equation 2.16:

$$Load(\tau) \stackrel{\text{def}}{=} sup_{t>0} \frac{DBF(\tau, t)}{t} \tag{2.16}$$

The *Load* function provides a schedulability NS-Test for pre-emptive EDF scheduler on uniprocessor platform: $Load(\tau) \leqslant 1$ and it has been widely studied by various researchers. Theorem 2.1 is the result of Fisher, Baker, and Baruah [FBB06a] showing how to reduce the number of time instants to consider during the computation of this function.

Theorem 2.1 (*Load* function [FBB06a])**.**
Let τ be a sporadic sequential task set, the Load function can be computed as follow:

$$Load(\tau) = \max\left(U_\tau, sup_{t \in S} \frac{DBF(\tau, t)}{t}\right) \tag{2.17}$$

$$where\ S \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \left\{ D_i + k_i \times T_i, 0 \leqslant k_i \leqslant \left\lceil \frac{P - D_i}{T_i} \right\rceil - 1 \right\} \tag{2.18}$$

and P the least common multiple of all task period

■

In 2009, George and Hermant [GH09b] show that the previous expression can be used to characterize the space of feasible WCETs as presented in Theorem 2.2.

Theorem 2.2 (C-Space characterization [GH09b])**.**
Let $\tau_{(X,T,D)}$ be a sporadic sequential task set, with $X = (x_1, \ldots, x_n)$ are variables and vectors D and T are constants. The Load $\left(\tau_{(X,T,D)}\right) \leqslant 1$ condition gives $s+1$ constraints which characterize the space of feasible WCETs, with s is the number of elements in the set S defined by Theorem 2.1.

The first s constraints are given by Equation 2.19 and the $(s+1)^{\text{th}}$ constraint is given by Equation 2.20.

$$\forall k \in [\![1; s]\!], t_k \in S,$$

$$DBF\left(\tau_{(X,T,D)}, t_k\right) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{t_k - D_i}{T_i} \right\rfloor\right) \times x_i \leqslant t_k \tag{2.19}$$

$$U_{\tau_{(X,T,D)}} \leqslant 1 \tag{2.20}$$

■

They also show how to prune the set S to extract the subset of elements representing the most constrained time instants where $sup_{t>0} \frac{DBF(\tau_{(X,T,D)},t)}{t}$ can be obtained. To this end, for any time instant $t_j \in S$, they formalize as a linear programming problem the question of determining whether a time instant t_j is relevant or if it could be ignored. For each time instant t_j the goal is to maximize the objective function $DBF(\tau_{(X,T,D)}, t_j)$ taking into account the constraints given in Equation 2.19 excluding the one produced by time instant t_j. Therefore, these $s-1$ constraints are imposed on the WCETs of the tasks without considering the constraint associated to time instant t_j. The problem to be solved can then be characterized with a linear programming approach formally defined in LPP 2.1.

Linear Programming Problem (LPP) 2.1 (C-Space – Relevance of time instant t_j)**.**
The objective is to:

Maximize	$DBF\left(\tau_{(X,T,D)}, t_j\right)$
Under the constraints	$\bigcup_{k=1,k\neq j}^{s} \left\{DBF\left(\tau_{(X,T,D)}, t_k\right) \leqslant t_k\right\}$
With	$\forall i \in [\![1; n]\!], x_i \geqslant 0$

■

George and Hermant [GH09b] propose using the simplex algorithm to solve the linear programming problem given in LPP 2.1. If for time instant t_j,

$DBF\left(\tau_{(X,T,D)}, t_j\right) \leqslant t_j$ when the $s-1$ constraints of LPP 2.1 are imposed, then it is not necessary to add the constraint $DBF\left(\tau_{(X,T,D)}, t_j\right) \leqslant t_j$ to the problem since it is already respected with the other constraints. Hence, t_j is not significant for characterizing the space of feasible WCETs and it can then be removed from the set S. Otherwise, time instant t_j should be kept in the set S.

2.4.3.1.2 Performance of LPP 2.1 with the simplex We now study the performance of the simplex for pruning the elements in the set of time instants S in the case of C-Deadline task sets. In order to evaluate the impact of the simplex on the reduction of the elements in the set S, we produce 10^5 task sets of three tasks with $s \geqslant 3500$. Notice that the number s of constraints in the set S depends more on the value of the periods than on the number of tasks.

To generate each task set, we proceed as follows:

- the period of each task is uniformly chosen within the interval $[1; 100]$,
- the deadline of any task τ_i is computed as $D_i = \alpha T_i$. α is discretized within the intervals $[0; 0.8]$ and $[0.8; 1]$ with a granularity of respectively 0.1 and 0.025.

We focus on the influence of α on the pruning of the set S after executing the simplex in LPP 2.1. Figure 2.5 shows the results of our analysis. The original number of elements in the set S is associated with the left axis while the right axis is used in association with the number of elements obtained after the simplex is applied to LPP 2.1.

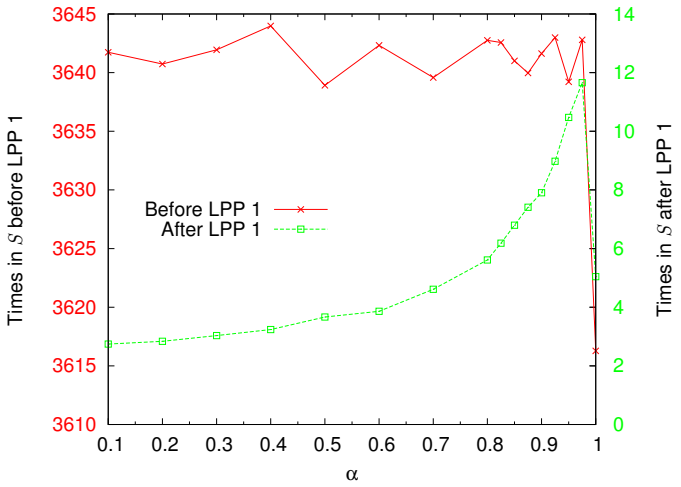


Figure 2.5 – Reduction of elements in the set S with LPP 2.1

We notice that the number of elements which curb the C-Space inch-up within the interval $[0.1; 0.6]$ then plunge downwards when α tends toward 1. If $\alpha = 1$,

we have the special case of I-Deadlines task sets where the only constraint is the utilization limit: $U_{\tau_{(X,T,D)}} \leqslant 1$.

In all cases, we found that the number of constraints before and after pruning the set S is respectively higher than 3570 and lower than 12. For a value of α lower than 0.6, the average number of constraints after pruning the elements in S is at most equal to 4. This confirms that the simplex can be very effective to reduce the number of elements characterizing the space of feasible WCETs. We use this property in Section 3.3.

2.4.3.1.3 Example using LPP 2.1 to compute the *Load* function Let us consider the example $\tau_{(X,T,D)} = \{\tau_1(x_1, T_1, D_1), \tau_2(x_2, T_2, D_2), \tau_3(x_3, T_3, D_3)\}$ of three sporadic sequential tasks where for any task $\tau_i(x_i, T_i, D_i)$, T_i and D_i are fixed and the WCET $x_i \in \mathbb{R}^+$ is a variable. We use the values given by George and Hermant [GH09b]:

- $\tau_1(x_1, T_1, D_1) = \tau_1(x_1, 7, 5)$,
- $\tau_2(x_2, T_2, D_2) = \tau_2(x_2, 11, 7)$,
- $\tau_3(x_3, T_3, D_3) = \tau_3(x_3, 13, 10)$.

In this example, we have the least common multiple of periods $P = 1001$. From Theorem 2.1, we have to consider the $s = 281$ elements in the set S given by:

$$\begin{aligned} S = &\{5 + 7k_1, k_1 \in \{0, \ldots, 142\}\} \cup \\ &\{7 + 11k_2, k_2 \in \{0, \ldots, 90\}\} \cup \\ &\{10 + 13k_3, k_3 \in \{0, \ldots, 76\}\} \end{aligned}$$

The simplex algorithm is applied on the LPP 2.1. We obtain the following set S after removing all the unnecessary constraints:

$$S = \{5, 7, 10, 12, 40\}$$

From Theorem 2.1, we therefore have:

$$\begin{aligned} Load\left(\tau_{(X,T,D)}\right) &= \max\left(U_{\tau_{(X,T,D)}}, sup_{t \in S} \frac{DBF\left(\tau_{(X,T,D)}, t\right)}{t}\right) \\ &= \max\left(\frac{x_1}{7} + \frac{x_2}{11} + \frac{x_3}{13}, \frac{x_1}{5}, \frac{x_1 + x_2}{7}, \frac{x_1 + x_2 + x_3}{10}, \right. \\ &\left. \frac{2x_1 + x_2 + x_3}{12}, \frac{6x_1 + 4x_2 + 3x_3}{40}\right) \end{aligned}$$

2.4.3.1.4 Useful properties of the *Load* function Finally, we list some useful properties of the *Load* function that we use in our work, especially in Section 3.3. Property 2.1 and Property 2.3 are originally expressed by George and Hermant [GH09a].

Property 2.1 shows that once we have computed the *Load* for a task set $\tau_{(C,T,D)}$, it is straightforward to compute the *Load* for the same task set where all the WCETs are multiplied by a real number $\alpha \geqslant 0$.

Property 2.1 ([GH09a])**.**
Let $\tau_{(C,T,D)}$ be a sequential sporadic task set.

$$Load\left(\tau_{(\alpha C,T,D)}\right) = \alpha Load\left(\tau_{(C,T,D)}\right)$$ ■

Property 2.2 shows that the *Load* of the union of two task sets is at most equal to the sum of the *Load* of each task set.

Property 2.2 ([GH09a])**.**
Let $\tau_{(C,T,D)}$ and $\tau'_{(C',T',D')}$ be two sporadic sequential task sets.

$$Load\left(\tau_{(C,T,D)} \cup \tau'_{(C',T',D')}\right) \leqslant Load\left(\tau_{(C,T,D)}\right) + Load\left(\tau'_{(C',T',D')}\right)$$ ■

Property 2.3 shows that the *Load* corresponding to a transformed task set where all the tasks have their period multiplied by a real number $\alpha \geqslant 0$ is equal to the *Load* corresponding to a transformed task set where all the tasks have their WCET and their deadline divided by the same value α.

Property 2.3.
Let $\tau_{(C,T,D)}$ be a sporadic sequential task set.

$$Load\left(\tau_{(C,\alpha T,D)}\right) = Load\left(\tau_{(C/\alpha,T,D/\alpha)}\right)$$ ■

Proof. From the definition of the *Load* function we have:

$$\begin{aligned} Load\left(\tau_{(C,\alpha T,D)}\right) &= sup_{t>0}\frac{DBF\left(\tau_{(C,\alpha T,D)}, t\right)}{t} \\ &= sup_{t>0}\frac{\sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{t-D_i}{\alpha T_i} \right\rfloor\right) \times C_i}{t} \end{aligned}$$

Hence, we can write:

$$Load\left(\tau_{(C,\alpha T,D)}\right) = sup_{t>0}\frac{\sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{\frac{t}{\alpha} - \frac{D_i}{\alpha}}{T_i} \right\rfloor\right) \times C_i}{t}$$

Finally, let $t' = {}^{t}/_{\alpha}$, changing t to t' leads to:

$$\begin{aligned} Load\left(\tau_{(C,\alpha T,D)}\right) &= sup_{t'>0}\frac{\sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{t' - \frac{D_i}{\alpha}}{T_i} \right\rfloor\right) \times \frac{C_i}{\alpha}}{t'} \\ &= Load\left(\tau_{(C/\alpha,T,D/\alpha)}\right) \end{aligned}$$

□

Let us take an example to verify Property 2.3. We choose a platform composed of $m = 2$ processors and generate the task set $\tau_{(C,T,D)} = \{\tau_1, \tau_2, \tau_3\}$ as following:

- $\tau_1(C_1, T_1, D_1) = \tau_1(x_1, 70, 60)$,
- $\tau_2(C_2, T_2, D_2) = \tau_2(x_2, 110, 72)$,
- $\tau_3(C_3, T_3, D_3) = \tau_3(x_3, 130, 84)$.

We then can generate task set $\tau_{(C,2T,D)}$ equivalent to $\tau_{(C,T,D)}$ with period of each task multiplied by 2. Thus, $\tau_{(C,2T,D)} = \left\{\tau_1^{2T}, \tau_2^{2T}, \tau_3^{2T}\right\}$ is composed of:

- $\tau_1^{2T}(x_1, 2T_1, D_1) = \tau_1^{2T}(x_1, 2 \times 70, 60) = \tau_1^{2T}(x_1, 140, 60)$,
- $\tau_2^{2T}(x_2, 2T_2, D_2) = \tau_2^{2T}(x_2, 2 \times 110, 72) = \tau_2^{2T}(x_2, 220, 72)$,
- $\tau_3^{2T}(x_3, 2T_3, D_3) = \tau_3^{2T}(x_3, 2 \times 130, 84) = \tau_3^{2T}(x_3, 260, 84)$.

Finally, we can generate task set $\tau_{(C/2,T,D/2)}$ equivalent to $\tau_{(C,T,D)}$ with deadline and WCET of each task divided by 2. Thus, $\tau_{(C/2,T,D/2)} = \left\{\tau_1^{D/2}, \tau_2^{D/2}, \tau_3^{D/2}\right\}$ is composed of:

- $\tau_1^{D/2}(\frac{x_1}{2}, T_1, \frac{D_1}{2}) = \tau_1^{D/2}(\frac{x_1}{2}, 70, \frac{60}{2}) = \tau_1^{D/2}(\frac{x_1}{2}, 70, 30)$,
- $\tau_2^{D/2}(\frac{x_2}{2}, T_2, \frac{D_2}{2}) = \tau_2^{D/2}(\frac{x_2}{2}, 110, \frac{72}{2}) = \tau_2^{D/2}(\frac{x_2}{2}, 110, 36)$,
- $\tau_3^{D/2}(\frac{x_3}{2}, T_3, \frac{D_3}{2}) = \tau_3^{D/2}(\frac{x_3}{2}, 130, \frac{84}{2}) = \tau_3^{D/2}(\frac{x_3}{2}, 130, 42)$.

Table 2.1 shows the results of the simplex algorithm applied to LPP 2.1 for the three previously defined task sets. To verify Property 2.3, we have to do a numeric application, we choose $x_1 = 20$, $x_2 = 48$ and $x_3 = 36$. As expected, according to Table 2.1, we have $Load(\tau_{(C,2T,D)}) = Load(\tau_{(C/2,T,D/2)})$ as:

$$\begin{aligned} Load\left(\tau_{(C,2T,D)}\right) &= \max\left(\frac{x_1}{140} + \frac{x_2}{220} + \frac{x_3}{260}, \frac{x_1}{60}, \frac{x_1 + x_2}{72}, \frac{x_1 + x_2 + x_3}{84}\right) \\ &= \frac{104}{84} = \frac{26}{21} \\ Load\left(\tau_{(C/2,T,D/2)}\right) &= \max\left(\frac{\frac{x_1}{2}}{70} + \frac{\frac{x_2}{2}}{110} + \frac{\frac{x_3}{2}}{130}, \frac{\frac{x_1}{2}}{30}, \frac{\frac{x_1}{2} + \frac{x_2}{2}}{36}, \frac{\frac{x_1}{2} + \frac{x_2}{2} + \frac{x_3}{2}}{42}\right) \\ &= \frac{52}{42} = \frac{26}{21} \end{aligned}$$

Task Set	Number of time instants within $[0; P]$	With LPP 2.1	
		Number of elements	List of elements
$\tau_{(C,T,D)}$	311	7	$60, 72, 84, 130, 200, 410, 622$
$\tau_{(C,2T,D)}$	311	3	$60, 72, 84$
$\tau_{(C/2,T,D/2)}$	311	3	$30, 36, 42$

Table 2.1 – Example using LPP 2.1 to verify Property 2.3

2.4.4 Allowance margin of task parameters

When a task set is scheduled on a processor set, it could be interesting to know the available margin for task parameters such that the task set is schedulable. For example, if a processor suffers from a failure that slows it, the running task may execute longer than expected. The margin of execution would give the additional time during which it can execute without compromising the schedulability of the system. We propose to give the results for the allowance of WCET in Subsection 2.4.4.1 and the allowance of deadline in Subsection 2.4.4.2 for pre-emptive EDF scheduler. These results will be used in Section 3.3.

2.4.4.1 Allowance of WCET for pre-emptive EDF scheduler

Bougueroua, George, and Midonnet [BGM07] proposed Theorem 2.3 which gives the allowance of WCET for a task with A-Deadline scheduled with pre-emptive EDF scheduler. It is the maximum value A_i such that the task set $\tau_i(C_i + A_i, T_i, D_i) \cup \left\{\cup_{j=1,j\neq i}^{n} \tau_j\right\}$ is schedulable with pre-emptive EDF scheduler, assuming that $\cup_{j=1,j\neq i}^{n} \tau_j$ is schedulable with pre-emptive EDF scheduler.

Theorem 2.3 (Allowance of WCET for pre-emptive EDF scheduler)**.**
Let $\tau = \{\tau_1, \ldots, \tau_n\}$ be a sporadic sequential task set composed of n tasks. Let $\tau \setminus \tau_i$ be the task set composed of the tasks in τ excluding task τ_i. If $\tau \setminus \tau_i$ is schedulable with EDF scheduler, the maximum allowance of WCET A_i of the task τ_i is given by Equation 2.21.

$$A_i \stackrel{\text{def}}{=} \min\left(\min_{t \geqslant D_i}\left(\frac{t}{1 + \left\lfloor \frac{t-D_i}{T_i} \right\rfloor} \times \left(1 - \frac{DBF(\tau, t)}{t}\right)\right), (1 - U_\tau) \times T_i\right) \tag{2.21}$$

■

Proof. The allowance of task τ_i must satisfy two conditions:

(i) $\forall t \geqslant 0,\ DBF(\tau, t) + \left(1 + \left\lfloor \frac{t-D_i}{T_i} \right\rfloor\right) \times A_i \leqslant t$

This leads to: $\forall t \geqslant 0,\ A_i \leqslant \frac{t}{1+\left\lfloor \frac{t-D_i}{T_i} \right\rfloor} \times \left(1 - \frac{DBF(\tau,t)}{t}\right)$

It follows that $A_i \leqslant \min_{t \geqslant 0}\left(\frac{t}{1+\left\lfloor \frac{t-D_i}{T_i} \right\rfloor} \times \left(1 - \frac{DBF(\tau,t)}{t}\right)\right)$.

(ii) $U + \frac{A_i}{T_i} \leqslant 1$

This leads to: $A_i \leqslant (1 - U_\tau) \times T_i$.

A_i is thus the minimum value satisfying both conditions.

□

Theorem 2.3 is an adaptation of a result presented by Balbastre, Ripoll, and Crespo [BRC02] which stated that $A_i = \min_{t \geqslant 0} \left(\frac{t}{1+\left\lfloor \frac{t-D_i}{T_i} \right\rfloor} \left(1 - \frac{DBF(\tau,t)}{t}\right) \right)$. But if we consider, for example, a task set composed of only one task defined by $\tau_1(C_1, T_1, D_1) = \tau_1(20, 100, 120)$, we have $A_1 = 100$ whose maximum value is obtained for $t = 120$. Nevertheless, in that case, $U_{\tau_1(C_1+A_1,T_1,D_1)} = {}^{(C_1+A_1)}/_{T_1} = {}^{120}/_{100} > 1$. Hence, the computation of A_i given by Balbastre, Ripoll, and Crespo [BRC02] is not valid for tasks with A-Deadlines.

2.4.4.2 Allowance of deadline for pre-emptive EDF scheduler

Balbastre, Ripoll, and Crespo [BRC06] already studied the computation of minimum acceptable deadline of a task scheduled with pre-emptive EDF scheduler. We present a modified version that addresses some problems identified in their algorithm that we detail in the following.

We present some explanations and counter examples showing our corrections on the algorithm given by Balbastre, Ripoll, and Crespo [BRC06]. In Algorithm 1, lines 7 to 11 have been added and line 6 has been modified with respect to the original algorithm.

Error with $t = l \times T_i + D_i$ The algorithm proposed by Balbastre, Ripoll, and Crespo [BRC06] does not appear to include the special case where we have $DBF(\tau, t) = t$ at time instant $t = l \times T_i + D_i$, the absolute deadline of considered task τ_i, with l being a positive integer. Here, the absolute deadline of τ_i should not be reduced and should be kept equal to D_i. For this specific case, the original algorithm does not seem to be perfectly clear since we do not successfully know if these time instants have to be considered or not. However, in both cases, we show with some examples that the condition is not correct. If we consider those time instants, the original algorithm will give task τ_i a deadline equal to $DBF(\tau, t) + C_i - l \times T_i = D_i + C_i > D_i$ leading to a higher deadline than D_i, not the minimum. If we do not consider those time instants, the original algorithm can provide deadlines that are too small. We have corrected the algorithm by adding lines 7 to 11.

Let us consider the example composed of three tasks $\tau = \{\tau_1, \tau_2, \tau_3\}$ with:

- $\tau_1(C_1, T_1, D_1) = \tau_1(10, 54, 16)$,
- $\tau_2(C_2, T_2, D_2) = \tau_2(12, 97, 91)$,

Algorithm 1: Minimum deadline computation for pre-emptive EDF scheduler

input : A task set τ, a task τ_i in τ
output : The minimum acceptable deadline $D_{i,min}$ of τ_i when τ is scheduled with EDF scheduler
Data: k, l are integers, t and *deadline* are variables and P is the least common multiple of periods of tasks in τ

```
1  deadline ← 0;
2  k ← ⌈P/T_i⌉;
3  D_{i,min} ← 0;
4  for l = 0 to k − 1 do
5      t ← l × T_i + D_i;
6      deadline ← max(C_i, DBF(τ, l × T_i + C_i) + C_i − l × T_i);
7      if t = DBF(τ, t) then
8          D_{i,min} ← D_i;
9          exit the for-loop;
10     else
11         t ← t − 1;
12         while t > l × T_i + C_i do
13             if t ∈ [0; P] is an absolute deadline of a task τ_j ∈ τ and
14             t − DBF(τ, t) < C_i then
15                 deadline ← DBF(τ, t) + C_i − l × T_i;
16                 exit the while-loop;
17             end if
18             t ← t − 1;
19         end while
20     end if
21     D_{i,min} ← max(D_{i,min}, deadline);
22 end for
23 return D_{i,min} ;
```

- $\tau_3(C_3, T_3, D_3) = \tau_3(44, 88, 54)$.

If we compute the minimum acceptable deadline of task τ_3 with the original algorithm we have:

1. For the first iteration, we initialize $D_{3,min} = C_3 = 44$.
2. We search for absolute deadlines within the interval $[C_3; D_3] = [44; 4]$. The only time instant to consider is $t = 54$ which is the deadline of task τ_3.
3. At time instant $t = 54$:

- if we consider this time instant, the new minimum deadline will be equal to $D_{3,min} = DBF(\tau, 44) + C_3 - 0 \times T_3 = 54 + 44 + 0 = 98$ which is higher than the actual deadline $D_3 = 54$.
- if we do not consider this time instant, the final minimum deadline will remain equal to the initial value $D_{3,min} = C_3 = 44$ which leads to an unschedulable task set with a *Load* equal to 1.2272.

With our modifications of lines 7 to 11, we find that $DBF(\tau, 54) = 54$, thus $DBF(\tau, D_3) = D_3$ and we fix the minimum possible deadline to $D_{3,min} = D_3 = 54$. Any reduction of this deadline will lead to a *Load* larger than 1.

Error with the initialization $deadline = C_i$ In the original algorithm, the variable *deadline* is initialized as $deadline = C_i$. At line 6 of Algorithm 1, we have replaced this initialization with $deadline = \max(C_i, DBF(\tau, l \times T_i + C_i) + C_i - l \times T_i)$.

Let us consider the example composed of three tasks $\tau = \{\tau_1, \tau_2, \tau_3\}$ with:

- $\tau_1(C_1, T_1, D_1) = \tau_1(10, 55, 16)$,
- $\tau_2(C_2, T_2, D_2) = \tau_2(12, 88, 80)$,
- $\tau_3(C_3, T_3, D_3) = \tau_3(44, 88, 80)$.

Computing the minimum acceptable deadline for task τ_3 with the original algorithm leads to the following:

1. For the first iteration, we initialize $D_{3,min} = 44$.
2. We search for absolute deadlines within the interval $[C_3; D_3] = [44; 80]$. We have to consider time instants $t = 80$, deadline of tasks τ_2 and τ_3 and $t = 71$, the second deadline of task τ_1.
3. At time instant $t = 80$:
   - if we consider this time instant, the new minimum deadline of τ_3 will be equal to $D_{3,min} = DBF(\tau, 80) + C_3 - 0 \times T_3 = 76 + 44 + 0 = 120$ which is higher than the actual deadline $D_3 = 80$.
   - if we do not consider this time instant, the minimum deadline remains equal to the initial value $D_{3,min} = C_3 = 44$ which leads to an unschedulable task set with a *Load* equal to 1.2272.
4. At time instant $t = 71$:
   - we have $DBF(\tau, 71) = 20$ and $t - DBF(\tau, t) = 71 - DBF(\tau, 71) > C_3 = 44$ thus this time instant is ignored and the minimum acceptable deadline for task τ_3 remains equal to 44 or 120 according to the previous point.

With our modification of line 6, we will initialize $D_{3,min} = \max(C_3, DBF(\tau, 0 \times T_3 + C_3) + C_3 - 0 \times T_3) = \max(44, DBF(\tau, 44) + 44) = \max(44, 10 + 44) = 54$. Any reduction of this deadline will lead to a *Load* larger than 1.

2.5 Scheduling on multiprocessor platforms

This section is dedicated to the presentation of the basics and state-of-the-art for RT scheduling on multiprocessor platforms. In Subsection 2.5.1, we propose an overview of the results for scheduling S-Tasks. Subsection 2.5.2 gives important results for scheduling P-Tasks.

2.5.1 Scheduling Sequential Tasks (S-Tasks)

Multiprocessor scheduling of S-Tasks is an active area of research that has mostly been studied with **P**artitioned **Scheduling** (P-Scheduling) approach. In the P-Scheduling case, tasks are assigned to the processors according to a placement heuristic and cannot migrate. A classical uniprocessor scheduling schedulability condition is then used to decide on the schedulability of the tasks. Subsection 2.5.1.1 presents this approach.

Another approach called **G**lobal **Scheduling** (G-Scheduling) is considered to have theoretically better performances in terms of successfully scheduled task sets compared to P-Scheduling approach. With G-Scheduling, jobs are allowed to migrate and processor utilization can reach 100% [BGP95; Bar+96; CRJ06]. Recent advances in multiprocessor technology have reduced migration cost, increasing the interest in such scheduling and making G-Scheduling an attractive solution. However, migration cost is not taken into account in current schedulability conditions. Subsection 2.5.1.2 presents this approach.

More recently, **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approach has been proposed. This approach can be seen as an intermediate solution between P-Scheduling and G-Scheduling approaches. In classical SP-Scheduling, the goal is to hold back the number of job migrations in order to reduce runtime overheads. The basic idea with SP-Scheduling is to execute tasks according to a static job migration pattern. Most results propose heuristics that first try to assign, as much as possible, tasks to a single processor according to a particular P-Scheduling approach. The jobs of the tasks that cannot be assigned to a single processor are then allowed to migrate between a set of fixed particular processors. Subsection 2.5.1.3 presents this approach.

2.5.1.1 Partitioned Scheduling (P-Scheduling)

Partitioned **Scheduling** (P-Scheduling) is attractive as it does not lead to job migration costs that can influence the schedulability of the system. However, it

can be shown that in some pathological task configuration, schedulability tests can only ensure the schedulability of a system with a system utilization less than 50% [LDG04; KYI09]. This is an indication of the pessimism of P-Scheduling.

Definition 2.15 (**P**artitioned **Scheduling** (P-Scheduling))**.**
P-Scheduling refers to a multiprocessor scheduling approach which consists in assigning each task to only one processor. After this assignment, tasks never migrate and each couple (*task subset*; *processor*) can be seen as an independent uniprocessor scheduling problem. Hence, given a task set τ and a processor set π composed of m processors, with P-Scheduling approach, τ is divided into a number of disjoint subsets less than or equal to m. Each of these subsets is assigned to one processor. Uniprocessor scheduling policies are then used locally on each processor. ■

The main advantage of this approach is to break up the problem with multiple processors to multiple well-known problems, each containing only one processor.

The main disadvantage of this approach is that assigning tasks to processors is equivalent to a Bin-Packing problem: how to place n objects of different sizes in m boxes such that the physical constraints of the objects and the boxes are met. This problem is known to be NP-hard in the strong sense [Joh74]. One way to find an optimal solution for this kind of problem is to enumerate all possible configurations and verify their correctness one by one which can be a time consuming process. We can reduce the complexity by seeking sub-optimal solution with placement heuristics. Therefore, with the P-Scheduling approach, we need to find a placement heuristic to assign tasks to processors and then to use a uniprocessor schedulability test on each processor to decide on the schedulability of the tasks assigned to it. We have extract different placement heuristics from the state-of-the-art. These include First-Fit, Next-Fit, Best-Fit and Worst-Fit [LDG04; GH09a]. First-Fit placement heuristic has received more attention.

- First-Fit: tasks are allocated sequentially, one by one to the first processor it fits into (according to a schedulability test). The process always starts from processor π_1 up to processor π_m.
- Next-Fit: tasks are allocated sequentially, one by one to the first processor it fits into (according to a schedulability test). The process always starts from the last processor where a task has been assigned (the first processor for the first task).
- Best-Fit: tasks are allocated sequentially but a task is assigned to the processor it fits best so that it will minimize the remaining processor capacity (for example the remaining utilization).

- Worst-Fit: the same as Best-Fit except that the goal is to maximize the remaining processor capacity.

A variant of these placement heuristics is first to sort the task set to be assigned, for example in decreasing order of task density, leading to First-Fit Decreasing or Best-Fit Decreasing variant. Baruah and Fisher [BF05] proposed a feasibility S-Test and demonstrate that the First-Fit Decreasing placement heuristic successfully partitions any sporadic sequential task set τ on $m \geqslant 2$ identical processors if τ and m satisfy Equation 2.22.

$$\Lambda_\tau \leqslant \begin{cases} m - (m-1)\max\limits_{\tau_i \in \tau}(\Lambda_{\tau_i}) & \text{if } \max\limits_{\tau_i \in \tau}(\Lambda_{\tau_i}) \leqslant \frac{1}{2} \\ \frac{m}{2} + \max\limits_{\tau_i \in \tau}(\Lambda_{\tau_i}) & \text{if } \max\limits_{\tau_i \in \tau}(\Lambda_{\tau_i}) \geqslant \frac{1}{2} \end{cases} \tag{2.22}$$

We evaluate various P-Scheduling algorithm in Section 3.2, here are some examples:

- For each task, a density computed from the task parameters is considered for the partitioning (see the results of Baker [Bak06] for an exhaustive list of density-based partitioning heuristics). For example, in conjunction with First-Fit placement heuristic and pre-emptive EDF scheduler, the schedulability S-Test $\Lambda_\tau \leqslant 1$, proposed by Liu [Liu00], is used to verify if a task can be added to a processor.

- Baruah and Fisher [BF06; BF07] proposed results for P-Scheduling based on a **D**emand **B**ound **F**unction (DBF) approximation given by Albers and Slomka [AS04] and a condition to verify if a task can be added to a processor according to EDF scheduler. This is given in Equation 2.23.

$$\forall \tau_i \in \tau, \begin{cases} D_i - DBF^*\left(\tau \backslash \{\tau_i\}, D_i\right) \geqslant C_i \\ 1 - \sum\limits_{\tau_j \in \tau, \tau_j \neq \tau_i} U_{\tau_j} \geqslant U_{\tau_i} \end{cases}$$
$$\text{with } DBF^*(\tau, t) = \begin{cases} \sum\limits_{i=1}^{n} \left(C_i + (t - D_i) \times U_{\tau_i}\right) & \text{if } t \geqslant D_i \\ 0 & \text{otherwise} \end{cases} \tag{2.23}$$

- George and Hermant [GH09b] propose a Worst-Fit Decreasing heuristic based on the *Load* function for EDF scheduler. The goal of this P-Scheduling algorithm is to maximize the remaining processor utilization characterized by the function $1 - Load(\tau^{\pi_k})$ on each processor π_k.

2.5.1.2 Global Scheduling (G-Scheduling)

Optimal strategies have been proposed for periodic S-Tasks: Baruah, Gehrke, and Plaxton [BGP95; Bar+96] introduced *Pfair* (Proportional fairness, for

discrete time), where each task is divided into quantum-size pieces denoted subtasks having pseudo deadlines, and Cho, Ravindran, and Jensen [CRJ06] introduced *LLREF* (for continuous time), with *T-Lplane* abstraction where the scheduling is done to bound the number of pre-emptions. These strategies, although optimal, can lead to a large number of migrations, thus leaving their applicability to RT systems uncertain. An active area of research aims to tackle the problem of reducing the number of job migrations, to reduce the impact of migration cost on the schedulability conditions. Bertogna [Ber09] showed that the schedulability tests proposed for **G**lobal **Scheduling** (G-Scheduling) are, in the current state-of-the-art, more pessimistic than the schedulability tests obtained for P-Scheduling. However, Baruah [Bar07] proved that G-Scheduling and P-Scheduling are incomparable: there are task sets which are schedulable by P-Scheduling approach but not by G-Scheduling approach and conversely.

Definition 2.16 (**G**lobal **Scheduling** (G-Scheduling))**.**
G-Scheduling refers to a multiprocessor scheduling approach which consists in scheduling each task on any available processor. Hence, given a task set τ and a processor set π composed of m processors, with G-Scheduling approach, at each time instant t, the m highest priority tasks are executed on the platform allowing the migration of tasks from one processor to another with the restriction that a task cannot be executed on different processors at the same instant. ■

The main advantage of this approach is to fully use the platform: if a processor is idle, you can execute a task on it without any assignment restriction.

The main disadvantage of this approach is that a migration of a task between processors has a cost which can make a feasible task set unschedulable.

However, with the evolution of processor architectures, the migration-related penalties of G-Scheduling have been reduced. Bastoni, Brandenburg, and Anderson [BBA10] have shown through some experiments that cache migration delays can be equivalent to pre-emption delays for a system under load. The evolution of $3D$ architectures presented by Coskun, Kahng, and Rosing [CKR09] also tends to reduce migration-related penalties. Here are some results for G-Scheduling approach with pre-emptive EDF scheduler:

- Goossens, Funk, and Baruah [GFB03] prove a utilization-based schedulability test called *GBF*.
- Baker [Bak03; Bak05a] offers a different approach based on an analysis of the workload. This test is similar to *GBF* for tasks with I-Deadlines but incomparable for tasks with C-Deadlines.

- Baruah [Bar07] proposes a parallel condition derived from the computation of the DBF.

- Baker and Baruah [BB09] base their schedulability test on the computation of the *Load* function. Previous schedulability S-Tests related to the *Load* function have been presented but this test is shown to dominate them.

- Bertogna, Cirinei, and Lipari [BCL05] present an iterative approach based on the slack of each task. This information is used to estimate the interfering workload in a scheduling window. Bertogna has named this test BCL.

- Bertogna and Cirinei [BC07] introduce RTA which is a schedulability test based on an iterative estimation of the WCRT of each task.

- Baruah et al. [Bar+09] focus on the DBF to derive a schedulability S-Test. This test has the smallest possible processor speed-up factor of $(2 - \frac{1}{m})$ for pre-emptive EDF scheduler.

- Bertogna [Ber09] compares the main existing results in this area. All these conditions are evaluated according to the number of task sets that are detected to be schedulable. Since these schedulability tests are incomparable in terms of task sets detected schedulable, Bertogna proposes the algorithm $COMP$ based on the sequence of the best previous techniques. According to this study, $COMP$ and RTA appear to detect the largest number of schedulable task sets.

- Megel, Sirdey, and David [MSD10] propose to express real-time constraints through linear equalities and inequalities with the objective to reduce the number of pre-emptions and migrations for periodic S-Tasks with I-Deadlines. Their linear program creates sub-jobs which are then scheduled using an algorithm named IZL. This solution is composed of an off-line part (linear program) and a runtime part (IZL algorithm) to find optimal global real-time schedules. Megel, Sirdey, and David emphasise that their approach significantly decrease the number of pre-emptions and migrations with a significant but acceptable investment in off-line computation time.

- Nelissen et al. [Nel+11; Nel+12] propose $U\text{-}EDF$, another algorithm to give optimal results but with an *unfair* approach. Indeed, the authors observe from the study of others global algorithms that the number of pre-emptions and migrations decreases as the fairness constraint is relaxed. Its optimality has been proven for pre-emptive sporadic and periodic tasks with I-Deadlines. As mention by the authors: "Contrarily to all other existing optimal multiprocessor scheduling algorithms for sporadic tasks,

U-EDF is not based on the fairness property. Instead, it extends the main principles of EDF so that it achieves optimality while benefiting from a substantial reduction in the number of pre-emptions and migrations."

- Regnier et al. [Reg+11] introduce *RUN* which is another optimal solution with the particularity to reduce the multiprocessor problem to a series of uniprocessor problems scheduled with EDF scheduler. Compared to *U-EDF*, it uses a weak version of proportional fairness and a task model composed of I-Deadlines sequential tasks with fixed-rate. Actually, they does not consider periodic tasks but tasks have a *fixed rate* and a job of a task with rate $U_{\tau_i} \leqslant 1$ requires $U_{\tau_i} \times (d - r)$ execution time, with d the absolute deadline of the job and r its activation instant. According to the authors, "*RUN* significantly outperforms existing optimal algorithms with an upper bound of $O(\log m)$ average pre-emptions per job on m processors and reduces to Partitioned EDF whenever a proper partitioning is found."

2.5.1.3 Semi-Partitioned Scheduling (SP-Scheduling)

The concept of **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) was introduced by Anderson, Bud, and Devi [ABD05] where the authors define two classes of tasks: those assigned to only one processor and those assigned to more than one processor. Tasks assigned to more than one processor are called *migrating tasks* while those assigned to only one processor are called *fixed tasks*.

Definition 2.17 (**S**emi-**P**artitioned **Scheduling** (SP-Scheduling))**.**
SP-Scheduling refers to a multiprocessor scheduling approach which consists in assigning some tasks to only one processor (fixed tasks) and others to multiple processors (migrating tasks). After this assignment, the jobs of fixed tasks never migrate while the jobs of migrating tasks can use different processors. ■

The main advantage of this approach is to reduce the number of migrations compared to a G-Scheduling approach while relaxing the assignment constraint introduced by a P-Scheduling approach.

The main disadvantage of this approach is that we also have the disadvantages of the others approaches: assigning tasks to processors is equivalent to the Bin-Packing problem which is known to be NP-hard, and we have introduced migrations which can lead to additional execution costs. Moreover, the scheduling on each processor is no longer independent.

Anderson, Bud, and Devi [ABD05] also define the degree of migration allowed by an algorithm:

1. No migration (i.e., task partitioning).

2. Migration allowed, but only at job boundaries (i.e., migration at the job level). A job is executed on one processor but successive jobs of a task can be executed on different processors. This degree of migration is called **Rest**ricted **Migration** (Rest-Migration): only tasks are allowed to migrate, job migration is forbidden.

3. Migration allowed and not restricted to be at job boundaries, for example a job can be portioned between multiple processors (i.e., jobs are also allowed to migrate during their execution). This degree of migration is called **UnRest**ricted **Migration** (UnRest-Migration): jobs are also allowed to migrate. Notice that "unrestricted" does not means that the migration points cannot be fixed, but, if they are fixed, they are not restricted to be at job boundaries.

They also proposed $EDF\text{-}fm$ which belongs to the second category. It splits jobs between two processors allocating r jobs over s to a processor with the index p, and the others jobs ($s - r$ over s) to a processor with the index $p + 1$. The number of migrations is reduced and the total utilization of this task can be adapted on each processor. However, $EDF\text{-}fm$ is best suited to soft RT systems since it cannot guarantee the deadlines of fixed tasks. Dorin et al. [Dor+10] also proposed an algorithm with Rest-Migrations but they designed their algorithm to handle hard RT task sets composed of sporadic S-Tasks with C-Deadlines. Their algorithm first assigns as much tasks as possible with a P-Scheduling algorithm, then jobs of remaining tasks are assigned to processors by using a cyclic job assignment algorithm. Dorin et al. developed a schedulability analysis based on an extension of the DBF function to assure the schedulability of the tasks and jobs assigned to each processor.

In terms of migrations, the following algorithms are classified in the third category (UnRest-Migration). They split tasks according to their WCET between two or more processors. Parts of the migratory job are executed on separate processors but the simultaneity of the execution is not allowed.

Anderson, Bud, and Devi [ABD05] lay the foundations for the assignment of tasks on processors. The principle is to fill each processor sequentially. If the remaining capacity of a processor with index p is not large enough to receive the entire task, this task is split into two parts. The first part is assigned to fill processor p and the second part is assigned to processor $p + 1$. Thus, there are at most two migratory tasks on each processor and $m - 1$ migratory tasks in the whole system. This technique is similar to a Next-Fit placement heuristic with task splitting. All the following algorithms up to $EDHS$ [KY08c] use this assignment.

- Andersson and Tovar [AT06] propose EKG which offers a complex but optimal solution to this problem. According to a parameter K which defines

the size of each group of processors accepting migratory tasks, EKG is able to adapt the utilization bound and the number of migrations. Although when $K = m$, EKG is optimal with an utilization bound of 100%, it incurs more migrations.

- Kato and Yamasaki [KY07] introduce $EDDHP$ (originally named $Ehd2\text{-}SIP$) which reduces the number of migrations and increases the success ratio with regard to a P-Scheduling algorithm. $EDDHP$ is outperformed by EKG in terms of its success ratio but is more convenient to implement and to use in practical cases.

- Kato and Yamasaki [KY07] introduce the notion of *portion* and named their algorithms *portioned scheduling*: "In portioning, the task is not really divided into two blocks, but its utilization is shared on the two processors". Furthermore, the authors propose an optimization that will be important subsequently. They may find a task set non schedulable according to their algorithm but schedulable with a simple P-Scheduling algorithm. It proves that their splitting method may degrade schedulability compared to some non-splitting methods. Thus, they optimize their algorithm to deal with this case.

- Kato and Yamasaki [KY08a] also propose $EDDP$ to improve the schedulability of $EDDHP$ by introducing some mechanisms of EKG. Indeed, these algorithms distinguish two types of tasks based on their utilization: light tasks and heavy tasks. $EDDP$ is still easier to implement than EKG and guarantees a new utilization bound of 65% with fewer migrations.

- $RMDP$ is a FTP version of $EDDHP$ presented by Kato and Yamasaki [KY08b]. The authors claim that a FTP scheduling is still widely used and it does not suffer from the domino-effect problem or the disadvantage of varying jitter in periodic execution.

- Kato and Yamasaki [KY08c] suggest fundamentally changing the assignment of tasks on processors with $EDHS$. For all previous algorithms, except for the optimization of $EDDHP$, if a task causes the total utilization of a processor to exceed its utilization bound, the WCET of this task is always split into two portions. For $EDHS$, a simple partitioning is performed before splitting the WCET of a task. If the P-Scheduling approach fails, the remaining WCET portions are shared on two or more processors. Each part of the task is defined in order to fill a processor. Kato and Yamasaki [KY08c] chose to attribute at most one migrating task to each processor. A task always migrates in the same way, between the same processors and at the same time instant of their execution. Here, the notion of SP-Scheduling takes its full meaning.

- *DM-PM* is a FTP version of *EDHS* given by Kato and Yamasaki [KY09]. If tasks are sorted by decreasing deadlines before assignment, migratory tasks naturally have a higher priority than fixed tasks. The scheduling of migratory tasks is thus easier.

- With *EDF-WM*, Kato, Yamasaki, and Ishikawa [KYI09] try to adapt the simplification introduce in *DM-PM* to DTP scheduling. Thus, a task is split according to its WCET but its deadline is also portioned into local deadlines used on each processor executing the task. This defines a window during which a subtask should be executed. The local deadline of a task $\tau_i(C_i, T_i, D_i)$ is equal to D_i/s (fair local deadline) where s is the number of processors executing the task. WCETs are chosen to fill the processor with respect to the fair local deadline. Schedulability analysis and complexity of the scheduler are improved with this technique. The implementation is also easier if we consider subtasks as independent tasks with a delayed activation instant.

- Andersson, Bletsas, and Baruah [ABB08] introduce the algorithm *EDF-SS*($DMIN/\delta$). The basic idea of this algorithm is to split tasks that cannot be scheduled on only one processor, between two processors. The WCETs of those tasks are divided into slots of length equal to $DTMIN/\delta$ where $DTMIN$ is the minimum of all deadlines and periods and δ is an integer parameter that is configurable. The smaller the value, and the smaller the slot size, the more migrations. The slots reserved for a task on any two different processors are synchronized in time. Tasks that are split have a higher priority than tasks executed on a single processor. This approach was first considered in the case of tasks with I-Deadlines by Andersson and Bletsas [AB08].

- Lakshmanan, Rajkumar, and Lehoczky [LRL09] introduce the algorithm *PDMS_HPTS_DS* based on Partitioned Deadline-Monotonic Scheduling (*PDMS*) with the Highest Priority Task Split (*HPTS*) heuristic. With this approach the task having the highest priority on a processor that cannot be executed on a single processor is split on two processors. Tasks are allocated in the Decreasing order of size. The authors assign local deadlines for a task τ_i (highest priority) equal to D_i on the first processor executing τ_i and $D_i - C_i^{(first)}$ on the second processor, where $C_i^{(first)}$ is the WCET of τ_i on the first processor, also equal to its WCRT. They show that *PDMS_HPTS_DS* achieves an utilization bound of 60%.

- Burns et al. [Bur+10] propose a new task-splitting $C{=}D$ scheme tested with EDF scheduler. They try to limit the number of subtasks and reduce the number of migrations by splitting at most $m - 1$ tasks. In this end, the first part of a split task is constrained to have a deadline equal to

its computation time. It therefore occupies its processor for a minimum interval. The second part of the task then has the maximum time available to complete its execution on a different processor.

We can conclude from this state-of-the-art of SP-Scheduling approach that the tendency is to find an algorithm able to schedule more task sets than P-Scheduling approach with fewer migrations than G-Scheduling approach. The complexity of the implementation is also a point to consider. $EDF\text{-}fm$ is based on migrations at job boundaries which leads to a simple implementation but the version proposed is only designed to soft RT scheduling. Other algorithms presented in this study focus on UnRest-Migration and split tasks into subtasks of execution time based on the WCETs of the tasks. This leads to optimal algorithm (EKG) but this solution is quite difficult to implement. With suboptimal algorithms, Kato et al. were able to achieve easier algorithms with reasonable utilization bounds and fewer migrations.

However those approaches require using an operating system that keeps track of job execution consumption in order to migrate a job when it has been executed. Many operating systems offer execution overrun timers to specify that a job has been executed for a given duration (e.g. AUTOSAR OS [Hla+07] or **R**eal-**T**ime **S**pecification for **J**ava (RTSJ) [BGM07]). Nevertheless, the migration time instant is not necessarily identical to the time instant at which an execution overrun occurs. This might introduce time overhead in the management of those timers to adapt them to migrate tasks. Notice that the approaches using local deadlines ($EDF\text{-}WM$, $C{=}D$, etc.) can overcome the problem since migrations occur at an offset time from the release of the task. Moreover, a migration during the execution of a job requires transferring the execution context between processors. Again, there is a solution thanks to the spread of multi-core processors that tends to eliminate this additional cost, but it remains complex and costly to use this method on a multiprocessor.

2.5.2 Scheduling Parallel Tasks (P-Tasks)

The state-of-the-art concerning the scheduling of hard RT and parallel recurring tasks is scarce. Here, we report some models of parallel tasks and some results (schedulers and schedulability/feasibility tests).

- Manimaran, Murthy, and Ramamritham [MMR98] consider the non-pre-emptive EDF scheduling of periodic parallel tasks for a task model from Gang class.

- Han and Park [HP06] consider the scheduling of a (finite) set of RT jobs allowing job parallelism.

- Collette, Cucu-Grosjean, and Goossens [CCGG08] provide a task model from Gang class which integrates job parallelism and uses malleable tasks. Malleable task model allows, at runtime, a *variable* number of threads for each task. They proved that the time-complexity of the feasibility problem of these systems is linear relative to the number of sporadic tasks.

- Lakshmanan, Kato, and Rajkumar [LKR10] consider the Fork-Join task model from Multi-Thread class. They provide a P-Scheduling algorithm and a competitive analysis for EDF and the Fork-Join task model.

- Saifullah et al. [Sai+11] proposed a "Generalized Parallel" task model from Multi-Thread class. In this task model, a periodic task is defined by a sequence of segments, each one composed of several threads. They defined a decomposition of their parallel tasks into a set of sequential tasks.

- Regarding the schedulability of recurrent RT tasks, to the best of our knowledge, we can only report results about the Gang scheduling. Kato and Ishikawa [KI09] consider the Gang EDF scheduling and provide a schedulability S-Test. Goossens and Berten [GB10] study Gang FTP scheduling and provide a schedulability NS-Test for periodic tasks.

2.6 Summary

In this chapter we introduced important models (processors and sequential or parallel tasks) and definitions. We also summarized the basics results for feasibility and schedulability analysis on uniprocessor platforms. Finally, we presented some results for the multiprocessor platform case which have motivated our work. In Chapter 3, we propose to study **S**equential **Tasks** (S-Tasks) and give results for the **P**artitioned **Scheduling** (P-Scheduling) and the **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approaches. In Chapter 4 we study **P**arallel **Tasks** (P-Tasks) and we propose a new generic parallel task model which can be adapted from a Fork-Join task model. We also propose some results for the Gang task model with a semi-clairvoyant scheduler.

Part II

Scheduling on multiprocessors platforms

Chapter 3

Scheduling Sequential Tasks (S-Tasks)

Troisième principe pour rester zen, le principe de Yunmen : "Quand tu marches, marche, quand tu es assis, sois assis. Surtout, n'hésites pas." L'autre jour, aux toilettes, je me suis surpris en train de me brosser les dents tout en répondant au téléphone. Selon le principe de Yunmen, il y avait au moins deux choses en trop.

Third principle to remain zen, the principle of Yunmen: "When you walk, walk, when you sit, be seated. Above all, do not hesitate." The other day, in the bathroom, I surprise myself by brushing my teeth while answering the phone. According to the principle of Yunmen, there were at least two things too many.

Alexandre Jollien [Jol12]

Contents

3.1 Introduction

In this chapter, we present our contributions to the **R**eal-**T**ime (RT) scheduling of **S**equential **Tasks** (S-Tasks) upon identical multiprocessor platform. Thus, we focus on the task model given in Subsection 2.2.2.2 by Definition 2.4 and Definition 2.5 on page 13. Based on the state-of-the-art presented in Subsection 2.5.1, we divided this chapter in two sections. In Section 3.2 we introduce a generalized algorithm for the **P**artitioned **Scheduling** (P-Scheduling) approach. Finally, Section 3.3 gathers our results on the **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approach.

3.2 Partitioned Scheduling (P-Scheduling)

3.2.1 Introduction

As previously stated in Subsection 2.5.1.1, the P-Scheduling approach is one of the first approaches used to schedule tasks on a multiprocessor platform. Its principle is simple to understand, it consists in breaking up the problem on multiple processors to multiple problems of only one processor. To this end, we split the task set to be scheduled into at most as many task subsets as there are processors available. Then each of these task subsets is assigned to a single processor and it can be seen as an independent scheduling problem. The challenge is therefore to find a way to partition the task set such that each task subset is schedulable. As this problem has been proven NP-hard in the strong sense [Joh74], placement heuristics have been proposed in an attempt to provide tractable solutions. Notice that the optimal partitioning in our context of identical multiprocessor platform is discussed in Subsection 3.2.2.2.

In this section, we propose a generalized P-Scheduling algorithm that adapts to the problem constraints (fixed or scalable number of processors, constrained time to find a partition of the task set etc.) and objectives (minimizing the number of processors, increased robustness to **W**orst **C**ase **E**xecution **T**ime (WCET) overruns, higher probability to find a solution etc.). We first detail our generic algorithm and we analyse each of its parameters and their influence on the final partitioning.

3.2.2 Generalized P-Scheduling algorithm

The state-of-the-art reveals that previously proposed P-Scheduling algorithms are composed of a placement heuristic, a uniprocessor schedulability test and, very often, a task sorting criterion. Indeed, as shown in Figure 3.1, a non-optimal P-Scheduling algorithm must answer three specific questions:

Q1 Which task should be considered first?

Q2 Which processor should be considered?

Q3 Is the considered task schedulable on the considered processor?

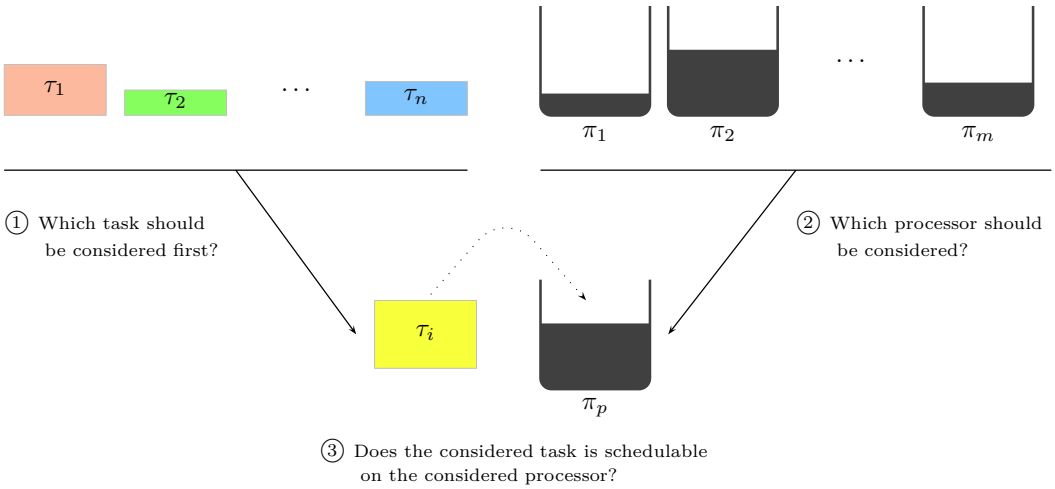


Figure 3.1 – Principle of a non-optimal P-Scheduling algorithm

Each of these questions lead to a parameter in our algorithm. A task sorting criterion allows us to select tasks in a particular order. A placement heuristic helps select candidate processors to assign the task. Finally, a uniprocessor schedulability test allows us to check on which candidate processor the task can actually be assigned. Our Generalized P-Scheduling algorithm is defined by Algorithm 2.

In the following sections we specify the interest of each parameter and we give some examples of such parameters.

3.2.2.1 Criteria for sorting tasks

This parameter responds to question Q1 for a non-optimal P-Scheduling algorithm: *Which task should be considered first?* Since we do not test each possible assignment of tasks to processors, when a task has been selected to be assigned to a processor, the decision will never be questioned again. Consequently, the order in which tasks are considered can lead to a successful partitioning or spoil everything.

Let us look at an example. A task set τ is composed of four tasks $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ with respective utilizations $U_{\tau_1} = 1/2$, $U_{\tau_2} = 1/2$, $U_{\tau_3} = 1/3$, $U_{\tau_4} = 2/3$. Notice that the total utilization of τ is $U_\tau = 2$ so, at least, two processors are necessary to schedule this task set and we take the processor set $\pi = \{\pi_1, \pi_2\}$. Consider that tasks have **I**mplicit **D**eadlines (I-Deadlines) and we use an **E**arliest **D**eadline **F**irst (EDF) scheduler on each processor, so we have to find two

Algorithm 2: Generalized P-Scheduling algorithm

input : A task set τ, a processor set π, a sorting task criterion sortTaskCriterion, a placement heuristic placHeuristic and a uniprocessor schedulability test schedTest

output : A boolean value which notify if a schedulable solution has been found and an assignment of some or all tasks of τ to a processor of π

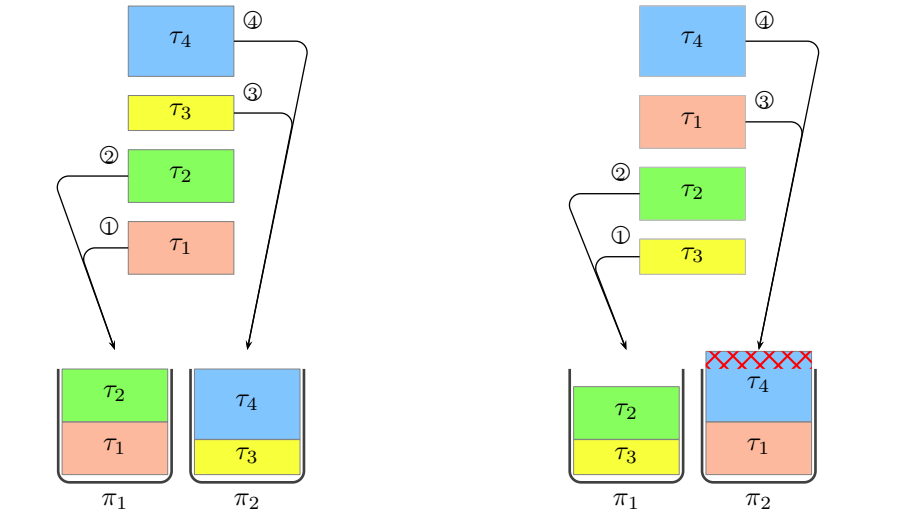
```
1  Sort tasks of τ according to sortTaskCriterion ;
2  foreach task in τ do
3      while the task is not assigned and placHeuristic gives a candidate
       processor do
4          if according to the schedTest, the task is schedulable on the
           candidate processor given by placHeuristic then
5              Assign the task to the candidate processor;
6          end if
7      end while
8  end foreach
9  if All tasks are assigned then
10     return Schedulable;
11 else
12     return unSchedulable;
13 end if
```

task subsets with a total utilization lower than or equal to 1 to ensure their schedulability. A simple partitioning solution is $\tau = \{\tau^1, \tau^2\}$ with $\tau^1 = \{\tau_1, \tau_2\}$ and $\tau^2 = \{\tau_3, \tau_4\}$. However, a P-Scheduling algorithm has not the global picture of the problem so it has to consider tasks in a particular order and assign them one by one to the processors. For this example, we examine two task orders to show the importance of sorting task criterion:

- tasks sorted by increasing ids (Figure 3.2.1): $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ will lead to assign $\tau^{\pi_1} = \{\tau_1, \tau_2\}$ to π_1 and $\tau^{\pi_2} = \{\tau_3, \tau_4\}$ to π_2 which is a working assignment.

- tasks sorted by increasing utilization (Figure 3.2.2): $\tau = \{\tau_3, \tau_2, \tau_1, \tau_4\}$ will lead to assign $\tau^{\pi_1} = \{\tau_3, \tau_2\}$ to π_1, $\tau^{\pi_2} = \{\tau_1\}$ to π_2 and leaving τ_4 unassigned. Indeed, after the assignment, the utilization of each processor is too high to accept τ_4 since $U_{\tau^{\pi_1}} = 5/6$, $U_{\tau^{\pi_2}} = 1/2$ and $U_{\tau_4} = 2/3$.

Examples of criteria for sorting tasks In the state-of-the-art, we generally find P-Scheduling algorithms with a decreasing utilization/density sorting



3.2.1: Sorted by increasing ids 3.2.2: Sorted by increasing utilization

Figure 3.2 – Importance of criteria for sorting tasks

criterion [Bak06; Bak05b], or possibly increasing relative deadline sorting criterion [FBB06b].
In our study, we decided to explore a wider range of criteria:

- Increasing/Decreasing order of *relative deadline*,
- Increasing/Decreasing order of *period*,
- Increasing/Decreasing order of *density*,
- Increasing/Decreasing order of *utilization*.

3.2.2.2 Placement

The second parameter corresponds to question Q2 for a non-optimal P-Scheduling algorithm: *Which processor should be considered?* In the optimal placement case, we should consider all the processors for each task, and keep the different solutions to choose at the end a schedulable assignment. This approach is investigated in Subsection 3.2.2.2.1. The heuristic approach corresponds to establish an order in which we consider the processors with the aim of selecting only one solution. This approach is investigated in Subsection 3.2.2.2.2.

3.2.2.2.1 Optimal placement To choose on which processor a task should be assigned, one way to find an optimal solution is to list all the possibilities. We refer to this as the optimal placement. Therefore, if we want to find a partition of a task set with n tasks on a platform with m processors, we will have to

test $\uplus_{heterogeneous} \stackrel{\text{def}}{=} m^n$ different placements. For example, if we consider two processors $\{\pi_1, \pi_2\}$ and three tasks with I-Deadlines and respective utilizations $U_{\tau_1} = 1/2$, $U_{\tau_2} = 1/3$ and $U_{\tau_3} = 2/3$, we have to test the eight different placements shown in Figure 3.3. This figure shows that placements ①, ④, ⑦ and ⑧ can not be schedulable since the total utilization on one processor exceeds 1. If we focus on the other placements, we notice a symmetry between placements ② and ③ on the one hand, and placements ⑤ and ⑥ on the other hand. If processors π_1 and π_2 are not identical, we will have to consider each of these placements. However, we study a platform with identical processors and we can reduce the number of solutions by considering each symmetric placements as equivalent. The total number of useful solution can be computed using the Stirling numbers of the second kind which count the number of ways to partition a set of n elements into m non-empty subsets [GKP88]. The Stirling numbers of the second kind are given by Equation 3.1. From this equation, we get Theorem 3.1.

$$\left\{ n \atop m \right\} \stackrel{\text{def}}{=} \sum_{j=1}^{m} (-1)^{m-j} \frac{j^{n-1}}{(j-1)!(m-j)!} \tag{3.1}$$

Theorem 3.1.
The total number of possible placements of n tasks upon an identical multiprocessor platform of m processors is given by Equation 3.2.

$$\uplus_{identical} \stackrel{\text{def}}{=} \sum_{i=1}^{\min(n,m)} \left\{ n \atop i \right\} = \sum_{i=1}^{\min(n,m)} \sum_{j=1}^{i} (-1)^{i-j} \frac{j^{n-1}}{(j-1)!(i-j)!} \tag{3.2}$$

■

Proof. The Stirling number given by Equation 3.1 allows us to compute the number of partitions of a set of n elements into m non-empty subsets. However, in order to compute the total number of possible placements of n tasks upon an identical multiprocessor platform with m processors, we also need to consider empty subsets (or empty processors) so we add to the previous value the number of ways to partition a set of n elements into $m-1$ non-empty subsets (considering 1 empty processor), then into $m-2$ (considering 2 empty processors) and so forth. Notice that the maximum number of partitions is given by $\min(n, m)$ since a task cannot be split into subtasks. □

In order to illustrate the reduction of studied placements according to Theorem 3.1, Table 3.1 gives the total number of possible placements for an heterogeneous multiprocessor platform ($\uplus_{heterogenous}$) and an identical multiprocessor platform ($\uplus_{identical}$) for a given number of processors and tasks.

3.2.2.2.2 Placement heuristics In the previous paragraph, we reminded that finding an optimal placement is a NP-hard problem. To reduce the process

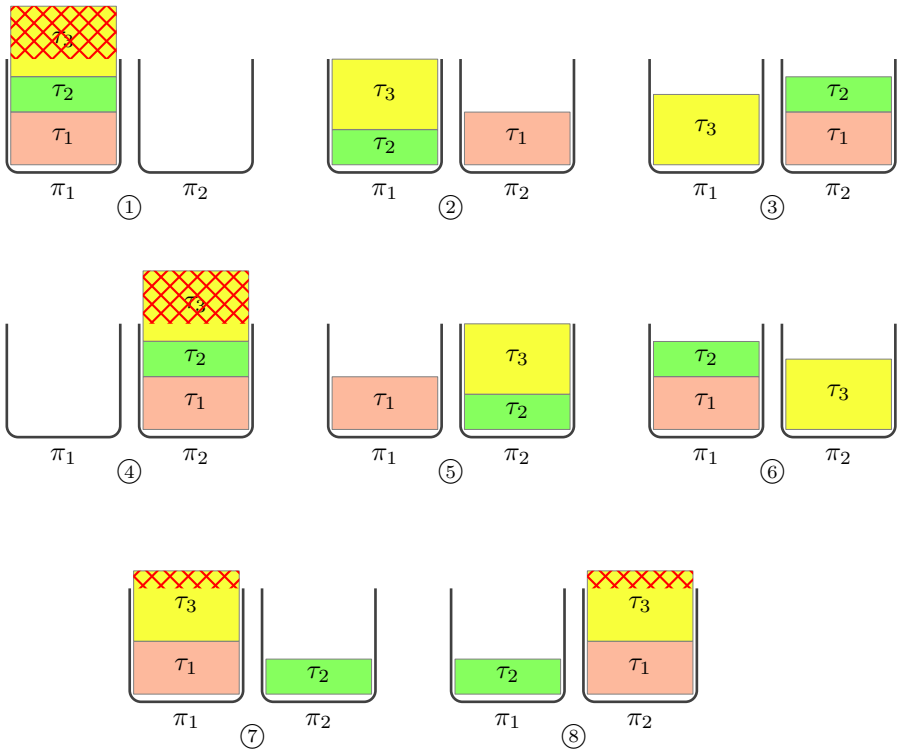
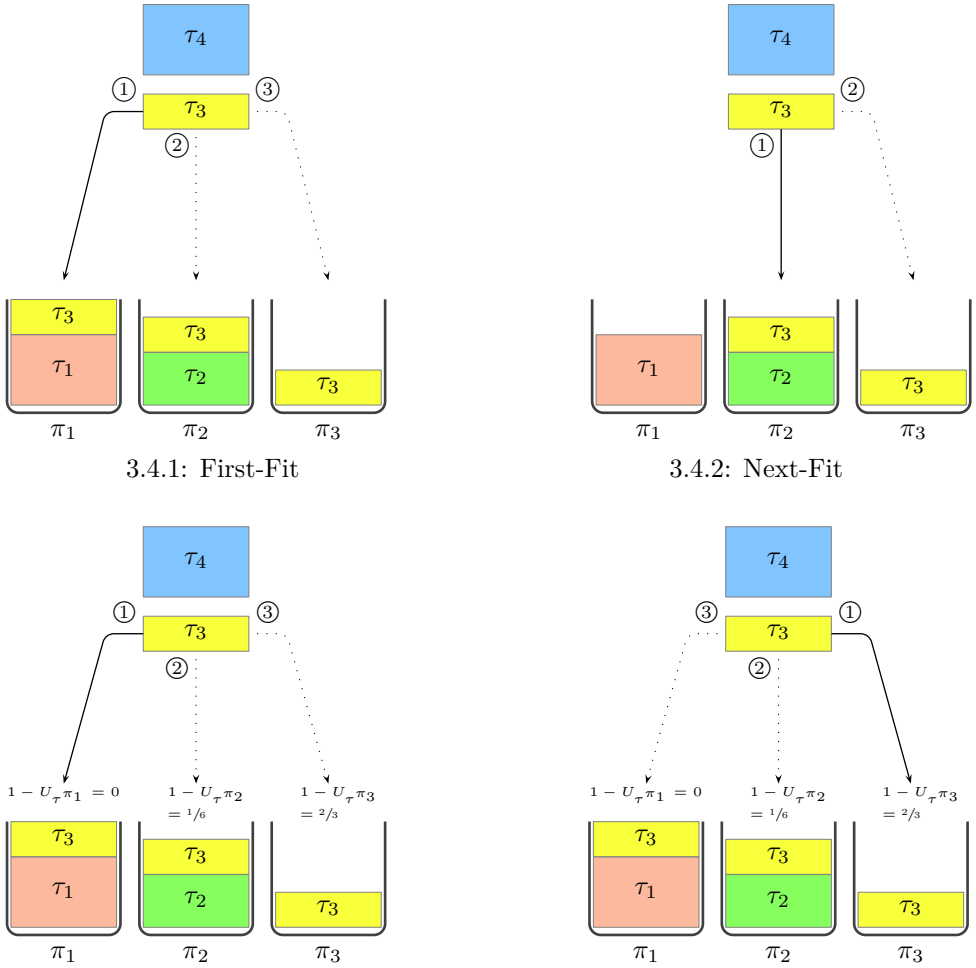


Figure 3.3 – All possible placements considered by an optimal placement for P-Scheduling approach with three tasks on two processors

Number of tasks	5	6	7	8	9	10
4 processors						
$\uplus_{heterogenous}$	1024	4096	16384	65536	262144	1048576
$\uplus_{identical}$	51	186	714	2794	11051	43947
$\uplus_{heterogenous}/\uplus_{identical}$	20.1	22.0	23.0	23.5	23.7	23.9
8 processors						
$\uplus_{heterogenous}$	32768	262144	2097152	16777216	134217728	1073741824
$\uplus_{identical}$	52	201	876	4139	21145	115928
$\uplus_{heterogenous}/\uplus_{identical}$	630.2	1304.2	2394.0	4053.4	6347.5	9262.1

Table 3.1 – Comparison of the number of possible placements for an heterogeneous and an identical multiprocessor platform

time, placement heuristics have been proposed in the state-of-the-art. The goal of such heuristics is to define a specific way to consider the processors: we do not consider all possibilities but only a specific one. The four main placement heuristics are shown in Figure 3.4. We consider the same placement problem: task τ_1 has been assigned to processor π_1, then task τ_2 has been assigned to processor π_2. We present the principle of the following heuristics:



3.4.1: First-Fit

3.4.2: Next-Fit

3.4.3: Best-Fit

3.4.4: Worst-Fit

Figure 3.4 – Principle of four basic placement heuristics

First-Fit considers the processors in a fixed order, for example by increasing ids. Then the task will be assigned to the first processor on which it can be scheduled. In Figure 3.4.1, we first consider processor π_1, then π_2 and π_3. Since the task fits into processor π_1, this processor will be selected. We will then process task τ_4 and so on.

Next-Fit considers also the processors in a fixed order, but it will start with the last processor on which tasks have been assigned and never go back to previous processors. This will reduce the number of considered processors in comparison with First-Fit. In Figure 3.4.2, we first consider processor π_2 as the last task has been assigned to this processor, then processor π_3. Since the task fits into processor π_2, this processor will be selected. We will then process task τ_4 by starting from processor π_2 and so on.

Best-Fit considers the processors in increasing order of a particular value, for example the remaining utilization on the processor. Then, this placement heuristic will select the processor on which the task "fit the best", that is minimizing the utilization value. In Figure 3.4.3, we compute the remaining utilization on each processor to determine the order: π_1, π_2 and finally π_3. Since the task fits into processor π_1, this processor will be selected. We will then process task τ_4 by calculating again the utilization on each processor and so on. Best-Fit will then try to minimize the number of processors used.

Worst-Fit considers the processors in decreasing order of a particular value, for example the remaining utilization on the processor. This placement heuristic is the dual of Best-Fit, it will select the processor on which the task "fit the worst", that is maximizing the utilization value. In Figure 3.4.4, we compute the remaining utilization on each processor to determine the order: π_3, π_2 and finally π_1. Since the task fits into processor π_3, this processor will be selected. We will then process task τ_4 by calculating again the utilization on each processor and so on. Worst-Fit will then try to fully use the platform by spreading tasks across all available processors.

Notice that we can put the previous placement heuristics into order of increasing complexity: the principle of First-Fit and Next-Fit are similar except that Next-Fit does not reconsider the past processors and so tests potentially less processors. We can then consider that Next-Fit is less complex than First-Fit. Finally, Best-Fit and Worst-Fit have larger and equal complexity since they test all processors for each choice.

3.2.2.3 Schedulability tests

The third parameter corresponds to question Q3 for a non-optimal P-Scheduling algorithm: *Is the considered task schedulable on the considered processor?* As a processor and a task have been selected, we now have to confirm that the task will be schedulable on the processor. Following the presentation of schedulability analysis in Section 2.4, we can use various schedulability tests: **S**ufficient **Tests** (S-Tests) or **N**ecessary and **S**ufficient **Tests** (NS-Tests). In our study, we decided to explore a wide variety of schedulability tests for EDF, **R**ate **M**onotonic (RM) and **D**eadline **M**onotonic (DM) schedulers.

EDF-LL is a polynomial NS-Test proposed by Liu and Layland [LL73] and designed for tasks with I-Deadlines. The test is defined by Equation 3.3.

$$U_\tau \leqslant 1 \tag{3.3}$$

EDF-BHR is a pseudo-polynomial NS-Test proposed by Baruah, Rosier, and Howell [BRH90] which is designed for tasks with **Arbitrary Deadlines** (A-Deadlines). The test is defined by Equation 3.4.

$$Load(\tau) \stackrel{\text{def}}{=} sup_{t>0} \frac{DBF(\tau, t)}{t} \leqslant 1 \tag{3.4}$$

EDF-BF is a polynomial S-Test proposed by Baruah and Fisher [BF06] and designed for tasks with A-Deadlines. The test is defined by Equation 3.5.

$$\forall \tau_i \in \tau, \begin{cases} D_i - DBF^*(\tau \backslash \{\tau_i\}, D_i) \geqslant C_i \\ 1 - \sum\limits_{\tau_j \in \tau, \tau_j \neq \tau_i} U_{\tau_j} \geqslant U_{\tau_i} \end{cases}$$
$$\text{with } DBF^*(\tau, t) = \begin{cases} \sum\limits_{i=1}^{n} (C_i + (t - D_i) \times U_{\tau_i}) & \text{if } t \geqslant D_i \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

DM-ABRTW is a pseudo-polynomial NS-Test based on the work of Joseph and Pandya [JP86] and extended by Audsley et al. [Aud+93] for DM scheduler. It is designed for tasks with **Constraint Deadlines** (C-Deadlines). The test is defined by Equation 3.6 with $WCRT_i$ given by Equation 2.11 in Subsection 2.4.2.

$$\forall \tau_i \in \tau, \ WCRT_i \leqslant D_i \tag{3.6}$$

RM-LL is a polynomial S-Test proven by Devillers and Goossens [DG00] (based on a previous proposition of Liu and Layland [LL73]) and designed for tasks with I-Deadlines. The test is defined by Equation 3.7.

$$U_\tau \leqslant n \left(\sqrt[n]{2} - 1 \right) \tag{3.7}$$

RM-BBB is a pseudo-polynomial S-Test proposed by Bini, Buttazzo, and Buttazzo [BBB03] and designed for tasks with I-Deadlines. The test is defined by Equation 3.8.

$$\prod_{i=1}^{n} (U_{\tau_i} + 1) \leqslant 2 \tag{3.8}$$

RM-LMM is a polynomial S-Test proposed by Lauzac, Melhem, and Mossé [LMM98] and designed for tasks with I-Deadlines. The test is defined by Equation 3.9. In this equation, τ' is a task set obtained after a scaling procedure proposed by the authors.

$$\begin{cases} U_\tau \leqslant n \left(\sqrt[n]{r_\tau} - 1 \right) + \frac{2}{r_\tau} - 1 & \text{if } 1 \leqslant r_\tau < 2 \\ U_{\tau'} \leqslant n \left(\sqrt[n]{r_{\tau'}} - 1 \right) + \frac{2}{r_{\tau'}} - 1 & \text{otherwise} \end{cases}$$
$$\text{with } r_\tau \stackrel{\text{def}}{=} \frac{\max(T_1, \ldots, T_n)}{\min(T_1, \ldots, T_n)} \tag{3.9}$$

3.2.3 Multi-Criteria evaluation of Generalized P-Scheduling algorithm

This section is an extension of our work with Lupu et al. [Lup+10] in which we evaluate each parameter defined in Subsection 3.2.2. We start with an overview of the conditions of the evaluation, followed by the commented results.

3.2.3.1 Conditions of the evaluation

We present in this section the conditions of the evaluation. First of all, we have to clarify how the optimal placement is used in this study. Since a criterion for sorting task is meaningless with an optimal placement, we only needed to choose a schedulability NS-Test. For EDF scheduler, we chose the NS-Test EDF-BHR and we refer to this algorithm as $OP[EDF]$. For **F**ixed **T**ask **P**riority (FTP) scheduler we focused on tasks with C-Deadline, we chose the NS-Test DM-ABRTW and we refer to this algorithm as $OP[FTP]$.

For the evaluation, we considered a platform of 4 identical processors.

Finally, in the following paragraphs, we detail the criteria used to compare the solutions and we explain the methodology applied to generate the task sets so that anyone could check our results.

3.2.3.1.1 Evaluation criteria To compare several combinations of generalized P-Scheduling algorithm parameters, we used four different performance criteria:

- *Success Ratio* is defined with Equation 3.10. It allows us to determine which combination of parameters successfully schedules the largest number of task sets.

$$\frac{\text{number of task sets successfully scheduled}}{\text{total number of task sets}} \tag{3.10}$$

- *Number of processors used* is defined as the number of processors where at least one task is assigned for a successfully scheduled task set. For instance, it allows us to determine which combination of parameters minimizes the number of processors used.

- *Processor spare capacity* is defined as the average of the remaining capacity on the used processors for a successfully scheduled task set. In Equation 3.11, the free capacity of the used processor π_j is computed with the expression $1 - Load(\tau^{\pi_j})$ for the schedulability test EDF-BHR and $1 - \Lambda_{\tau^{\pi_j}}$ otherwise. For instance, it allows us to determine which combination of parameters fulfils the used processors.

$$\frac{\sum_{\text{used processors}} (\text{spare capacity of the processor})}{\text{total number of processors used}} \tag{3.11}$$

- *Sub-optimality degree* is defined as the degree by which the success ratio of algorithm $\mathcal{A}$ is overpassed by the one of $\mathcal{A}_{ref}$. With Equation 3.12 we understand that smaller the value of $sd(\mathcal{A}, \mathcal{A}_{ref})$, the better the performance of $\mathcal{A}$ according to the one of $\mathcal{A}_{ref}$.

$$sd(\mathcal{A}, \mathcal{A}_{ref}) \stackrel{\text{def}}{=} \frac{\text{Success ratio of } \mathcal{A}_{ref} \text{ - Success ratio of } \mathcal{A}}{\text{Success ratio of } \mathcal{A}_{ref}} \times 100 \quad (3.12)$$

3.2.3.1.2 Task set generation methodology The task generation methodology used in this evaluation is based on the one presented by Baker [Bak06]. However, in our case, task generation is adapted to each type of deadline considered. In the following, $k_i \in \{D_i, T_i\}$ and $\rho_i \in \{U_{\tau_i}, \Lambda_{\tau_i}\}$. For I-Deadline task sets, $(k_i, \rho_i) = (T_i, U_{\tau_i})$ and for C-Deadline task sets $(k_i, \rho_i) = (D_i, \Lambda_{\tau_i})$. The procedure is then:

1. k_i is uniformly chosen within the interval $[1; 100]$,
2. ρ_i (truncated between 0.001 and 0.999) is generated using the following distributions:
   - uniform distribution within the interval $[1/k_i; 1]$,
   - bimodal distribution: light tasks have an uniform distribution within the interval $[1/k_i; 0.5]$, heavy tasks have an uniform distribution within the interval $[0.5; 1]$; the probability of a task being heavy is of $1/3$,
   - exponential distribution of mean 0.25,
   - exponential distribution of mean 0.5.

Task sets are generated so that those obviously not feasible ($U_\tau > m = 4$) or trivially schedulable ($n \leqslant m$ and $\forall i \in [\![1; n]\!]$, $U_{\tau_i} \leqslant 1$) are not considered during the evaluation, so the procedure is:

Step 1 initially we generate a task set which contains $m + 1 = 5$ tasks.

Step 2 we create new task sets by adding task one by one until the density of the task set exceeds $m = 4$.

For our evaluation, we generated 10^6 task sets uniformly chosen from the distributions mentioned above with I-Deadlines and C-Deadlines.

3.2.3.2 Results

This section presents a comparative study of several combinations of generalized P-Scheduling algorithm parameters. This evaluation is structured as follows:

1. we study the sub-optimality of FTP over EDF in terms of success ratio upon identical multiprocessor platform,
2. we evaluate the sub-optimality of each placement heuristic with respect to an optimal placement,
3. we determine the success ratio of each schedulability test when associated with placement heuristics,
4. for each given schedulability test, we determine the sorting criterion that maximizes its success ratio when associated with placement heuristics,
5. we compare the success ratios, number of processors used and processor spare capacities of all placement heuristics (all schedulability tests and criteria for sorting tasks included),
6. based on the best placement heuristic determined previously, we find the best association *placement heuristic* versus *criterion for sorting tasks* maximizing the success ratio.

3.2.3.2.1 Sub-optimality of FTP over EDF The degree of sub-optimality of FTP schedulers according to EDF scheduler has been previously analysed in the uniprocessor case by Davis et al. [Dav+09]. Our study determines this degree for the multiprocessor scenario (through simulation) with respect to the total density of the task set.

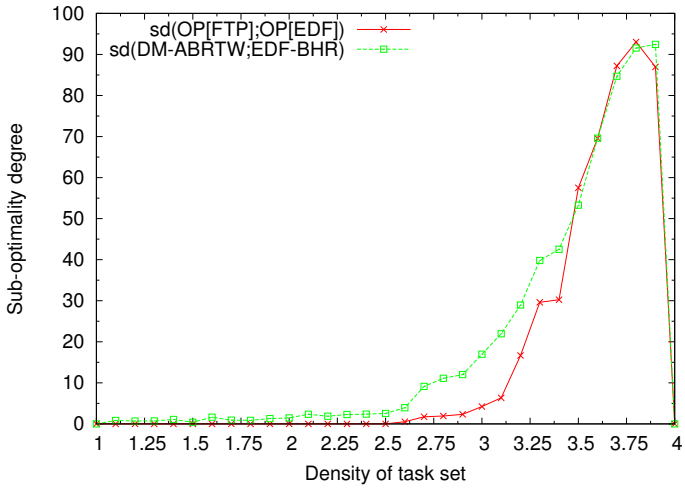


Figure 3.5 – FTP/EDF sub-optimality

Figure 3.5 shows the evaluation results as follows:

- $sd(OP[FTP], OP[EDF])$ is the sub-optimality degree in the case of an optimal task placement.

- sd(DM-ABRTW, EDF-BHR) is the sub-optimality degree in the case where the same schedulability NS-Test is combined with all four placement heuristics (all heuristics are considered one by one in order to find a schedulable placement).

For total density lower than 50% of the platform capacity, FTP and EDF are relatively equivalent. The sub-optimality degree increases starting from a density of 2 to reach a peak around a density of 3.75 for which EDF could schedule up to 93% more task sets than FTP.

When schedulability NS-Test are associated with the four heuristics the sub-optimality degree of FTP over EDF slightly increases. Though, the two curves have generally the same shape which means that the placement heuristics do not influence significantly the sub-optimality degree of the schedulability tests, especially for high density.

3.2.3.2.2 Sub-optimality of placement heuristics By definition, a placement heuristic is potentially a sub-optimal solution. In this paragraph, we present the sub-optimality degree of each placement heuristic according to the optimal placement. The associated schedulability test is the NS-Test EDF-BHR and the evaluation results include all sorting criteria (all sorting criteria are considered one by one in order to find a schedulable placement).

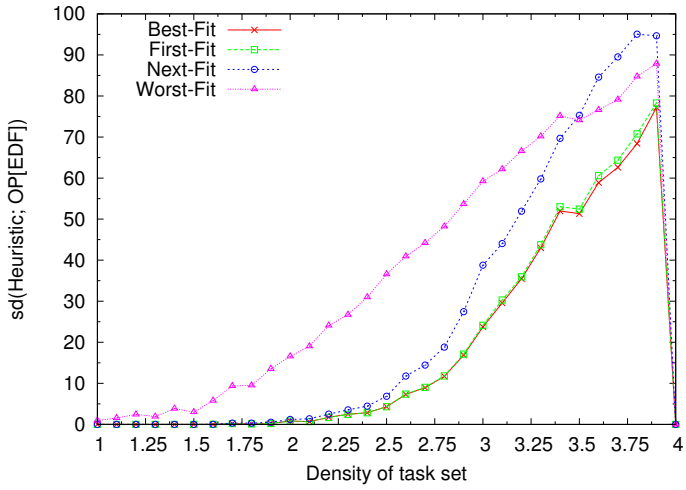
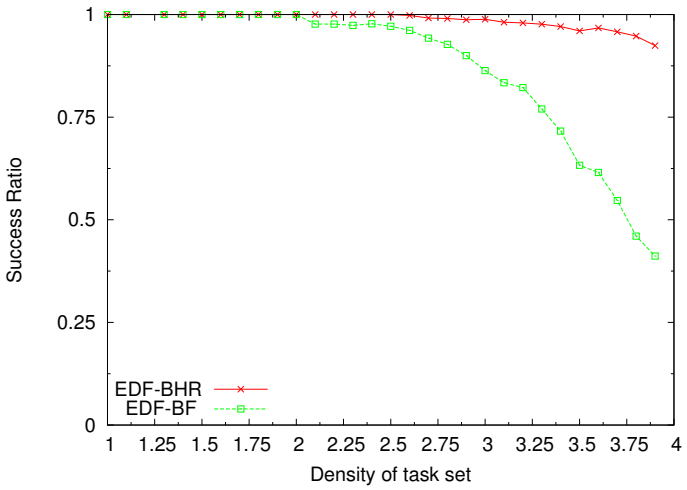


Figure 3.6 – Heuristics sub-optimality

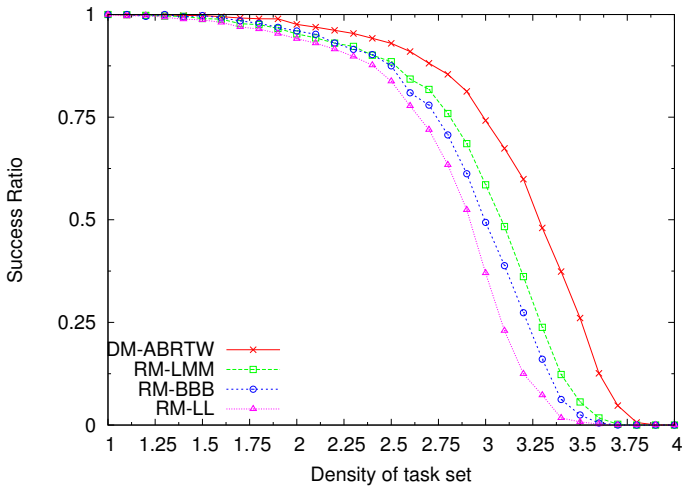
Figure 3.6 shows the results computed as sd(Heuristic, $OP[EDF]$). First of all, we remind that, in terms of complexity, the four placement heuristics can be listed in decreasing order as follows: Best-Fit and Worst-Fit (equal complexities), First-Fit and finally, Next-Fit. Figure 3.6 shows that for task sets with total density bounded by half the capacity of the platform, the performance of Best-Fit, First-Fit and Next-Fit is similar. As Next-Fit is the least complex, it is more

convenient to choose it in that case. For the scenario where the total density exceeds half of the platform capacity, Best-Fit is the best choice. Taking into account the very slight difference between the sub-optimality degree of First-Fit and Best-Fit (the difference is always lower than 2.5) and the fact that First-Fit has lower complexity, First-Fit should be also considered.

3.2.3.2.3 Choosing a schedulability test In this paragraph, we analyse the success ratios of schedulability tests for all possible combinations with the four placement heuristics and the eight criteria for sorting tasks. The analysis is divided in two sub-paragraphs: firstly, EDF scheduler tests, secondly, FTP scheduler tests.



3.7.1: EDF – **C**onstrained **Deadline** (C-Deadline)



3.7.2: FTP – **I**mplicit **Deadline** (I-Deadline)

Figure 3.7 – Schedulability tests analysis

EDF scheduler For tasks with I-Deadlines, all EDF schedulability tests reduce to EDF-LL which is a NS-Test, so we do not have anything to compare.

For the case of tasks with C-Deadlines and total task set density less than half of the platform capacity, the two schedulability tests have the same performance as seen in Figure 3.7.1. So, EDF-BF is the best option in this case because of its polynomial complexity. In the case where the total density exceeds half of the platform capacity, EDF-BHR is then a better choice despite its pseudo-polynomial time complexity, especially for high total density for which it can find a solution for up to 50% more task sets.

FTP scheduler For tasks with I-Deadlines, all the FTP schedulability tests were taken into account during the evaluation. As DM-ABRTW is a NS-Test,

it has the best performance even when associated with placement heuristics. For S-Tests, Figure 3.7.2 allows us to identify the relative performance of each schedulability test according to the success ratio: RM-LMM is the best S-Test followed by RM-BBB which outperforms RM-LL as identified in the uniprocessor case by Bini, Buttazzo, and Buttazzo [BBB03].

For the case of tasks with C-Deadlines, only DM-ABRTW is designed for these task sets, so we do not have anything to compare.

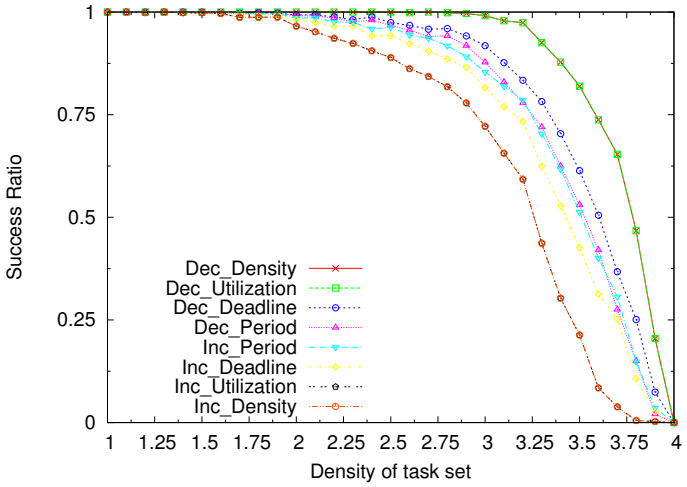
3.2.3.2.4 Choosing criterion for sorting tasks This section deals with the impact of a task sorting criterion on the success ratio of a schedulability test. In the corresponding graphs (Figures 3.8 and 3.9), *Dec* stands for *Decreasing* and *Inc* means *Increasing*.

Figure 3.8 and sub-figures show the success ratios of EDF schedulability tests for each sorting task criteria. We obtain exactly the same behaviour for every schedulability tests: the sorting task criterion which maximizes the success ratio is Decreasing Density, similar to Decreasing Utilization. It is followed by Decreasing Deadline, Decreasing Period and Increasing criteria in a symmetric way: Increasing Period, Increasing Deadline, Increasing Utilization and Increasing Density. Notice that this result has been recently confirmed by Baruah [Bar13] for EDF scheduler and I-Deadlines tasks. The demonstration proposed by Baruah used another metric referred to as *speedup factor* and defined as "the speedup factor of an approximation algorithm $\mathcal{A}$ is the smallest number f such that any task set that can be partitioned by an optimal algorithm upon a particular platform can be partitioned by $\mathcal{A}$ upon a platform in which each processor is f times as fast." The conclusion of its work is that the best P-Scheduling algorithm for EDF scheduler and tasks with I-Deadlines are those that first sort tasks according to decreasing order of utilization.

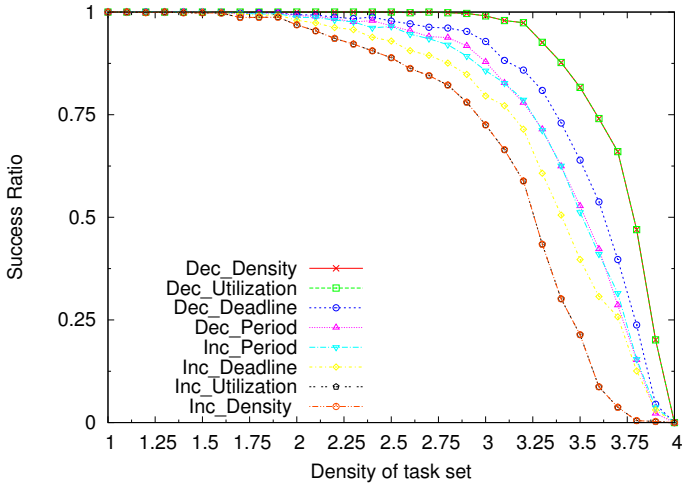
The results are exactly the same for FTP schedulability tests in Figure 3.9 and sub-figures.

3.2.3.2.5 Choosing a placement heuristic In this paragraph we evaluate the performance of the placement heuristics according to our evaluation criteria. In this analysis each placement heuristic is combined with all the schedulability tests and all the criteria for sorting tasks.

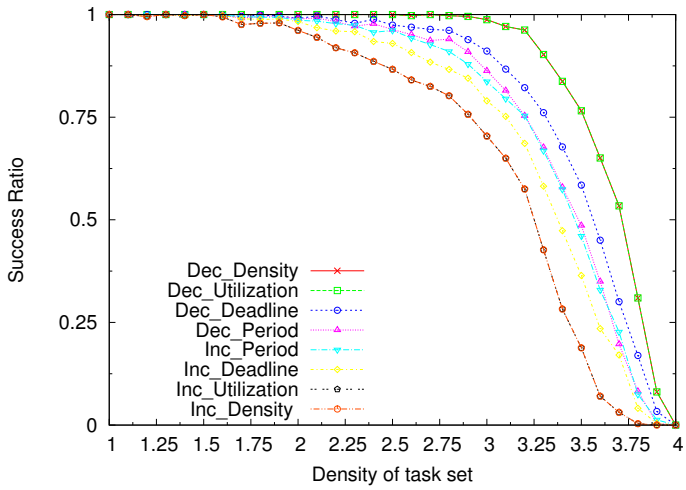
Number of used processors As seen in Figure 3.10.1, the placement heuristic that uses the smallest number of processors is Best-Fit, slightly better than First-Fit, and the one uses the largest is Worst-Fit. For low total density task sets, Best-Fit and First-Fit could use up to 50% less processors than Worst-Fit. For very high density, all heuristics give the same result. Notice that considering the relative complexity of the two best heuristics, First-Fit should be preferred to minimize the number of processors used.



3.8.1: EDF-LL



3.8.2: EDF-BHR

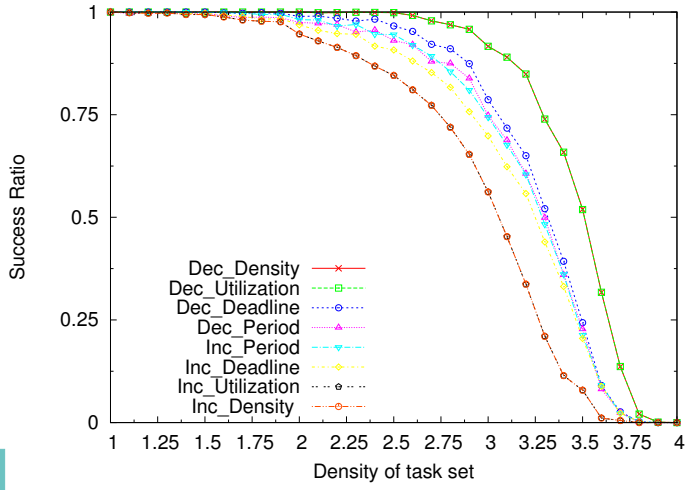


3.8.3: EDF-BF

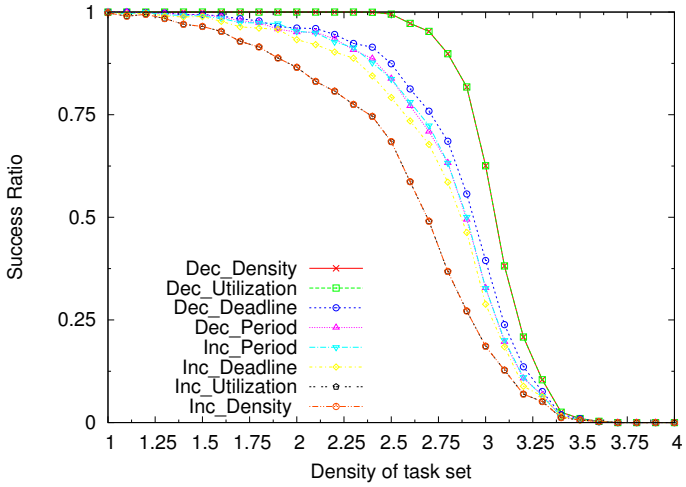
Figure 3.8 – EDF – Criteria for sorting tasks analysis

Success ratio In Figure 3.10.2 we can observe that the success ratio of placement heuristics (when combined with all schedulability tests and all the criteria for sorting tasks) follows the same performance order as in Figure 3.6: Best-Fit, First-Fit, Next-Fit and finally Worst-Fit. Taking into account the complexity of the placement heuristics and the density of the task set, we can choose: Next-Fit, if the task set requires no more than 50% of the platform capacity for execution (due to its low complexity) or, if the task set requires more than this 50% bound, First-Fit should be used for task placement on processors.

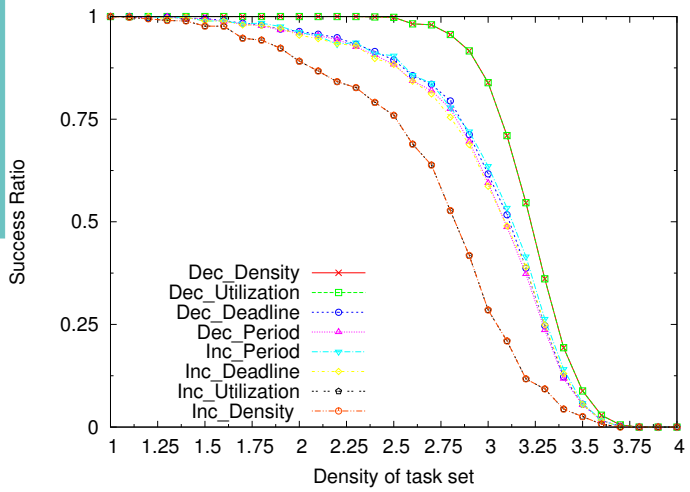
Processor spare capacity As Worst-Fit utilizes the maximum number of processors, the available spare capacity is also maximized. Figure 3.10.3 shows



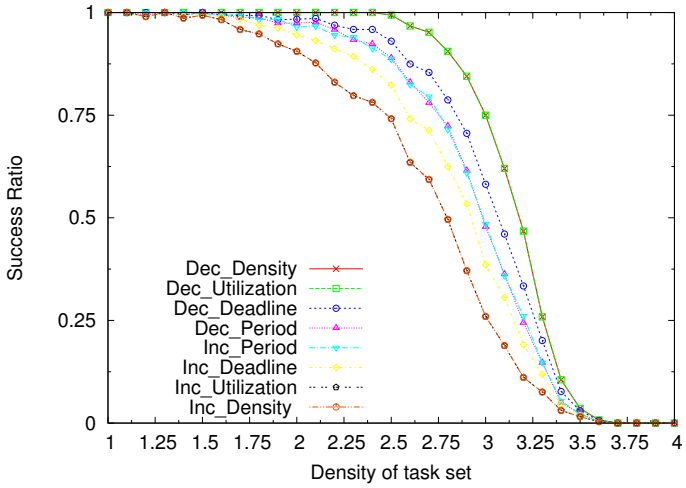
3.9.1: DM-ABRTW



3.9.2: RM-LL



3.9.3: RM-BBB



3.9.4: RM-LMM

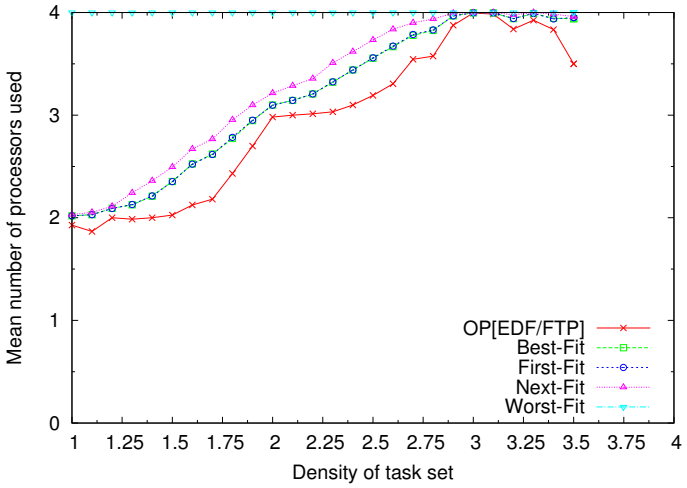
Figure 3.9 – FTP – Criteria for sorting tasks analysis

that Worst-Fit behaves as the optimal placement according to the $1-\Lambda_\tau$ criterion. Also for the $1-Load(\tau)$ criterion, Worst-Fit has the closest behaviour to the optimal task placement, as shown in Figure 3.10.4.

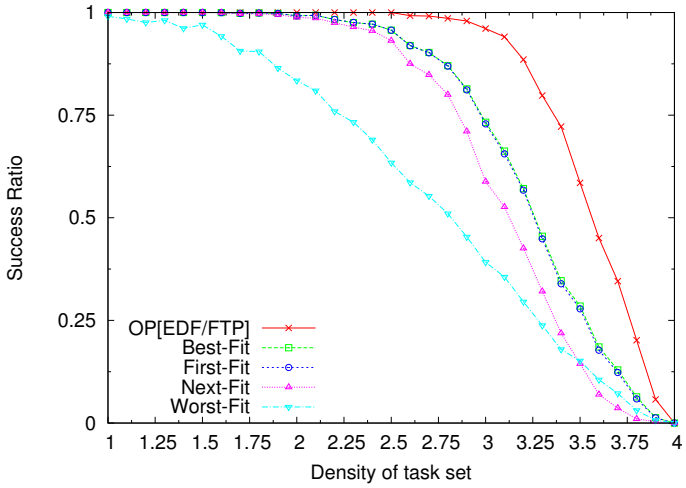
According to the evaluation results presented above, we can conclude:

- if we want to minimize the number of used processors and maximize the chance to find a schedulable placement, the best placement heuristics are Best-Fit or First-Fit.
- if we want to ensure an execution time slack (for the case where there is a risk to encounter software or hardware errors), the most suitable heuristic

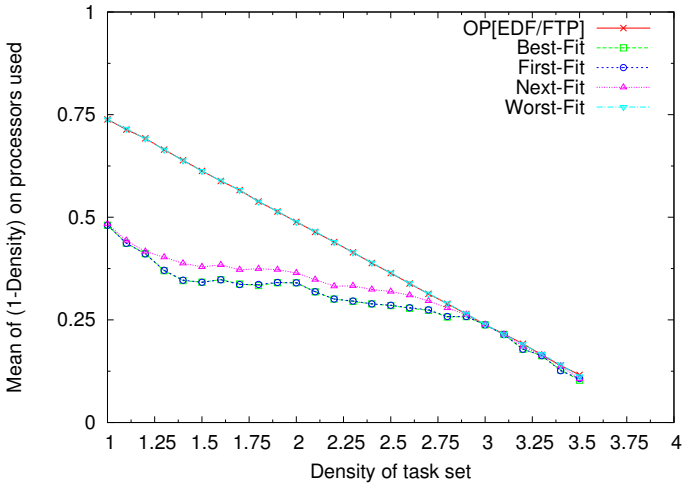
is Worst-Fit with a behaviour close to the one of an optimal placement.



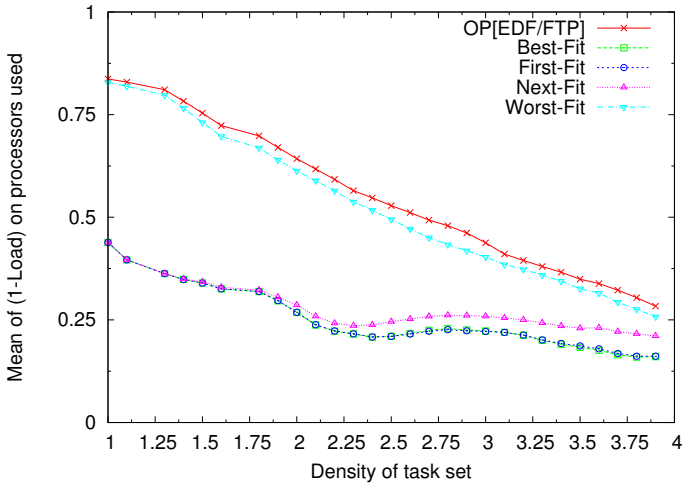
3.10.1: Number of processors used



3.10.2: Success ratio



3.10.3: Processor spare capacity – $1 - \Lambda_\tau$



3.10.4: Processor spare capacity – $1 - Load(\tau)$

Figure 3.10 – Placement heuristics analysis

3.2.3.2.6 Choosing a task criteria for the best placement heuristic According to Paragraph 3.2.3.2.5, the best placement heuristics to maximize the success ratio are Best-Fit and First-Fit. Due to its lower complexity, First-Fit is usually considered when designing P-Scheduling algorithms. It is generally agreed that the best association placement heuristic–criterion for sorting tasks is FFD (First-Fit Decreasing Utilization/Density).

Figure 3.11 shows that for task sets with the total density inferior to 75% of the platform capacity, all criteria for sorting tasks give the same performance.

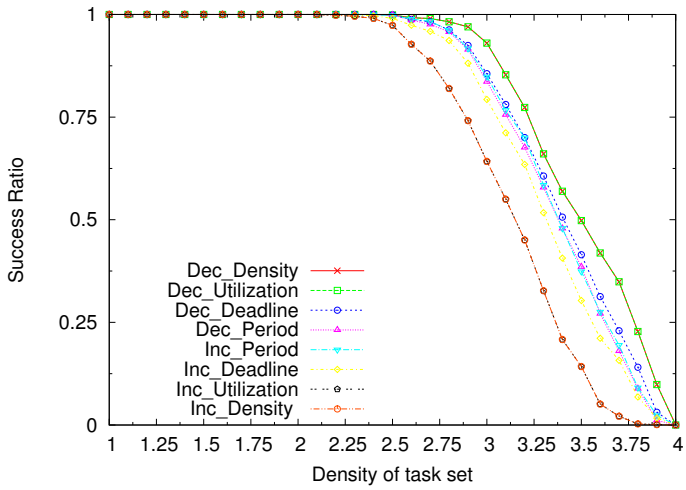


Figure 3.11 – First-Fit – Criteria for sorting task analysis

However, for task sets with total density higher than 75% of the platform capacity, Decreasing Density and Decreasing Utilization exhibit the best behaviour.

3.2.4 Summary

In this section on **Partitioned Scheduling** (P-Scheduling), we introduced a generalized algorithm. We analysed, through an evaluation, each of its parameters to know their importance and their influence according to various criteria. To conclude, we put ourselves in a practical case where we have to choose the parameters of the algorithm according to the constraints of our problem. We have identified three main practical cases:

- we only want to find a functional partitioning. Then, we would like to have a solution as fast as possible.
- we want to minimize the number of processors used. For instance, our platform is not completely defined and we want to reduce the cost minimizing the number of processors.
- we want to maximize the fault tolerance of our system. For instance, our platform is completely defined and large enough so that we can provide more robustness to execution overruns.

First of all, the solution depends on the time available to find the functional partitioning. If we are not in a constrained by the time to solve the scheduling problem, we would have to consider the optimal placement solution, especially if the problem size is small enough. For instance, if the platform contains four identical processors and the task set contains only five tasks, Table 3.1 shows that we only have 51 possible placements to consider.

Therefore, we consider in the following that the problem size is large enough or the time available to find the solution is limited. We sum up some of our results in Table 3.2 for **I**mplicit **Deadline** (I-Deadline) task sets (Table 3.2a) and for **C**onstrained **Deadline** (C-Deadline) task sets (Table 3.2b). Let us consider an example, we want to partition an I-Deadline task set with a total density which does not exceed 50% of the platform capacity, and our main objective is only to find a functional partitioning. According to Table 3.2a, the best partitioning algorithm is composed of:

- Next-Fit placement heuristic since it performs as First-Fit with a task set with low density but it has a lower complexity,
- schedulability tests RM-LL for **F**ixed **T**ask **P**riority (FTP) scheduler or EDF-LL for **E**arliest **D**eadline **F**irst (EDF) scheduler. They have the lowest complexity but give the same success ratio in this context,
- no specific sorting task as their performance is similar in this context.

	Find a functional partitioning	**Minimize number of processors**	**Maximize the fault tolerance**
	$\Lambda_\tau \leqslant 50\% \times m$		
Placement heuristic	Next-Fit	Best-Fit	Worst-Fit
Schedulability test	RM-LL for FTP scheduler, EDF-LL for EDF scheduler		
Sort tasks by	any sorting task criterion		
	$\Lambda_\tau > 50\% \times m$		
Placement heuristic	First-Fit	Best-Fit	Worst-Fit
Schedulability test	DM-ABRTW for FTP scheduler, EDF-LL for EDF scheduler		
Sort tasks by	Decreasing Utilization		

(a) **I**mplicit **Deadline** (I-Deadline) task sets

	Find a functional partitioning	**Minimize number of processors**	**Maximize the fault tolerance**
	$\Lambda_\tau \leqslant 50\% \times m$		
Placement heuristic	Next-Fit	Best-Fit	Worst-Fit
Schedulability test	DM-ABRTW for FTP scheduler, EDF-BF for EDF scheduler		
Sort tasks by	any sorting task criterion		
	$\Lambda_\tau > 50\% \times m$		
Placement heuristic	First-Fit	Best-Fit	Worst-Fit
Schedulability test	DM-ABRTW for FTP scheduler, EDF-BHR for EDF scheduler		
Sort tasks by	Decreasing Density		

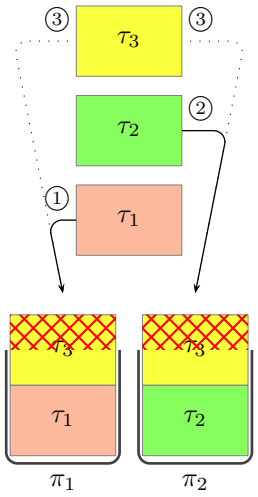
(b) **C**onstrained **Deadline** (C-Deadline) task sets

Table 3.2 – Generalized **P**artitioned **Scheduling** (P-Scheduling) algorithm parameters

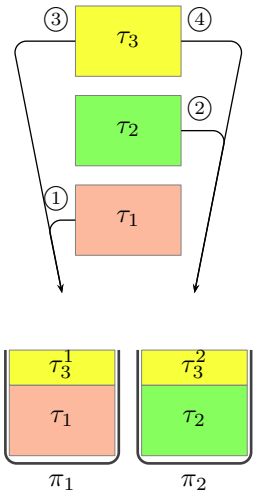
3.3 Semi-Partitioned Scheduling (SP-Scheduling)

3.3.1 Introduction

In Subsection 2.5.1.3, we expounded the **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approach which is a mix between the P-Scheduling and the **G**lobal **Scheduling** (G-Scheduling) approaches. As previously presented, the main goal of SP-Scheduling approach is to increase the number of schedulable task sets compared to the P-Scheduling, while controlling the number of migrations introduced by the G-Scheduling. The principle of a SP-Scheduling approach is also simple to understand: we try to partition the tasks until we encounter an impossibility. We then try to split the tasks into subtasks and assign those subtasks on different processors. Figure 3.12 shows an example comparing P-Scheduling and SP-Scheduling approaches. Remember that, in this chapter, we do not allow job parallelism. Therefore, a task can be split into multiple subtasks but two subtasks of a task can not execute at the same time instant.



3.12.1: Unschedulable with P-Scheduling



3.12.2: May be schedulable with SP-Scheduling

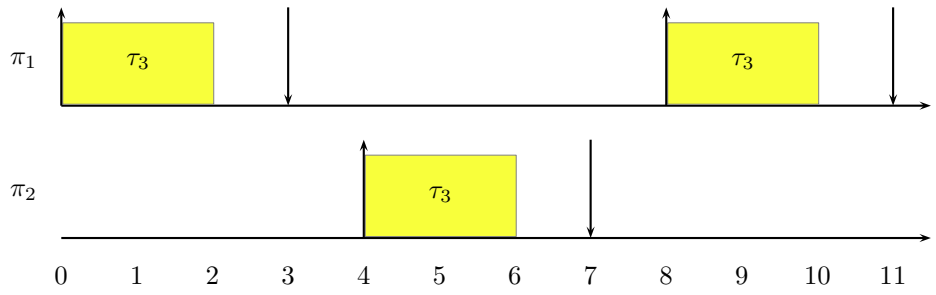
Figure 3.12 – Example of a SP-Scheduling approach

As presented in Subsection 2.5.1.3, the concept of SP-Scheduling was introduced by Anderson, Bud, and Devi [ABD05] in 2005. Let us remind the three possible degrees of migration allowed by a SP-Scheduling algorithm which will be used to split our study:

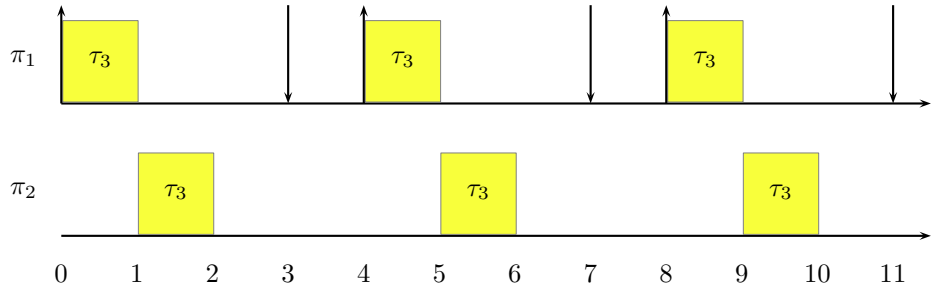
- No migration is allowed. In this case, the algorithm is a P-Scheduling algorithm.

- Migration is allowed, but only at job boundaries. A job is executed on one processor but successive jobs of a task can be executed on different processors. This solution is also referred to as the **Rest**ricted **Migration** (Rest-Migration) case, as shown in Figure 3.13.1.

- Migration is allowed and not restricted to be at job boundaries, for example a job can be portioned, each portion being executed on one processor. We will refer to this solution as the **UnRest**ricted **Migration** (UnRest-Migration) case, as shown in Figure 3.13.2. As stated in Subsection 2.5.1.3, notice that "unrestricted" does not means that the migration points cannot be fixed, but, if they are fixed, they are not restricted to be at job boundaries.



3.13.1: Rest-Migration – Migration *between* the jobs



3.13.2: UnRest-Migration – Migration *during* the job

Figure 3.13 – SP-Scheduling – Two degrees of migration allowed

In most research work, the SP-Scheduling approach is used only if the P-Scheduling approach fails. Since a migration is not cost-free for the system, the idea is to reduce the number of migrating tasks. Algorithm 3 is then a generic SP-Scheduling algorithm based on our generalized P-Scheduling Algorithm 2 where we try to split a task only if necessary.

In the following sections, we present our contribution for each of the two degrees of migration exposed in Figure 3.13. For the Rest-Migration case, we propose a

Algorithm 3: Generic SP-Scheduling algorithm

input : A task set τ, a processor set π, a sorting task criterion sortTaskCriterion, a placement heuristic placHeuristic and a uniprocessor schedulability test schedTest

output : A boolean value which notify if a schedulable solution has been found and an assignment of some or each task of τ to the processors of π

```
1  Sort tasks of τ according to sortTaskCriterion ;
2  foreach task in τ do
3      while the task is not assigned and placHeuristic gives a candidate
       processor do
4          if according to the schedTest, the task is schedulable on the
           candidate processor given by placHeuristic then
5              Assign the task to the candidate processor;
6          end if
7      end while
       /* If P-Scheduling approach fails, we try SP-Scheduling   */
8      if the task is not assigned then
9          Try to use a SP-Scheduling algorithm to split the task on multiple
           processors;
10     end if
11 end foreach
12 if All tasks are assigned then
13     return Schedulable;
14 else
15     return unSchedulable;
16 end if
```

heuristic for task splitting based on a static job migration pattern. We establish a schedulability **N**ecessary and **S**ufficient **Test** (NS-Test) for EDF scheduler associated with our static job migration pattern. For the UnRest-Migration case, we show how to generalize the approaches given in the state-of-the-art of SP-Scheduling to the general case of schedulers applying jitter cancellation before migrating a job. The basic idea is to postpone the migration of a job on a processor as long as it has not reached its maximum response time. To this end, we use intermediate deadlines. Finally, we compare the two cases using an evaluation. The results presented in this section are based on our work with George, Courbin, and Sorel [GCS11].

3.3.2 Rest-Migration approaches – RRJM

We present in this section our results for the Rest-Migration case where migrations are allowed at job boundaries only. We explain our approach called **R**ound-**R**obin **J**ob **M**igration (RRJM) and we propose an application to EDF scheduler with a schedulability NS-Test.

The RRJM is a job placement heuristic which consists in assigning the jobs of a task to a set of processors and define a recurrent pattern of successive migrations using a Round-Robin pattern, as presented in Definition 3.1.

Definition 3.1 (RRJM)**.**
Let τ_i be a sporadic sequential task assigned to a set of $\alpha_i \leqslant m$ processors according to a job placement heuristic. The job placement heuristic is a **R**ound-**R**obin **J**ob **M**igration (RRJM) placement heuristic if the job migration of τ_i follows a Round-Robin pattern, e.g.: first on π_1, then on π_2, ..., then on π_{α_i} and then again on π_1, π_2 and so forth. Notice that, in this work, a processor can appear only once in the Round-Robin pattern. ■

We now propose to define a new task model in order to represent periodic tasks following a RRJM placement heuristic.

Definition 3.2 (RRJM – Periodic task model)**.**
Let $\pi = \{\pi_1, \ldots, \pi_m\}$ be a platform of m identical processors. Let $\tau_i(O_i, C_i, T_i, D_i)$ be a periodic sequential task assigned to a set of $\alpha_i \leqslant m$ processors according to the RRJM placement heuristic. Consider that the placement is given by:

$$\begin{aligned} \uplus_{\tau_i} &= \left\{\pi^1, \ldots, \pi^{\alpha_i}\right\} \\ \text{with } \forall \alpha \in [\![1; \alpha_i]\!], \pi^\alpha &\in \pi \\ \text{and } \forall \alpha, \alpha' \in [\![1; \alpha_i]\!] \text{ with } \alpha \neq \alpha' \text{ then } \pi^\alpha &\neq \pi^{\alpha'} \end{aligned}$$

The jobs of τ_i assigned to a processor π^α could be seen as a subtask:

$$\tau_i^{\pi^\alpha,\alpha_i}(O_i^{\pi^\alpha,\alpha_i}, C_i^{\pi^\alpha,\alpha_i}, T_i^{\pi^\alpha,\alpha_i}, D_i^{\pi^\alpha,\alpha_i}) = \tau_i^{\pi^\alpha,\alpha_i}(O_i + (\alpha - 1) \times T_i, C_i, \alpha_i \times T_i, D_i)$$

Notice that the set of subtasks of any task τ_i follows the utilization conservation constraint:

$$\sum_{\alpha=1}^{\alpha_i} U_{\tau_i^{\pi^\alpha,\alpha_i}} = \sum_{\alpha=1}^{\alpha_i} \frac{C_i}{\alpha_i T_i} = \frac{1}{\alpha_i T_i} \sum_{\alpha=1}^{\alpha_i} C_i = \frac{\alpha_i C_i}{\alpha_i T_i} = \frac{C_i}{T_i} = U_{\tau_i}$$

■

Let us explain Definition 3.2 and the parameters of the subtasks. When $\tau_i(O_i, C_i, T_i, D_i)$ is strictly periodic and its first arrival instant is equal to O_i, we obtain the following pattern of arrivals on the α_i processors: the j^{th} job of

τ_i is activated at time instant $O_i + (j-1) \times T_i$ on processor $\pi^{((j-1) \mod \alpha_i)+1}$, where $(A \mod B)$ stands for the modulo function. This leads to activate jobs of τ_i on each processor executing it, with a period equal to $\alpha_i \times T_i$. Therefore, successive jobs on processor π^α can be seen as an independent subtask $\tau_i^{\pi^\alpha,\alpha_i}$ given by Definition 3.2.

Property 3.1.
The RRJM placement heuristic enables us to analyse the schedulability of Rest-Migration approaches for sporadic task sets on each processor independently. ■

Proof. Firstly, if we consider a strictly periodic task set as proposed in Definition 3.2, the RRJM placement heuristic can be seen as a new task set composed of independent subtasks and assigned to processors following a P-Scheduling approach. Secondly, since the worst case activation scenario on a uniprocessor platform is the periodic case, we have to consider periodic activations in order to propose a schedulability NS-Test for sporadic tasks using our RRJM placement heuristic. Thus, a sporadic task set is schedulable on a platform of m processors with the RRJM placement heuristic if it is schedulable on each processor independently considering a periodic activation scenario. □

Finally, we give Algorithm 4 which, in conjunction with Algorithm 3, gives a generic algorithm to use our RRJM placement heuristic.

3.3.2.1 Application to EDF scheduler

In this section, we apply our RRJM approach to the EDF scheduler. Theorem 3.2 gives a schedulability NS-Test for a task set scheduled with the *EDF-RRJM* SP-Scheduling algorithm.

Theorem 3.2 (*EDF-RRJM* schedulability NS-Test)**.**
Let $\tau_{(C,T,D)}$ be a sporadic sequential task set of n tasks scheduled with the EDF-RRJM *SP-Scheduling algorithm on m processors. A schedulability NS-Test for* EDF-RRJM *SP-Scheduling algorithm is:*

$$\forall k \in [\![1;m]\!], \quad Load\left(\tau_{(X_1^{\pi_k},T,D)} \cup \tau_{(X_2^{\pi_k},2T,D)} \cup \cdots \cup \tau_{(X_m^{\pi_k},mT,D)}\right) \leqslant 1 \tag{3.13}$$

*with $\forall j \in [\![1;m]\!]$, $X_j^{\pi_k} = (x_1^{\pi_k,j}, \ldots, x_n^{\pi_k,j})$ denotes the **W**orst **C**ase **E**xecution **T**imes (WCETs) of all subtasks assigned to processor π_k when they have a corresponding period in vector jT. Notice that $\forall i \in [\![1;n]\!]$, $x_i^{\pi_k,j} = 0$ indicates that the subtask $\tau_i^{\pi_k,j}(x_i^{\pi_k,j}, jT_i, D_i)$ is not assigned on processor π_k. Moreover, $\forall i \in [\![1;n]\!]$, $\sum_{k=1}^{m}\sum_{j=1}^{m} x_i^{\pi_k,j}/_{jT_i} = {}^{C_i}/_{T_i}$ since the subtasks of each task τ_i are an exact split of its jobs.* ■

Algorithm 4: Generic SP-Scheduling algorithm for RRJM placement heuristic

input : A task set τ, a processor set π with m processors, a task τ_i in τ, a placement heuristic placHeuristic and a uniprocessor schedulability test schedTest

output : A boolean value which notify if a schedulable solution has been found and the number α_i of processors used to execute τ_i

Data: α, k are integers, π' is a processor set used to select the processors on which τ_i will be assigned

```
1  for α = 1 to m do
2      Clear processor set π' ;
3      for k = 1 to α do
4          Create a subtask of τ_i with a period equal to α × T_i;
5          while the subtask is not assigned and placHeuristic gives a
           candidate processor do
6              if according to the schedTest, the subtask is schedulable on the
               candidate processor given by placHeuristic then
7                  Add the processor π_k to π' ;
8              end if
9          end while
10     end for
11     if π' contains α processor(s) then
           /* Task τ_i can be assigned to α processor(s)          */
12         α_i ← α;
13         Assign subtasks of τ_i to processors in π' ;
14         return Schedulable;
15     end if
16 end for
17 return unSchedulable;
```

Proof. The idea behind a SP-Scheduling approach is to split each task into subtasks when it cannot be entirely assigned to one processor. Besides, Definition 3.2 and Property 3.1 show that the subtasks generated by *EDF-RRJM* are independent from each other so they can be partitioned with a P-Scheduling algorithm. Finally, for each processor, we only have to validate the schedulability of the assigned tasks and subtasks with the schedulability NS-Test *Load* function. Furthermore, $\tau_{(X_1^{\pi_k},T,D)} \cup \tau_{(X_2^{\pi_k},2T,D)} \cdots \cup \tau_{(X_m^{\pi_k},mT,D)}$ represents exactly the tasks $(\tau_{(X_1^{\pi_k},T,D)})$ and the subtasks $(\tau_{(X_2^{\pi_k},2T,D)} \cdots \cup \tau_{(X_m^{\pi_k},mT,D)})$ assigned to processor π_k. □

Considering Theorem 3.2, if we create a complete task set $\tau_{(X_1,T,D)} \cup \tau_{(X_2,2T,D)} \cup$

$\cdots \cup \tau_{(X_m, mT, D)}$ composed of $m \times n$ tasks and subtasks, we then can apply the simplex with LPP 2.1 (see Subsection 2.4.3.1) to reduce the number of time instants to consider. The computation time of the *Load* function for each processor will then be drastically reduced.

3.3.3 UnRest-Migration approaches – MLD

This section is dedicated to the UnRest-Migration case where migrations are allowed *during* the execution of a job. We make explicit the two main problems posed by this approach (size of the execution of each portion and local deadline) and we propose an application to EDF scheduler with a schedulability NS-Test.

With the UnRest-Migration approaches, the jobs of a task $\tau_i(C_i, T_i, D_i)$ that cannot be executed on a single processor is *portioned* and executed by subtasks on a set of processors. The two main problems of portioning jobs are given by the following questions:

- Which portion of the WCET can I give to each processor?
- When will the migration occur for each portion?

Subsection 2.5.1.3 presents the state-of-the-art and gives some directions used by researchers in this field. Our work is an extension of the solution proposed by Kato, Yamasaki, and Ishikawa [KYI09] in which they decided to create local deadlines for each portion of job and use them as migration points. They fairly divide the total deadline of the task in order to create these local deadlines. Then, the portion of WCET allocated to each portion of job is maximized with an allowance study. The idea is to minimize the number of processors required to execute a task by assigning the maximum possible portion of WCET to subtask while preserving the schedulability of the task.

We refers to the solution of using local deadlines to specify migration points as the **M**igration at **L**ocal **D**eadline (MLD) approach. We propose to define a new task model in order to represent sporadic tasks following a MLD approach.

Definition 3.3 (MLD – Sporadic task model)**.**
Let $\pi = \{\pi_1, \ldots, \pi_m\}$ be a platform of m identical processors. Let $\tau_i(C_i, T_i, D_i)$ be a sporadic sequential task assigned to a set of $\alpha_i \leqslant m$ processors according to a MLD approach. Consider that the placement is given by:

$$\uplus_{\tau_i} = \left\{\pi^1, \ldots, \pi^{\alpha_i}\right\}$$
$$\text{with } \forall \alpha \in [\![1; \alpha_i]\!], \pi^\alpha \in \pi$$

The portion of jobs of τ_i assigned to a processor π^α could be seen as a subtask:

$$\tau_i^{\pi^\alpha, \alpha_i}(C_i^{\pi^\alpha, \alpha_i}, T_i^{\pi^\alpha, \alpha_i}, D_i^{\pi^\alpha, \alpha_i}) = \tau_i^{\pi^\alpha, \alpha_i}(C_i^{\pi^\alpha, \alpha_i}, T_i, D_i^{\pi^\alpha, \alpha_i})$$

Moreover, the subtasks of any task τ_i follow a precedence constraint:

if subtask $\tau_i^{\pi^{k-1},\alpha_i}$ is activated at time instant t on processor π^{k-1}

then subtask $\tau_i^{\pi^k,\alpha_i}$ will be activated at time instant $t + D_i^{\pi^{k-1},\alpha_i}$ on processor π^k

Finally, the set of subtasks of any task τ_i also follows the constraints:

$$\sum_{k=1}^{\alpha_i} C_i^{\pi_k} \geqslant C_i \tag{3.14}$$

$$\sum_{k=1}^{\alpha_i} D_i^{\pi_k} \leqslant D_i \tag{3.15}$$

in such a way that the task can be entirely executed on α_i processors and that the end-to-end deadline does not exceed the total deadline of the task. ■

We now show that the solution of using local deadlines corresponds to the case where all the processors apply jitter cancellation before migrating the job. With jitter cancellation, the job inter-arrival times are identical on all the processors executing a portion of a job. Hence, the schedulability conditions done on each processor are independent and no jitters should be taken into account in the schedulability conditions. Indeed, if we do not control the migration time instants, a task can experience release jitter that increases with the number of processors executing the task. This problem is well known in distributed systems. The holistic approach has been considered by Tindell and Clark [TC94] to compute the worst case end-to-end response time of a sporadic task, taking into account the release jitter resulting from all the visited nodes. With this approach, the **W**orst **C**ase **R**esponse **T**ime (WCRT) on each node are not independent and the jitter increases with the number of processors used.

We propose a solution based on jitter cancellation. With jitter cancellation, we cancel the release jitter of jobs before migrating them. The job arrival pattern is therefore the task arrival pattern on all processors. We are then able to apply a uniprocessor schedulability test on any processor that only depends on the subtasks assigned to the processors. For example, Balbastre, Ripoll, and Crespo [BRC06] propose this technique in the context of distributed systems. Definition 3.4 and Property 3.2 show the importance and advantage of jitter cancellation. Figure 3.14 illustrates the principle of migration at local deadline of subtask given in Definition 3.4.

Definition 3.4.
Considering task model given by Definition 3.3, with jitter cancellation, a job of the subtask $\tau_i^{\pi^k,\alpha_i}$ of a task τ_i activated at time instant t on a processor π^k will do a migration at time instant $t + D_i^{\pi^k,\alpha_i}$, where $D_i^{\pi^k,\alpha_i}$ is the local deadline of the subtask $\tau_i^{\pi^k,\alpha_i}$ on processor π^k. The duration of $D_i^{\pi^k,\alpha_i}$ is chosen to be at least equal to the WCRT of the subtask $\tau_i^{\pi^k,\alpha_i}$. ■

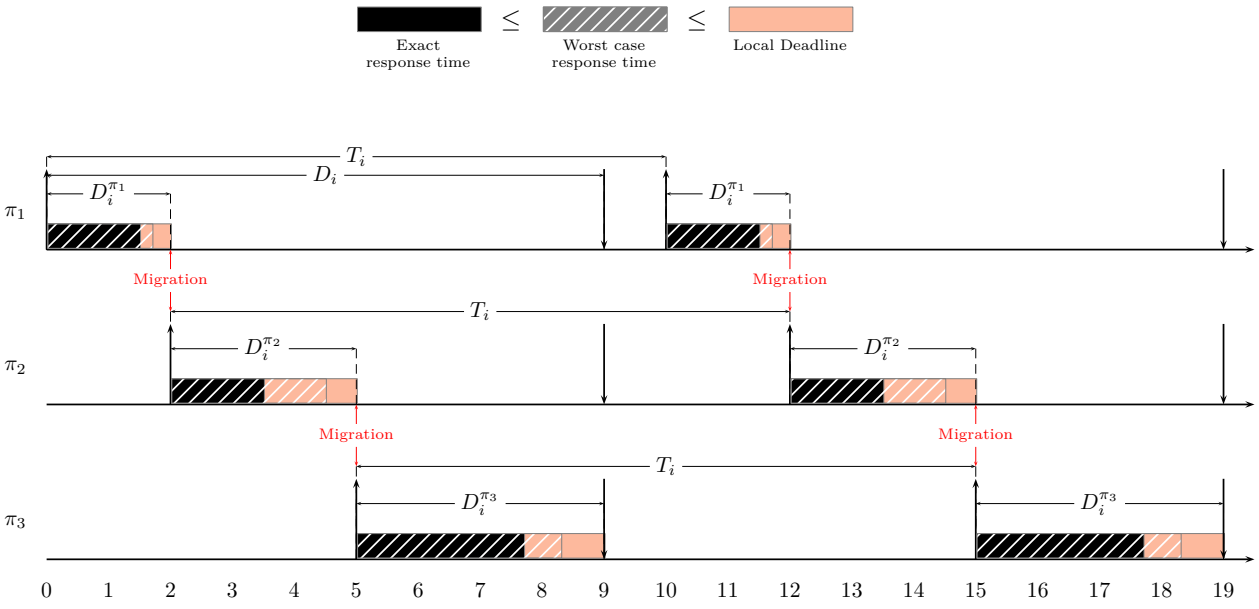


Figure 3.14 – Example of migration at local deadline

Property 3.2.
Jitter cancellation enables us to analyse the schedulability of UnRest-Migration approaches on each processor independently. ■

Proof. With a migration following the proposition given in Definition 3.4, we cancel the possible release jitter on each processor. The WCRT of a task or a subtask on a processor thus only depends on the tasks and subtasks executed on this processor with no release jitter.

With jitter cancellation, the recurrence of subtasks follows an identical pattern on the different processors. The job arrival instants of a subtask on a processor therefore follow the sporadic arrival instants of the task it comes from as the migration does not constrain the worst case scenario on each processor. □

Henceforth, we can give Theorem 3.3 which is a generic schedulability NS-Test for this MLD approach.

Theorem 3.3 (MLD schedulability NS-Test)**.**
Let $\tau_{(C,T,D)}$ be a sporadic sequential task set of n tasks. $\tau_{(C,T,D)}$ is decomposed to a new task set following Definition 3.3 with a MLD approach. Each subtask is assigned with a P-Scheduling algorithm to the m processors. A schedulability NS-Test for this MLD UnRest-Migration SP-Scheduling algorithm is:

$$\forall i \in [\![1;n]\!], \forall \alpha \in [\![1;\alpha_i]\!], \text{ the local deadline } D_i^{\pi^{\alpha},\alpha_i} \text{ is met} \tag{3.16}$$

■

Proof. According to Definition 3.3, if each subtask of a task τ_i met its deadline, the total WCET of τ_i will be executed (Equation 3.14) and the last subtask will complete at most before the deadline of τ_i (Equation 3.15), therefore we can consider that task τ_i will also meet its deadline. Needless to say that checking that all tasks meet their deadline can be done by a classical uniprocessor schedulability NS-Test for FTP or **D**ynamic **T**ask **P**riority (DTP) schedulers. □

Using Algorithm 5, we now describe the solution used to decide how we will portion a sporadic task $\tau_i(C_i, T_i, D_i)$ with a MLD approach. As Kato, Yamasaki, and Ishikawa [KYI09], we first try to assign as many tasks as possible with a classical P-Scheduling algorithm (see our generic SP-Scheduling Algorithm 3) and Algorithm 5 is called only when a task cannot be fully assigned to one processor.

The first step is to compute the local deadline $D_i^{\pi_k,\alpha}$ and the local allowance $A_i^{\pi_k,\alpha}$ of WCET on each processor π_k for a given value α. Notice that we call *allowance of WCET* the amount of execution time that we can add (if $A_i^{\pi_k,\alpha} \geqslant 0$) or that we have to subtract (if $A_i^{\pi_k,\alpha} \leqslant 0$) to the original value of WCET in order to be schedulable on a given processor. If a task cannot be fully assigned to one processor, $A_i^{\pi_k,\alpha}$ is then a negative value. We then sort processors, for example by decreasing order of allowance in order to consider first the processors which will accept a larger part of execution time for our subtasks. We finally have to verify that the maximum execution time that can be assigned to the first α processors is sufficient to execute the whole WCET of the task, thus $\sum_{j=1}^{\alpha}(C_i + A_i^{\pi_k,\alpha}) = \sum_{j=1}^{\alpha} C_i^{\pi_j,\alpha} \geqslant C_i$. If this is the case then τ_i can be assigned to the $\alpha_i = \alpha$ processors, otherwise, we increment α and try with more subtasks until reaching $\alpha = m$ subtasks.

Algorithm 5 is a generic algorithm which needs to be specialised with concrete functions to compute the local deadlines and the local allowance of WCET. Subsections 3.3.3.1 and 3.3.3.2 are dedicated to give such functions and Subsection 3.3.3.3 is an application to EDF scheduler.

3.3.3.1 Computing local deadlines

The first unknown parameter of Algorithm 5 is the computation of local deadlines. In this work we consider two different function to compute the local deadline of subtasks:

1. The *fair local deadline* computation which corresponds to the solution given by Kato, Yamasaki, and Ishikawa [KYI09]. If we consider a task τ_i and α subtasks, each one will have a same and fair deadline equal to D_i/α.

2. The *minimum local deadline* computation which corresponds to search, for a given processor and a given subtask, the minimum acceptable deadline.

Algorithm 5: Generic SP-Scheduling algorithm for MLD approaches

input : A task set τ, a processor set π with m processors, a task τ_i in τ
output : A boolean value which notify if a schedulable solution has been found and the number α_i of processors used to execute τ_i
Data: α, k are integers, π' is a processor set used to select the processors on which τ_i will be assigned

1 **for** $\alpha = 1$ **to** m **do**
2 Clear processor set π' ;
  `/* Compute local deadline and local WCET for each processor */`
3 **for** $k = 1$ **to** m **do**
4 $D_i^{\pi_k,\alpha} \leftarrow$ `computeLocalDeadline`(τ_i, π_k);
5 $A_i^{\pi_k,\alpha} \leftarrow$ `computeLocalWCETAllowance`$(\tau_i, D_i^{\pi_k,\alpha}, \pi_k)$;
6 **end for**
7 Sort processors in π, e.g. by decreasing local allowance of WCET;
8 **for** $k = 1$ **to** α **do**
9 $C_i^{\pi_k,\alpha} \leftarrow C_i + A_i^{\pi_k,\alpha}$;
10 Add the processor π_k to π' ;
11 **end for**
12 **if** $\sum_{j=1}^{\alpha} C_i^{\pi_j,\alpha} \geqslant C_i$ **then**
  `/* Task` τ_i `can be assigned to` α `processor(s) */`
13 $\alpha_i \leftarrow \alpha$;
14 Assign subtasks of τ_i to processors in π' ;
15 **return** Schedulable;
16 **end if**
17 **end for**
18 **return** unSchedulable;

The *fair local deadline* approach seems not to need more details, while the *minimum local deadline* computation depends on many parameters and especially the WCET of the subtask considered. We must therefore deal with it in depth.

First of all, if the *minimum local deadline* computation depends on the WCET of the subtask, our Algorithm 5 needs to be adapted. The principle we use is to start with a *fair local deadline* computation, then compute the local allowance of WCET and finally compute the *minimum local deadline* to get a *deadline margin* which can be used by the following subtasks. Hence, for a given value α, a subtask of task τ_i receives a fair local deadline equal to D_i/α. If we can find a processor π_k to which this subtask can be assigned with a WCET equal to $C_i^{\pi_k,\alpha}$, then we compute the *minimum local deadline* $D_{i,min}^{\pi_k,\alpha}$. The difference

$D_{reserve} = {}^{D_i}/_{\alpha} - D_{i,min}^{\pi_k,\alpha}$ is given to the following subtasks in order to increase the allowance of WCET that can be assigned on the other processors. A detailed procedure is given in Algorithm 6.

Algorithm 6: SP-Scheduling algorithm for MLD approach with minimum deadline computation

input : A task set τ, a processor set π with m processors, a task τ_i in τ
output : A boolean value which notify if a schedulable solution has been found and the number α_i of processors used to execute τ_i
Data: α, k, l are integers, π' is a processor set used to select the processors on which τ_i will be assigned, $D_{reserve}$ is the reserve of deadline

1 $D_{reserve} = 0$;
2 **for** $\alpha = 1$ **to** m **do**
3 Clear processor set π' ;
4 **for** $l = 1$ **to** α **do**
  /* Compute local deadline and local WCET for each processor */
5 **for** $k = 1$ **to** m **do**
6 $D_i^{\pi_k,\alpha} \leftarrow \frac{D_i}{\alpha} + D_{reserve}$;
7 $A_i^{\pi_k,\alpha} \leftarrow$ computeLocalWCETAllowance$(\tau_i, D_i^{\pi_k,\alpha}, \pi_k)$;
8 **end for**
9 Sort processors in π by decreasing local allowance of WCET;
  /* Since processors are sorted, π_1 is the one with the largest allowance of WCET */
10 $C_i^{\pi_1,\alpha} \leftarrow C_i + A_i^{\pi_1,\alpha}$;
11 $D_i^{\pi_1,\alpha} \leftarrow$ computeDeadlineMin$(\tau_i^{\pi_1,\alpha}, \pi_1)$;
12 $D_{reserve} \leftarrow \frac{D_i}{\alpha} - D_i^{\pi_1,\alpha}$;
13 Add the processor π_1 to π' ;
14 **end for**
15 Sort processors in π in order to place those also present in π' first.;
16 **if** $\sum_{j=1}^{\alpha} C_i^{\pi_j,\alpha} \geqslant C_i$ **then**
  /* Task τ_i can be assigned to α processor(s) */
  /* If $D_{reserve} > 0$ it is re-assigned uniformly */
17 $\alpha_i \leftarrow \alpha$;
18 Assign subtasks of τ_i to processors in π' ;
19 **return** Schedulable;
20 **end if**
21 **end for**
22 **return** unSchedulable;

Again, another parameter remains unclear in Algorithm 6: we have not specify how to compute the minimum deadline of a task. For our application to EDF scheduler, this response has already been given in Subsection 2.4.4.2 in which we study the allowance of the deadline of a task.

3.3.3.2 Computing local allowance of WCET

The second unknown parameter of Algorithm 5 is the computation of allowance of WCET in order to decide the part of execution which can be assigned to each subtask. This response has already been given in Subsection 2.4.4.1 in which we study the allowance of WCET for tasks with **A**rbitraty **Deadline** (A-Deadline) with either a FTP scheduler or EDF scheduler.

3.3.3.3 Application to EDF scheduler

In this section, we applied the MLD approach to the EDF scheduler. Theorem 3.4 gives a schedulability NS-Test for a task set scheduled with *EDF-MLD* SP-Scheduling. Notice that this theorem is only a specialization of Theorem 3.3 to the EDF scheduler case.

Theorem 3.4 (*EDF-MLD* schedulability NS-Test)**.**
Let $\tau_{(C,T,D)}$ be a sporadic sequential task set of n tasks scheduled with the EDF-MLD *SP-Scheduling algorithm on m processors. A schedulability NS-Test for* EDF-MLD *SP-Scheduling algorithm is:*

$$\forall k \in [\![1;m]\!], \quad Load\left(\tau_{(X_1^{\pi_k},T,Y_1^{\pi_k})} \cup \tau_{(X_2^{\pi_k},T,Y_2^{\pi_k})} \cup \cdots \cup \tau_{(X_m^{\pi_k},T,Y_m^{\pi_k})}\right) \leqslant 1 \tag{3.17}$$

with $\forall j \in [\![1;m]\!]$, $X_j^{\pi_k} = (x_1^{\pi_k,j}, \ldots, x_n^{\pi_k,j})$ denotes the WCETs of all subtasks assigned to processor π_k when they have a corresponding deadline in vector $Y_j^{\pi_k} = (y_1^{\pi_k,j}, \ldots, y_n^{\pi_k,j})$. Notice that $\forall i \in [\![1;n]\!]$, $x_i^{\pi_k,j} = y_i^{\pi_k,j} = 0$ indicates that the subtask $\tau_i^{\pi_k,j}(x_i^{\pi_k,j}, T_i, y_i^{\pi_k,j})$ is not assigned on processor π_k. Moreover, $\forall i \in [\![1;n]\!]$, $\sum_{k=1}^{m}\sum_{j=1}^{m} x_i^{\pi_k,j} \geqslant C_i$ and $\sum_{k=1}^{m}\sum_{j=1}^{m} y_i^{\pi_k,j} \leqslant D_i$ since the subtasks of each task τ_i are a split of its WCET and deadline. ■

Proof. The idea behind a SP-Scheduling approach is to split each task into subtasks when it cannot be entirely assigned to one processor. Besides, Definition 3.3 and Property 3.2 show that the subtasks generated by *EDF-MLD* are independent from each other so they can be partitioned with a P-Scheduling algorithm. Finally, for each processor, we only have to validate the schedulability of the assigned task and subtasks with the schedulability NS-Test *Load* function. Furthermore, $\tau_{(X_1^{\pi_k},T,Y_1^{\pi_k})} \cup \tau_{(X_2^{\pi_k},T,Y_2^{\pi_k})} \cdots \cup \tau_{(X_m^{\pi_k},T,Y_m^{\pi_k})}$ represents exactly the tasks $(\tau_{(X_1^{\pi_k},T,Y_1^{\pi_k})})$ and the subtasks $(\tau_{(X_2^{\pi_k},T,Y_2^{\pi_k})} \cdots \cup \tau_{(X_m^{\pi_k},T,Y_m^{\pi_k})})$ assigned to processor π_k. □

As in the conclusion of Subsection 3.3.2.1, it would be interesting to use a complete task set $\tau_{(X_1^{\pi_k},T,Y_1^{\pi_k})} \cup \tau_{(X_2^{\pi_k},T,Y_2^{\pi_k})} \cup \cdots \cup \tau_{(X_m^{\pi_k},T,Y_m^{\pi_k})}$ composed of $m \times n$ tasks and subtasks in order to apply the simplex with LPP 2.1 (see Subsection 2.4.3.1) to reduce the number of time instants to consider. However, LPP 2.1 helps to determine the relevant time instants in a set composed of the absolute deadlines of each task. In our case, the deadlines are variables and $y_i^{\pi_k,j}$ represents the deadline of a subtask of task τ_i assigned to processor π_k when task τ_i is split into j subtasks. Furthermore, for a given value of j, all deadlines in the set $Y_j^{\pi_k}$ are computed independently from each other. The only way to use the simplex with LPP 2.1 would be to create new task sets with all possible combinations of values of deadline for each subtask. Especially, if we use real values for deadline and not only integers, it is simply not feasible to create all possible task sets.

However, if we consider the special case of *fair local deadline* computation, then for a given value of j, $Y_j^{\pi_k}$ is composed of fixed values of deadlines equal to the original deadline of the task divided by j, $Y_j^{\pi_k} = {}^{D}/_{j}$. Then, if we create a complete task set $\tau_{(X_1^{\pi_k},T,D)} \cup \tau_{(X_2^{\pi_k},T,{}^{D}/_{2})} \cup \cdots \cup \tau_{(X_m^{\pi_k},T,{}^{D}/_{m})}$ composed of $m \times n$ tasks and subtasks, we can apply the simplex with LPP 2.1 to reduce the number of time instants to consider. Then the computation time of the *Load* function for each processor will be drastically reduced for the *fair local deadline* computation case.

3.3.4 EDF Rest-Migration versus UnRest-Migration evaluation

This section is an extension of our work with George, Courbin, and Sorel [GCS11] in which we evaluate the Rest-Migration versus UnRest-Migration SP-Scheduling approaches. We continue to focus on an application to EDF scheduler. We start with an overview of the conditions of the evaluation, followed by the commented results.

3.3.4.1 Conditions of the evaluation

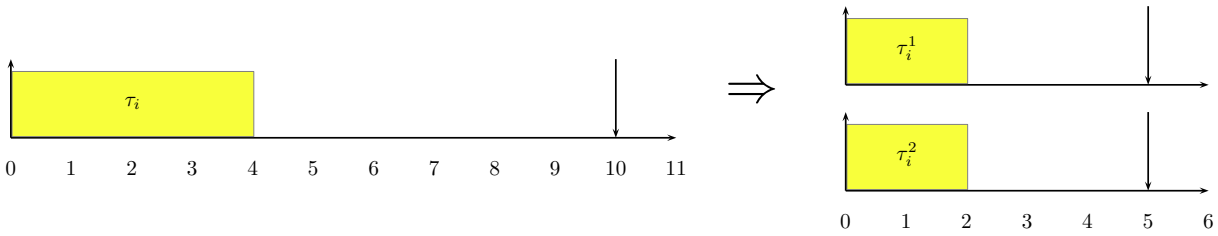
We present in this section the conditions of the evaluation. First of all, we have to clarify which algorithms are compared, then we make explicit the criteria used to compare the solutions and we explain the methodology applied to generate the task sets so that anyone could check our results. Notice, about the platform, we considered identical multiprocessor platform containing 4 processors and 8 processors.

3.3.4.1.1 Evaluated algorithms In this section we define the algorithms used in this evaluation and especially the parameters of the MLD approach for

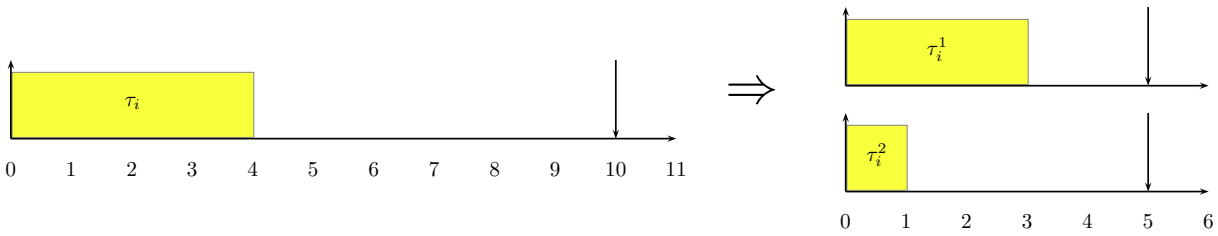
local deadline and local allowance of WCET computation. We compare the only Rest-Migration algorithm presented previously, three UnRest-Migration algorithms from the MLD approach and one P-Scheduling algorithm:

- For the Rest-Migration approach, we use our *EDF-RRJM* algorithm defined in Subsection 3.3.2 by Algorithm 4 in conjunction with Algorithm 3 and Theorem 3.2 for the schedulability NS-Test.
- For the UnRest-Migration approach, we use our generic *EDF-MLD* algorithm defined in Subsection 3.3.3 by Algorithm 5 in conjunction with Theorem 3.4 for the schedulability NS-Test. For Algorithm 5, we consider various parameters for local deadline and local allowance of WCET:
  - *EDF-MLD-Dfair-Cfair* refers to a fair local deadline computation and a fair local execution time. In other words, a task τ_i is split into α_i subtasks such that each subtask as a local execution time equal to C_i/α_i and a local deadline equal to D_i/α_i. Let us take an example: we have to split the task $\tau_i(4, 10, 10)$. Figure 3.15.1 gives the result for two subtasks. We fairly split the parameters such that the deadline of each subtask is equal to $10/2 = 5$ and the WCET of each subtask is equal to $4/2 = 2$.
  - *EDF-MLD-Dfair-Cexact* refers to a fair local deadline computation and a local execution time computed with an allowance of WCET study proposed in Subsection 3.3.3.2. This algorithm is equivalent to the solution proposed by Kato, Yamasaki, and Ishikawa [KYI09] and named *EDF-WM*. Let us take the same example: we have to split the task $\tau_i(4, 10, 10)$. Figure 3.15.2 gives a possible result for two subtasks. The deadline is fairly split such that the deadline of each subtask is equal to $10/2 = 5$. Then, the WCET of the first subtask is maximized on its relative processor, let us consider that it can be equal to 3. Finally, the last subtask receives the remaining WCET, so $4 - 3 = 1$.
  - *EDF-MLD-Dmin-Cexact* refers to a minimum local deadline computation and a local execution time computed with an allowance of WCET study. Algorithm 6 is used in this case. We take the same example: we have to split the task $\tau_i(4, 10, 10)$. Figure 3.15.3 gives a possible result for two subtasks. Firstly, the deadline of each subtask is fairly split and equal to $10/2 = 5$. Then, the WCET of the first subtask is maximized on its relative processor, let us consider that it can be equal to 3. Its deadline is then reduced to its minimum value, let us consider that it can be reduced to 4 without affecting the schedulability. Finally, the last subtask receives the remaining WCET, so $4 - 3 = 1$, and the remaining deadline, so $10 - 4 = 6$.

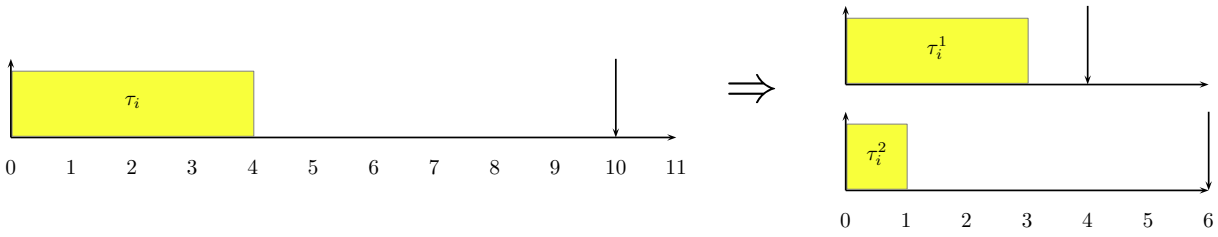
- For the P-Scheduling algorithm, we named it *EDF-P-Sched* and it corresponds to the P-Scheduling part presented in the following.



3.15.1: *EDF-MLD-Dfair-Cfair*



3.15.2: *EDF-MLD-Dfair-Cexact*



3.15.3: *EDF-MLD-Dmin-Cexact*

Figure 3.15 – Example of a task split using the three algorithms of the UnRest-Migration approach

Since all these algorithms have a P-Scheduling part, we also have to make explicit its parameters:

- tasks are sorted in Decreasing Density order as it is an optimization for all P-Scheduling algorithms (see our results in Subsection 3.2.3.2.4),
- we consider two different placement heuristics: First-Fit and Worst-Fit,
- the schedulability test for EDF scheduler is always the NS-Test based on the *Load* function.

3.3.4.1.2 Evaluation criteria To compare the various algorithms previously presented, we use two different performance criteria:

- *Success Ratio* is defined with Equation 3.18. For instance, it allows us to determine which algorithm schedules the largest number of task sets.

$$\frac{\text{number of task sets successfully scheduled}}{\text{total number of task sets}} \tag{3.18}$$

- *Density of migrations* is defined as the number of migration per time unit. Thus, the density of migrations of one task is equal to the number of migrations generated during a time interval equal to its period.
  - For a Rest-Migration approach such as *EDF-RRJM*, each migratory task generates only one migration between each job for any number of subtasks. Then the density of migrations of a Rest-Migration approach is given by Equation 3.19.

$$\sum_{\text{migratory tasks}} \frac{1}{\text{period of the task}} = \sum_{\tau_i \in \text{ migratory tasks}} \frac{1}{T_i} \tag{3.19}$$

  - For a UnRest-Migration approach such as *EDF-MLD*, each migratory task generates one migration between each subtask. Then the density of migrations of a UnRest-Migration approach is given by Equation 3.20.

$$\sum_{\text{migratory tasks}} \frac{\text{number of subtasks}}{\text{period of the task}} = \sum_{\tau_i \in \text{ migratory tasks}} \frac{\alpha_i}{T_i} \tag{3.20}$$

3.3.4.1.3 Task set generation methodology The task generation methodology used in this evaluation is based on the one presented by Baker [Bak06]. However, in our case, task generation is adapted to each type of deadline considered. In the following, $k_i \in \{D_i, T_i\}$ and $\rho_i \in \{U_{\tau_i}, \Lambda_{\tau_i}\}$. For I-Deadline task sets, $(k_i, \rho_i) = (T_i, U_{\tau_i})$ and for C-Deadline task sets $(k_i, \rho_i) = (D_i, \Lambda_{\tau_i})$. The procedure is then:

1. k_i is uniformly chosen within the interval $[1; 100]$,
2. ρ_i (truncated between 0.001 and 0.999) is generated using the following distributions:
   - uniform distribution within the interval $[1/k_i; 1]$,
   - bimodal distribution: light tasks have an uniform distribution within the interval $[1/k_i; 0.5]$, heavy tasks have an uniform distribution within the interval $[0.5; 1]$; the probability of a task being heavy is of $1/3$,
   - exponential distribution of mean 0.25,

- exponential distribution of mean 0.5.

Task sets are generated so that those obviously not feasible ($U_\tau > m = \{4, 8\}$) or trivially schedulable ($n \leqslant m$ and $\forall i \in [\![1; n]\!],\ U_{\tau_i} \leqslant 1$) are not considered during the evaluation, so the procedure is:

Step 1 initially we generate a task set which contains $m + 1 = 5$ tasks.

Step 2 we create new task sets by adding task one by one until the density of the task set exceeds m.

For our evaluation, we generated 10^6 task sets uniformly chosen from the distributions mentioned above with I-Deadlines and C-Deadlines.

We decided to reduce the time granularity (the minimum possible value of each parameter) to 1. Thus, for the evaluation, for each task τ_i its parameters C_i, T_i and D_i are considered as integers. Considering that the values are discretized according to the clock tick, it is always possible to modify all the parameters to integer values by multiplying them by an appropriate factor. To simplify testing, we used this approach and all the parameters are limited to integer values. This does not imply, however, that the algorithms used and presented in this evaluation cannot be applied to non-integer values.

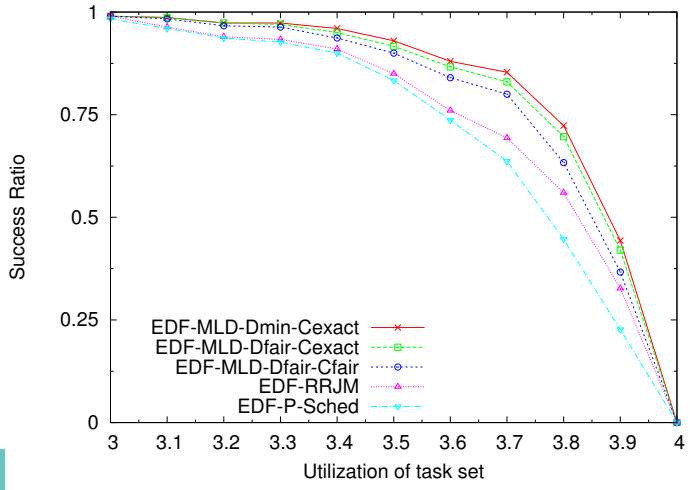
3.3.4.2 Results

Now, we show the evaluation results, obtained for 4 and 8 processors under discrete time granularity, in terms of success ratio in Subsection 3.3.4.2.1 and then in terms of density of migrations in Subsection 3.3.4.2.2.

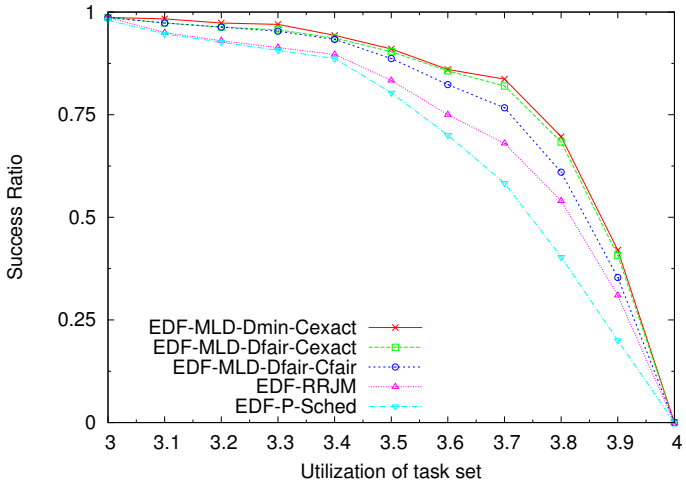
3.3.4.2.1 Success Ratio Figure 3.16 shows the results of simulations based on the success ratio with 4 processors. Our graphs focus on the range of utilization $[3; 4]$ since all algorithms of P-Scheduling and SP-Scheduling approaches implemented in this study have the same performance with a lower utilization. In the same way, Figure 3.17 shows the results obtained with 8 processors and focus on the range $[7; 8]$. As expected, SP-Scheduling approaches become useful for high utilization task sets.

We have carried out a study to compare the behaviour of these algorithms with task sets exclusively composed of light tasks (based on task set generated with an exponential distribution of mean 0.25) or heavy tasks (with an exponential distribution of mean 0.75) but the difference between these results and the general case did not appear significant. We therefore focus on the arbitrary case of experiments.

SP-Scheduling approaches improve the success ratio of EDF P-Scheduling. Since the split of tasks is done only when the P-Scheduling algorithm fails to

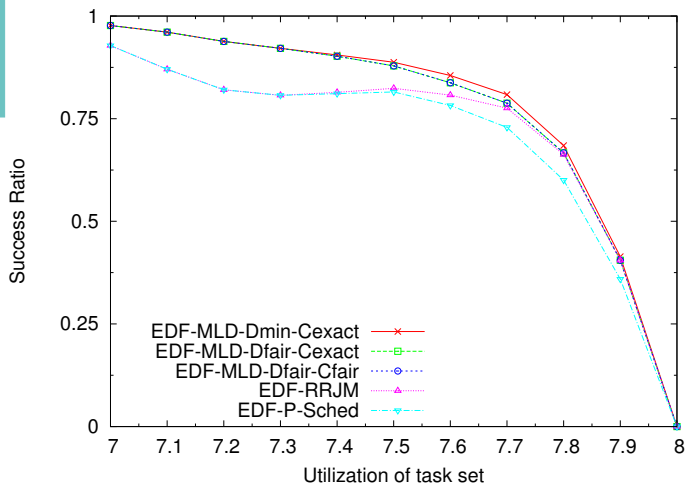


3.16.1: First-Fit placement heuristic

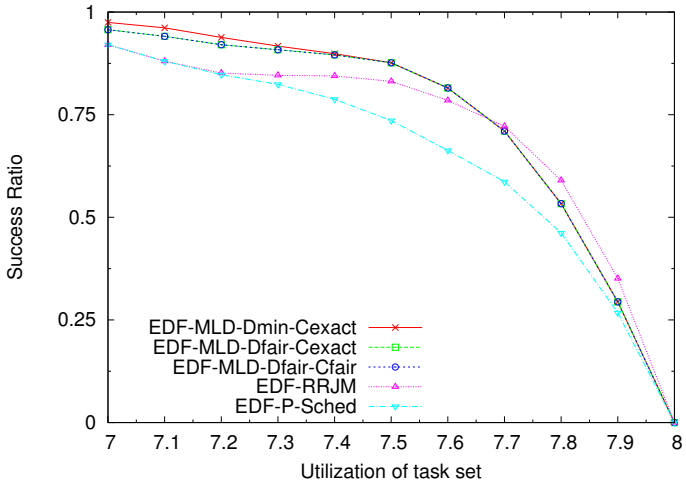


3.16.2: Worst-Fit placement heuristic

Figure 3.16 – Success Ratio analysis – 4 processors



3.17.1: First-Fit placement heuristic



3.17.2: Worst-Fit placement heuristic

Figure 3.17 – Success Ratio analysis – 8 processors

assign a task on a processor, SP-Scheduling algorithms schedule all task sets that are schedulable with a EDF P-Scheduling algorithm.

We compare for 4 and 8 processors the percentage of the improvement in the success ratio (the difference between the success ratio of EDF SP-Scheduling algorithms and EDF P-Scheduling algorithm multiplied by 100) when First-Fit and Worst-Fit are used.

With 4 processors, the trends obtained with First-Fit and Worst-Fit are similar. In Table 3.3, we present a comparative table of the percentage of the

	Utilization	P-Scheduling success ratio	SP-Scheduling success ratio	Improvement (%)
	First-Fit placement heuristic			
EDF-RRJM	3.9	0.2267	0.3267	44.11
EDF-MLD-Dfair-Cfair	3.9	0.2267	0.3667	61.76
EDF-MLD-Dfair-Cexact	3.9	0.2267	0.4200	85.27
EDF-MLD-Dmin-Cexact	3.9	0.2267	0.4433	95.54
	Worst-Fit placement heuristic			
EDF-RRJM	3.9	0.2000	0.3100	55.00
EDF-MLD-Dfair-Cfair	3.9	0.2000	0.3533	76.65
EDF-MLD-Dfair-Cexact	3.9	0.2000	0.4067	103.35
EDF-MLD-Dmin-Cexact	3.9	0.2000	0.4200	110.00

Table 3.3 – Best improvement, in %, of success ratio for each SP-Scheduling algorithms with respect to the P-Scheduling algorithm for 4 processors

success ratio improvement. We also provide the value of task set utilization for which the percentage of difference is reached. *EDF-MLD-Dmin-Cexact* slightly outperforms all the others. The performance of *EDF-MLD-Dfair-Cexact* remains higher that *EDF-MLD-Dfair-Cfair*. Thus, the success ratio is clearly proportional to the complexity of computation. In terms of the percentage of improvement, *EDF-RRJM* outperforms *EDF-P-Sched*: between 1% and 10% for a processor utilization of less than 3.6 and up to 55% for task sets with higher utilization. In the same range, *EDF-MLD-Dfair-Cexact* improves the schedulability respectively by 1% to 20% and up to 103.3% for high utilization. Finally, *EDF-MLD-Dmin-Cexact* reaches an improvement of 110% which represents a gain of about 5.56% compared to *EDF-MLD-Dfair-Cexact* with a Worst-Fit placement heuristic. As we expected, this approach become interesting for task sets with a very high total utilization.

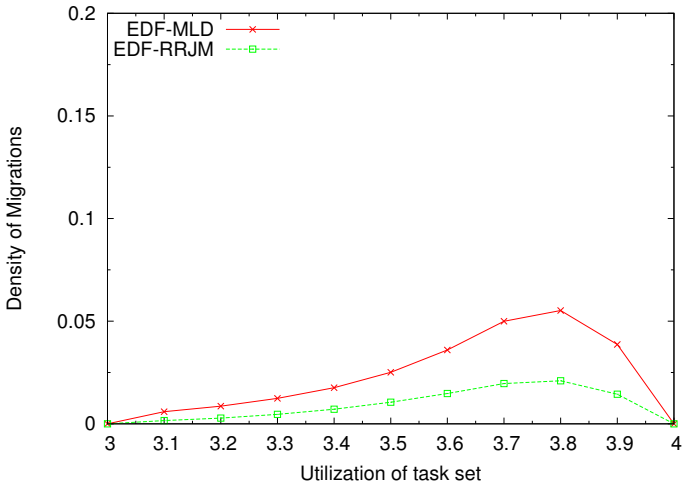
With 8 processors, we present in Table 3.4 a comparative table of maximum percentage of success ratio improvement. We also provide the value of task set utilization for which this percentage is reached. If with First-Fit the performance of the algorithms is similar with 4 or 8 processors, with Worst-Fit, at very high utilization, the job placement *EDF-RRJM* reveals its potential. When a SP-Scheduling MLD approach is limited by the time granularity, *EDF-RRJM* can always create from a task up to m subtasks of period multiplied by m. For example, suppose that a task $\tau_i(C_i, T_i, D_i) = \tau_i(1, 2, 2)$ cannot be fully assigned to one processor and the time granularity is 1. All *EDF-MLD-** algorithms fail to split this task while *EDF-RRJM* can create up to m subtasks with period equal to $m \times T_i$ and potentially succeed in scheduling the task set. Consequently, it seems that an UnRest-Migration approach cannot always take advantage of an increase in the number of processors while the Rest-Migration approach is able to take advantage of it. For very high utilization and discrete time granularity,

	Utilization	P-Scheduling success ratio	SP-Scheduling success ratio	Improvement (%)
	First-Fit placement heuristic			
EDF-RRJM	7.9	0.3593	0.4052	12.78
EDF-MLD-Dfair-Cfair	7.2	0.8204	0.9381	14.35
EDF-MLD-Dfair-Cexact	7.2	0.8204	0.9381	14.35
EDF-MLD-Dmin-Cexact	7.9	0.3593	0.4141	15.25
	Worst-Fit placement heuristic			
EDF-RRJM	7.9	0.2674	0.3511	31.30
EDF-MLD-Dfair-Cfair	7.6	0.6630	0.8152	22.96
EDF-MLD-Dfair-Cexact	7.6	0.6630	0.8152	22.96
EDF-MLD-Dmin-Cexact	7.6	0.6630	0.8152	22.96

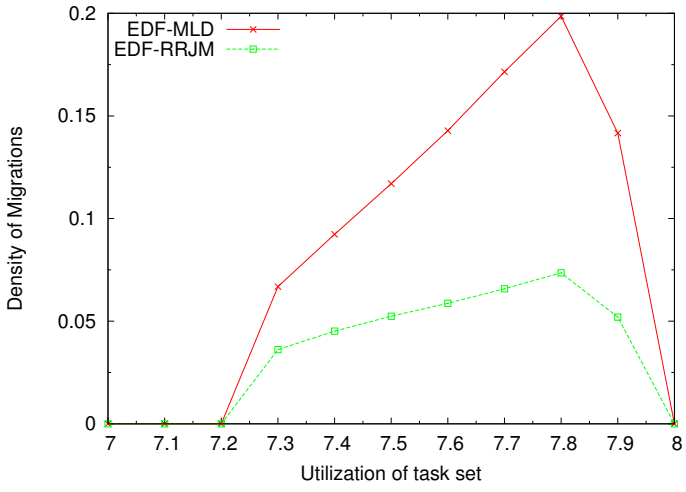
Table 3.4 – Best improvement, in %, of success ratio for each SP-Scheduling algorithms with respect to the P-Scheduling algorithm for 8 processors

EDF-RRJM reaches an improvement of 31.3% compared to *EDF-P-Sched* which represents a gain of about 19.3% compared to *EDF-MLD-Dfair-Cexact*.

3.3.4.2.2 Density of migrations Figure 3.18 shows the results of the evaluation based on the density of migrations. Since a migration occurs only when the SP-Scheduling technique is used, the results show no migration with low task sets utilization. Our graphs focus on the same range of utilization $[3; 4]$ and $[7; 8]$, respectively with 4 and 8 processors.



3.18.1: 4 processors



3.18.2: 8 processors

Figure 3.18 – Density of migrations analysis

In order to obtain representative graphs, we compute the density of migrations

only for task sets schedulable with all the SP-Scheduling algorithms. *EDF-RRJM* leads to one migration per task activation, whereas *EDF-MLD* approaches produce a number of migrations per task activation at most equal to the number of subtasks. The density of migrations for *EDF-RRJM* is on average 37% (respectively 43%) of the density of migrations for *EDF-MLD* algorithms for 4 (respectively 8) processors. Hence, the average number of migrations obtained with *EDF-MLD* algorithms is 2.69 (respectively 2.32) times the number of migrations of *EDF-RRJM* for 4 (respectively 8) processors.

3.3.5 Summary

In this section we have considered the problem of **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) according to two approaches: **Rest**ricted **Migration** (Rest-Migration) and **UnRest**ricted **Migration** (UnRest-Migration). We evaluate the two approaches through an application to **E**arliest **D**eadline **F**irst (EDF) scheduler.

The first approach, for which we propose an algorithm denoted **R**ound-**R**obin **J**ob **M**igration (RRJM), is based on migrations at job boundaries with a Round-Robin job migration pattern. The solution is easy to implement and results in few migrations. With a First-Fit heuristic, it is outperformed by the UnRest-Migration approach but performs better than classical **P**artitioned **Scheduling** (P-Scheduling) algorithm by a ratio that can reach 44.11%. For a Worst-Fit heuristic with high task set utilization and 8 processors, our Rest-Migration approach performs better than the UnRest-Migration approach. In this case, the algorithm based on Round-Robin job migration heuristic outperforms the best UnRest-Migration **M**igration at **L**ocal **D**eadline (MLD) algorithm by a ratio that can reach 19.3% (under discrete time granularity).

For the second approach, referred to as the UnRest-Migration approach, we propose a generalization denoted MLD in which we assign local deadlines to subtasks. Based on this local deadline, the maximum acceptable portion of **W**orst **C**ase **E**xecution **T**ime (WCET) is computed. We have considered two local deadline assignment schemes, according to a fair local deadline computation or a minimum local deadline computation. The migration is done at local deadline of the subtasks to cancel the release jitter before doing a migration. We show that these algorithms outperform the classical P-Scheduling algorithm by a ratio that can reach 110% for the best algorithm at very high utilization.

Considering the number of migrations, UnRest-Migration approaches produce at least two times more migrations (on the average) than the Rest-Migration approach.

Chapter 4

Scheduling Parallel Task (P-Task)

La fourmi est un animal intelligent collectivement et stupide individuellement ; l'homme c'est l'inverse.

The ant is a collectively intelligent and individually stupid animal; man is the opposite.

Karl Von Frisch

Contents

4.1 Introduction

In this chapter, we present our contributions to the **R**eal-**T**ime (RT) scheduling of **P**arallel **Tasks** (P-Tasks) upon identical multiprocessor platform. Based on the state-of-the-art analysis in Subsection 2.5.2, we studied the two main classes of P-Task model: Gang and Multi-Thread. In Section 4.2 we give more details about the Gang task model. Section 4.3 is dedicated to the presentation of our new Multi-Thread task model called **M**ulti-**P**hase **M**ulti-**T**hread (MPMT) published by Courbin, Lupu, and Goossens [CLG13]. In Section 4.4 we present the schedulers for parallel Multi-Thread RT tasks. Section 4.5 contains our results on the schedulability analysis for our MPMT task model: schedulability **N**ecessary and **S**ufficient **Tests** (NS-Tests) for two schedulers and an analysis of the **W**orst **C**ase **R**esponse **T**ime (WCRT) of MPMT tasks. Finally, in Sections 4.6 and 4.7 we compare Gang and Multi-Thread task models.

First, let us introduce our specific vocabulary. Throughout this work, we distinguished between off-line entities (task, see Definition 2.2 on page 9) and runtime entities (job, or task instance, see Definition 2.3 on page 10). In this chapter we use the same distinction with an extended version and vocabulary to link the theory of RT scheduling (e.g., task) to the reality of parallel programming (e.g., process or thread). Moreover, it will be used to distinguish schedulers which attribute priority according to runtime or off-line parameters. Definition 4.1 and Definition 4.3 are the extended versions of definitions for tasks and jobs. Definition 4.2 and Definition 4.4 are new specific definitions for the case of parallel tasks.

Definition 4.1 (Task (extended))**.**
A *task* is defined as the set of common *off-line* properties of a set of works that need to be done. In addition to properties, a task can be composed of sub-programs. By analogy with object-oriented programming and **U**nified **M**odeling **L**anguage (UML) standard, a task can be seen as a *class* with multiple attributes and linked with another class "sub-programs" by a *composition* relationship: *task* is composed of *sub-programs*. ■

Definition 4.2 (Sub-program)**.**
A *sub-program* is defined as the set of common *off-line* properties of a set of works that need to be done and which are a part of a larger work. By analogy with object-oriented programming and UML standard, a sub-program can be seen as a *class* with multiple attributes and linked with another class "task" by a *composition* relationship: *sub-program* is a component part of *task*. ■

Definition 4.3 (Process (or Job))**.**
A *process*, or *instance* of task, is the *runtime* occurrence of a task. By analogy with object-oriented programming, a process can be seen as the *object* created after instantiation of the corresponding task class. ■

Definition 4.4 (Thread)**.**
A *thread*, or *instance* of sub-program, is the *runtime* occurrence of a sub-program. By analogy with object-oriented programming, a thread can be seen as the *object* created after instantiation of the corresponding sub-program class. ■

In other words, we consider that a *task* is defined off-line while its instance exists only at runtime under the denomination *process* (or *job* in the sequential case). In the same vein we consider that a *sub-program* is defined off-line while its instance exists only at runtime under the denomination *thread*. Consequently the scheduler manages processes and/or threads (only *jobs* in the sequential case). Meanwhile the process or thread priority can (or cannot) be based on the static task and sub-program characteristics. A summary of this vocabulary is presented in Table 4.1.

Off-line	Runtime
Task	Process/Job
Sub-program	Thread

Table 4.1 – Off-line versus Runtime vocabulary

Remark 4.1. Notice that a task is generally considered as a computer program in this thesis, even if research on RT and theories developed here apply to other considerations. This is one reason why we chose the term "sub-program". Notice also that the term "subtask" is used in this thesis to represent tasks artificially created from a splitting of a real task whereas the term "sub-program" refers to a real and not artificial part of a task. ◇

4.2 Gang task model

We already presented the Gang task model in Subsection 2.2.2.3.1 which was proposed by Kato and Ishikawa [KI09]. In this section, we present a modified version which allows us to define a variable number of processors for each task. Indeed, in the task model presented in Definition 2.6, the number of processors used by a task τ_i is defined by the fixed value V_i. As we discussed in a paper with Berten, Courbin, and Goossens [BCG11], Definition 4.5 and Figure 4.1 give the details of this task model.

Definition 4.5 (Periodic parallel Gang task set)**.**
Let $\tau_{(O,C,T,D)} = \{\tau_1(O_1, C_1, T_1, D_1), \ldots, \tau_n(O_n, C_n, T_n, D_n)\}$ be a periodic parallel Gang task set composed of n periodic parallel Gang tasks. The task set $\tau_{(O,C,T,D)}$ can be abbreviated as τ. A periodic parallel Gang task $\tau_i(O_i, C_i, T_i, D_i)$, abbreviated as τ_i (Figure 4.1), is characterized by the 4-tuple (O_i, C_i, T_i, D_i) where:

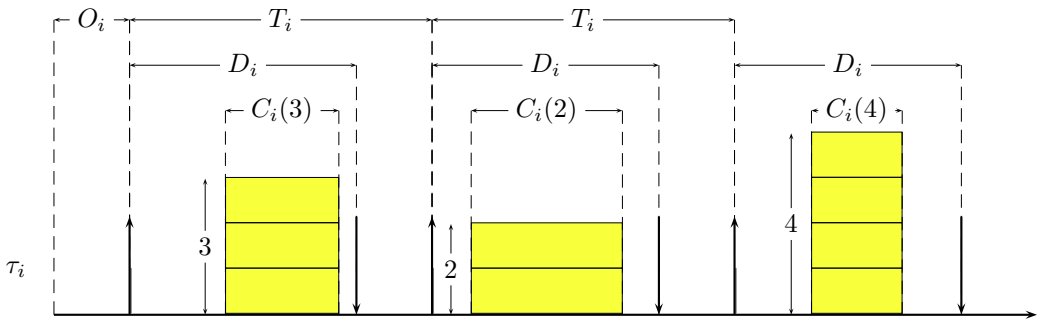


Figure 4.1 – Representation of a periodic parallel Gang task, from Definition 4.5

- O_i is the *first arrival instant* of τ_i, i.e., the instant of the first activation of the task since the system initialization.
- C_i is the ***W**orst **C**ase **E**xecution **T**ime (WCET)* of τ_i. $C_i(v)$ is a function which gives, for each value v, the WCET when executed in parallel on v processors, i.e., the maximum execution time required when simultaneously executed on v processors.
- T_i is the *period* of τ_i, i.e., the exact inter-arrival time between two successive activations of τ_i.
- D_i is the *relative deadline* of τ_i, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant. Notice that we consider **C**onstrained **D**eadline (C-Deadline), so $D_i \leqslant T_i$.

■

Since all threads of a Gang task have to execute simultaneously, the execution of a job of τ_i is represented as a "$C_i(v) \times v$" rectangle in "time $\times$ processor" space. Moreover, we specify Property 4.1 and Property 4.2 which constrain the values of C_i.

Property 4.1.
We consider that adding a processor to schedule a Gang task cannot increase the execution time as expressed by Equation 4.1.

$$\forall v < w, C_i(v) \geqslant C_i(w) \tag{4.1}$$

■

Property 4.2.
We consider that adding a processor introduces a parallelism cost, i.e., the area of a task increases with the parallelism as expressed by Equation 4.2.

$$\forall v < w, C_i(v) \times v \leqslant C_i(w) \times w \tag{4.2}$$

■

As presented by Goossens and Berten [GB10], the Gang task family can be split in three sub-families given by Definitions 4.6 and 4.7.

Definition 4.6 (Rigid, Moldable and Malleable Job [GB10])**.**
A job of a Gang parallel task is said to be:

Rigid if the number of processors assigned to this job is specified externally to the scheduler a priori, and does not change throughout its execution.

Moldable if the number of processors assigned to this job is determined by the scheduler, and does not change throughout its execution.

Malleable if the number of processors assigned to this job can be changed by the scheduler at runtime.

■

Definition 4.7 (Rigid, Moldable and Malleable Recurrent Task [GB10])**.**
A periodic/sporadic parallel Gang task is said to be:

Rigid if all its jobs are rigid, and the number of processors assigned to the jobs is specified externally to the scheduler. Notice that a rigid task does not necessarily have jobs with the same size. For instance, if the user/application decides that odd instances require v processors, and even instances v' processors, the task is said to be rigid.

Moldable if all its jobs are moldable.

Malleable if all its jobs are malleable.

■

4.2.1 Metrics for Gang task sets

A task set composed of Gang tasks is also characterized by some metrics. We define in this section various metrics and we give some evident constraints.

Utilization The utilization of a Gang task τ_i scheduled on v processors is given by Equation 4.3.

$$U_{\tau_i}(v) \stackrel{\text{def}}{=} \frac{C_i(v)}{T_i} \tag{4.3}$$

Density The density of a Gang task τ_i scheduled on v processors is given by Equation 4.4.

$$\Lambda_{\tau_i}(v) \stackrel{\text{def}}{=} \frac{C_i(v)}{\min(D_i, T_i)} \tag{4.4}$$

With Berten, Courbin, and Goossens [BCG11], we give some trivial results on the feasibility of a Gang task set:

Unfeasible A Gang task set composed of n tasks is not feasible if the sum of the utilization of each task scheduled on one processor is greater than the number m of identical reference processors in the platform. This condition is expressed in Equation 4.5.

$$\sum_{i=1}^{n} U_{\tau_i}(1) > m \tag{4.5}$$

Feasible A Gang task set composed of n tasks is feasible if the sum of the utilization of each task scheduled on the m identical reference processors of the platform is lower than 1. Indeed, in this case we always give the m processors to all the jobs (i.e., only one job is running at any time instant), and the schedule is then equivalent to a uniprocessor problem which can be scheduled by **E**arliest **D**eadline **F**irst (EDF) scheduler if Equation 4.6 is valid.

$$\sum_{i=1}^{n} U_{\tau_i}(m) \leqslant 1 \tag{4.6}$$

4.3 Multi-Thread task model

In Subsection 2.2.2.3.2 we presented the Fork-Join task model which belongs to the Multi-Thread class. With Courbin, Lupu, and Goossens [CLG13], we proposed a new Multi-Thread task model named MPMT. This work was based on a previous development proposed by Lupu and Goossens [LG11] where the authors introduced a Multi-Thread task model composed of only one phase.

4.3.1 Multi-Phase Multi-Thread (MPMT) task model

In this section we define a new parallel task model of the Multi-Thread class called MPMT.

Definition 4.8 (Periodic parallel MPMT task set)**.**
Let $\tau_{(O,\Phi,T,D)} = \{\tau_1(O_1, \Phi_1, T_1, D_1), \ldots, \tau_n(O_n, \Phi_n, T_n, D_n)\}$ be a periodic parallel MPMT task set composed of n periodic parallel MPMT tasks. The task set $\tau_{(O,\Phi,T,D)}$ can be abbreviated as τ. A periodic parallel MPMT task $\tau_i(O_i, \Phi_i, T_i, D_i)$, abbreviated τ_i (Figure 4.2), is characterized by the 4-tuple (O_i, Φ_i, T_i, D_i) where:

- O_i is the *first arrival instant* of τ_i, i.e., the instant of the first activation of the task since the system initialization.
- Φ_i is a vector of the ℓ_i phases of τ_i such as $\Phi_i = \left(\phi_i^1, \ldots, \phi_i^{\ell_i}\right)$.

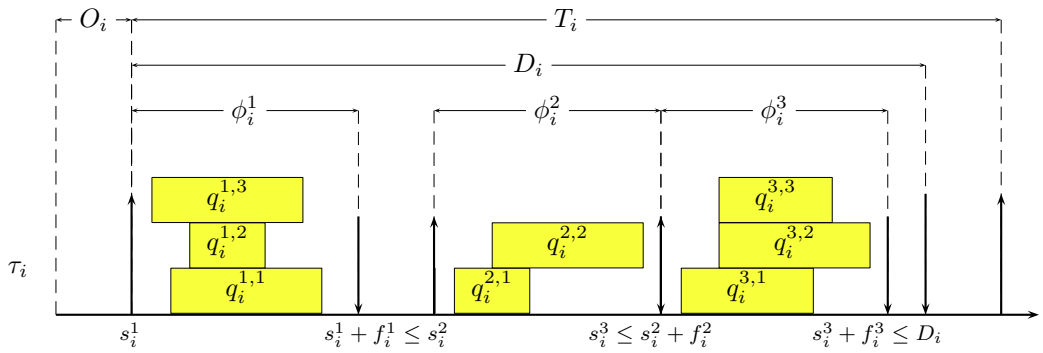


Figure 4.2 – Representation of a periodic parallel MPMT task, from Definition 4.8

- T_i is the *period* of τ_i, i.e., the exact inter-arrival time between two successive activations of τ_i.
- D_i is the *relative deadline* of τ_i, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant. Notice that we consider C-Deadline, so $D_i \leqslant T_i$.

A phase ϕ_i^j is characterized by a 3-tuple $\phi_i^j = \left(s_i^j, Q_i^j, f_i^j\right)$ where

- s_i^j is the *relative arrival offset* of the phase, i.e., for any arrival instant t of the task τ_i, the phase will be activated at time instant $t + s_i^j$.
- Q_i^j is the set of WCET of the v_i^j sub-programs of the phase ϕ_i^j such that $Q_i^j = \left\{q_i^{j,1}, \ldots, q_i^{j,v_i^j}\right\}$. At runtime these sub-programs generate threads which can be executed simultaneously, i.e., we allow task parallelism.
- f_i^j is the relative deadline of the phase.

■

In our work, the model is constrained as follows:

- $s_i^1 = 0$, i.e., the arrival instant of the first phase of the task corresponds to the arrival instant of the task itself.
- $\forall j > 1, s_i^j \geqslant s_i^{j-1} + f_i^{j-1}$, i.e., the relative arrival offset of a phase is larger than the deadline of the previous phase. In other words, we solve the precedence constraint between successive phases using relative arrival offsets and local deadlines. In the following, we will use $\forall j > 1, s_i^j = s_i^{j-1} + f_i^{j-1}$.
- $s_i^{\ell_i} + f_i^{\ell_i} \leqslant D_i$, i.e., the deadline of the last phase must not be larger than the deadline of τ_i. In the following, we will set the relative deadline $f_i^{\ell_i}$ such that $s_i^{\ell_i} + f_i^{\ell_i} = D_i$.

Remark 4.2. If $s_i^{\ell_i} + f_i^{\ell_i} < D_i$ or $\exists j > 2$ such that $s_i^j > s_i^{j-1} + f_i^{j-1}$ then a portion of the total available deadline D_i is not used by phases of task τ_i. Even if we do not use this possibility in this thesis, it is not necessary to restrict the model. As shown in Subsection 4.3.2, relative arrival offsets and relative deadlines of phases are not necessarily given at the beginning and could be fixed in order to guarantee the schedulability. For example, if we consider **D**eadline **M**onotonic (DM) as a **F**ixed **P**rocess **P**riority (FPP) scheduler (See Subsection 4.4.1 for definition), a lower deadline assigned to a phase will give a higher priority to the corresponding processes, so in some cases it would be useful to artificially reduce the value of D_i in order to increase the priorities of processes. ◇

4.3.1.1 Metrics, definitions and properties for MPMT task sets

A task set composed of MPMT tasks is also characterized by some metrics. We define in this section various metrics and we introduce some definitions and properties of our task model.

A MPMT task is characterized by the following metrics:

Utilization The utilization of a MPMT task τ_i is given by Equation 4.7.

$$U_{\tau_i} \stackrel{\text{def}}{=} \frac{\sum_{j=1}^{\ell_i} \sum_{k=1}^{v_i^j} q_i^{j,k}}{T_i} \tag{4.7}$$

Density The density of a MPMT task τ_i is given by Equation 4.8.

$$\Lambda_{\tau_i} \stackrel{\text{def}}{=} \frac{\sum_{j=1}^{\ell_i} \sum_{k=1}^{v_i^j} q_i^{j,k}}{\min(D_i, T_i)} \tag{4.8}$$

A MPMT task set is characterized by the following metrics:

Utilization The utilization of a task set τ composed of n MPMT tasks is given by Equation 4.9.

$$U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_{\tau_i} \tag{4.9}$$

Density The density of a task set τ composed of n MPMT tasks is given by Equation 4.10.

$$\Lambda_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \Lambda_{\tau_i} \tag{4.10}$$

Property 4.3 (Periodicity of sub-programs)**.**
Since tasks are periodic and phases have fixed relative arrival offsets, each sub-program has a periodic behaviour as well. ■

Property 4.4 (Independence of phases)**.**
Since for each phase the relative arrival offset is greater than or equal to the deadline of the previous phase, i.e., $\forall j > 1, s_i^j \geqslant s_i^{j-1} + f_i^{j-1}$*, and since tasks are periodic, each phase can be considered as independent from each other. In other words, if the scheduler respects all offsets, phases will be scheduled regardless the scheduling of the other phases.* ■

Property 4.5 (Independence of sub-programs)**.**
With Property 4.4 and since sub-programs of one phase are independent from each other (they generate threads which can be executed simultaneously), we can extend the property of independence to sub-programs. ■

4.3.1.2 Sub-program notation of the MPMT task model

It is important to notice that, according to the constraints on relative arrival offsets and relative deadlines of phases and since we deal with periodic tasks, each sub-program can be considered by the scheduler as an independent task. In this section we propose a new notation for sub-programs and define the sequential task set composed of these sub-programs.

Remark 4.3. In the following, according to the sequential task model presented in Subsection 2.2.2.2, we will use $\tau_i^{j,k}(O_i^{j,k}, C_i^{j,k}, T_i^{j,k}, D_i^{j,k})$ to represent a sub-program of τ_i. ◇

A sub-program is then characterized by $\tau_i^{j,k}(O_i + s_i^j, q_i^{j,k}, T_i, f_i^j)$. So a *periodic task with only one phase* and a first arrival instant equal to $O_i + s_i^j$. Notice that i represents the main task, $j \in [\![1; \ell_i]\!]$ represents the phase ϕ_i^j of the task τ_i and $k \in [\![1; v_i^j]\!]$ represents the WCET $q_i^{j,k}$ of the k^{th} sub-program of the phase ϕ_i^j of the task τ_i.

Since each sub-program is periodic (Property 4.3) and independent (Property 4.5) we can create a well known sequential task set given by Definition 4.9.

Definition 4.9.
Let $\tau_{(O,\Phi,T,D)} = \{\tau_1(O_1, \Phi_1, T_1, D_1), \ldots, \tau_n(O_n, \Phi_n, T_n, D_n)\}$ be a periodic parallel MPMT task set composed of n periodic parallel MPMT tasks as given by Definition 4.8.

Then, let $\tau^*_{(O,C,T,D)} = \{\tau_1^*(O_1^*, C_1^*, T_1^*, D_1^*), \ldots, \tau_r^*(O_r^*, C_r^*, T_r^*, D_r^*)\}$ be a periodic sequential task set composed of $r \stackrel{\text{def}}{=} \sum_{i=1}^{n} \sum_{j=1}^{\ell_i} v_i^j$ periodic independent sequential tasks as given by Definition 2.4. A periodic independent sequential task $\tau_s^* \in \tau^*$ is linked with a sub-program $\tau_i^{j,k} \in \tau$ and it is characterized by the 4-tuple $(O_s^*, C_s^*, T_s^*, D_s^*)$ where:

- $i \in [\![1; n]\!]$, $j \in [\![1; \ell_i]\!]$ and $k \in [\![1; v_i^j]\!]$, so τ_s^* corresponds to the k^{th} sub-program of the j^{th} phase of τ_i.
- O_s^* is the *first arrival instant* of τ_s^*, i.e., the instant of the first activation of the task since the system initialization. We have $O_s^* = O_i + s_i^{j,k}$.
- C_s^* is the *WCET* of τ_s^*, i.e., the maximum execution time required by the task to complete. We have $C_i^* = q_i^{j,k}$.
- T_s^* is the *period* of τ_s^*, i.e., the exact inter-arrival time between two successive activations of τ_s. We have $T_s^* = T_i$.
- D_s^* is the *relative deadline* of τ_s^*, i.e., the time by which the current instance of the task has to complete its execution relatively to its arrival instant. We have $D_s^* = f_i^{j,k}$.

■

4.3.2 Fork-Join to MPMT task model

Our task model presented in Definition 4.8 is based on the use of relative arrival offset and relative deadline for each phase of a task. Other task models of the Multi-Thread class define parallel task with multi-phase and multi-thread without these parameters. Fork-Join is an example of such task model and it is the most currently used. The purpose of this section is to allow a Fork-Join task set to use our results: Algorithm 7 shows how to translate a periodic Fork-Join task set (Definition 2.7) to our periodic MPMT task model.

Most tasks are not necessarily defined with a relative arrival offset or relative deadline to all its phases. This section explains how to attribute these parameters in order to obtain a schedulable task set using our task model.

First of all, let us define the vocabulary. In the Fork-Join task model some terms are used and could be translated to our model:

- "Thread" is equivalent to "Sub-program". In our model, a sub-program is the abstract (off-line) definition for which a thread could be seen as an instance (runtime).
- "Segment" is equivalent to "Phase". Notice that for this part, the Fork-Join task model is a specialization of our model since we do not impose an alternation between sequential and parallel phases.

Actually the Fork-Join task model is a particular case of MPMT one. Indeed the number of parallel sub-programs is the same for all parallel phases in the Fork-Join task model since this restriction is relaxed in our model. The WCET is identical for all sub-programs of one phase in the Fork-Join task model, again

this restriction is relaxed in ours. Finally, we could handle tasks with deadline not equal to their period ($D_i \neq T_i$) and we do not need a strict alternation of sequential and parallel phases.

A task $\tau_i(O_i, \{C_i^1, P_i^2, C_i^3, P_i^4, \ldots, P_i^{s_i-2}, P_i^{s_i-1}, C_i^{s_i}\}, T_i, V_i)$ defined with the Fork-Join task model is then translated to the MPMT task model as follows $\tau_i(O_i, \Phi_i, T_i, T_i)$ with:

- $\ell_i = s_i$, the number of phases is equal to s_i.
- $\forall j \in [\![1; \ell_i]\!]$ and j is an odd number, $v_i^j = 1$ and $q_i^{j,1} = C_i^j$, all odd phases are sequential with a WCET equal to C_i^j.
- $\forall j \in [\![1; \ell_i]\!]$ and j is an even number, $v_i^j = V_i$ and $\forall k \in [\![1; v_i^j]\!]$, $q_i^{j,k} = P_i^j$, all even phases are parallel with V_i sub-programs and each sub-program has a WCET equal to P_i^j.

Finally, the notation of subtasks $\tau_i^{j,k}$ has exactly the same signification in both models.

4.3.2.1 Compute relative arrival offsets and relative deadlines

A last thing is missing in the Fork-Join task model: phases parameters such as relative arrival offsets and relative deadlines. We propose Algorithm 7 to assign relative arrival offset and relative deadline to each phase of a task and test the schedulability at the same time. The main idea of the algorithm is to assign a relative deadline equal to the WCRT of the phase and set the same value as relative arrival offset of the next phase. Notice that Algorithm 7 could be used only with schedulers which do not need relative arrival offsets and relative deadlines of the phases to assign priorities, all priorities need to be known before the schedule.

As presented in Algorithm 7, we have to compute the WCRT for each sub-program of each phase of each task. At the beginning of the algorithm, sub-programs are not fully defined since relative arrival offsets and relative deadlines are not known. However, since we focus on schedulers which assign fixed priorities without taking into account relative arrival offsets and relative deadlines of phases (e.g. **F**ixed **T**ask **P**riority (FTP), **F**ixed **S**ub-program **P**riority (FSP) such as **L**ongest **S**ub-program **F**irst (LSF), (RM,LSF), etc. See Subsection 4.4.1 for definitions), the WCRT of a sub-program is affected only by sub-programs with higher priority. As a consequence, we can fulfil phases parameters in decreasing order of priority.

Remark 4.4. As we will see in the next sections, Schedulability Test 4.1 and Schedulability Test 4.2 give feasibility intervals for FSP and (FTP,FSP) schedulers respectively (See Subsection 4.4.1 for definitions fo schedulers). It is then simple

to compute the WCRT of a sub-programs by simulating the schedule on the corresponding feasibility interval, taking into account only higher priority sub-programs. For example, if tasks are sorted in decreasing order of priority, for (FTP,FSP) scheduler, the WCRT of sub-program $\tau_i^{j,k}$ is equal to the maximum response time of the corresponding threads during the schedule on the feasibility interval equals to $[0, S_i + P_i)$ with $P_i = \text{lcm}\{T_1, \ldots, T_i\}$ where S_i is defined by Equation 4.11. ◇

4.4 Schedulers for Multi-Thread P-Task

This section is dedicated to the presentation of the schedulers for Multi-Thread parallel RT tasks. The work described in this section has been originally presented by Lupu and Goossens [LG11]. It has been redeveloped in our joint publication with Courbin, Lupu, and Goossens [CLG13]. In this work we consider that the scheduling is *priority-driven*: the threads are assigned *distinct* priority levels. According to these priority levels the scheduler decides at each time instant t what is executed on the multiprocessor platform: the m highest (if any) priority threads will be executed simultaneously on the given platform. We also consider the following properties and notations for schedulers:

- The thread-processor assignment is *uni-vocally* determined by the following rule: "*higher the priority, lower the processor index*". If less than m threads are active, the processors with the higher indexes are left idle.
- We consider the *work-conserving* multi-thread scheduling: no processor is left idle while there are active tasks.
- We consider *pre-emptive* scheduling: a higher priority thread can interrupt the executing lower priority thread.

Notice that according to our task model, at time instant t, at most one phase of a process is active thanks to relative arrival offsets and relative deadlines of phases. So we do not care about priority between phases of a given task.

4.4.1 Taxonomy of schedulers

In this work we consider two classes of RT schedulers for our parallel task model: *Hierarchical schedulers* and *Global Thread schedulers*.

- At top-level *Hierarchical schedulers* manage processes with a process-level scheduling rule and use a second (low-level) scheduling rule to manage threads *within* each process.

Algorithm 7: Assign phases parameters and test schedulability

```
/* computeWCRT (τ, τ_i^{j,k}) return the WCRT of the k^th sub-program
   of the j^th phase of task τ_i in the task set τ. For FSP and
   (FTP,FSP) schedulers, Remark 4.4 gives a way to compute it.
   */
```

input : A task set τ with n tasks defined with a task model with multi-phase but without relative arrival offsets and relative deadlines such as Fork-Join task model

output : A boolean value which notify if a schedulable solution has been found and task set τ redefined with the MPMT task model presented in Definition 4.8

Data: s, f are integers

1 **foreach** $\tau_i \in \tau$, *higher priority first* **do**
  /* Relative arrival offset of the first phase is equal to 0 */
2 $s \leftarrow 0$;
3 **for** $j = 0$ **to** ℓ_i **do**
4 $f \leftarrow 0$;
5 **for** $k = 0$ **to** v_i^j **do**
6 $f \leftarrow \max\left(f, \texttt{computeWCRT}\left(\tau, \tau_i^{j,k}\right)\right)$;
7 **end for**
8 $s_i^{j,k} \leftarrow s$;
  /* Relative deadline is equal to the WCRT */
9 $f_i^{j,k} \leftarrow f$;
  /* Relative arrival offset of the next phase will be equal to this previous deadline */
10 $s \leftarrow s_i^{j,k} + f_i^{j,k}$;
11 **end for**
12 **if** $f_i^{j,v_i^j} \leqslant D_i$ **then**
13 $f_i^{j,v_i^j} \leftarrow D_i$;
14 **else**
15 **return** unSchedulable;
16 **end if**
17 **end foreach**
18 **return** Schedulable;

- *Global Thread schedulers* assign priorities to threads regardless of the task and sub-program that generated them.

In order to define rigorously our Hierarchical and Global Thread schedulers we have to introduce the following schedulers.

Definition 4.10 (**F**ixed **T**ask **P**riority (FTP))**.**
A fixed task priority scheduler assigns a fixed and distinct priority to each task before the execution of the system. At runtime each process priority corresponds to its task priority. ■

Among the FTP schedulers we can mention DM [LL73] and **R**ate **M**onotonic (RM) [Aud+91].

Definition 4.11 (**F**ixed **P**rocess **P**riority (FPP))**.**
A fixed process priority scheduler assigns a fixed and distinct priority to processes upon arrival. Each process preserves the priority level during its entire execution. ■

The EDF [LL73] scheduler is an example of FPP scheduler.

Definition 4.12 (**D**ynamic **P**rocess **P**riority (DPP))**.**
A dynamic process priority scheduler assigns, at each time instant t, priorities to the active processes according to their runtime characteristics. Consequently, during its execution, a process may have different priority levels. ■

The **L**east **L**axity **F**irst (LLF) scheduler is a DPP scheduler since the laxity is a dynamic process metric (see [Leu89; DM89] for details).

In the same vein, the following schedulers can be defined at thread level:

Definition 4.13 (**F**ixed **S**ub-program **P**riority (FSP))**.**
A fixed sub-program priority scheduler assigns a fixed and distinct priority to each sub-program before the execution of the system. At runtime each thread priority corresponds to its sub-program priority. ■

An example of FSP scheduler is the **L**ongest **S**ub-program **F**irst (LSF) scheduler.

Definition 4.14 (**F**ixed **Th**read **P**riority (FThP))**.**
A fixed thread priority scheduler assigns a fixed and distinct priority to threads upon arrival. Each thread preserves the priority level during its entire execution. ■

If we exclude FSP schedulers which can clearly be seen as FThP schedulers, and to the best of our knowledge, no FThP scheduler can be defined based *only* on the characteristics of the tasks in our model.

Definition 4.15 (**D**ynamic **Th**read **P**riority (DThP))**.**
A dynamic thread priority scheduler assigns, at time instant t, priorities to the existing threads according to their characteristics. During its execution, a thread may have different priority levels. ■

An example of DThP is LLF applied at thread level.

4.4.1.1 Hierarchical schedulers

Hierarchical schedulers are built following the next two steps:

1. at process level, one of the following schedulers is chosen in order to assign priorities to process: FTP, FPP and DPP.

2. for assigning priorities *within* process, one of the following schedulers will be chosen: FSP, FThP, DThP.

In the following an Hierarchical scheduler will be denoted by the couple (α, β), where $\alpha \in \{\text{FTP}, \text{FPP}, \text{DPP}\}$ and $\beta \in \{\text{FSP}, \text{FThP}, \text{DThP}\}$.

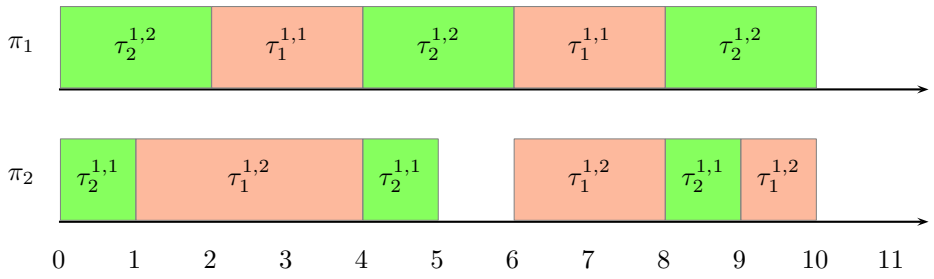


Figure 4.3 – Example of scheduler (RM,LSF)

An example of such a scheduler is presented in Figure 4.3. We consider the task set $\tau = \{\tau_1, \tau_2\}$ with $\tau_1(0, (\phi_1^1), 6, 6)$, $\phi_1^1 = (0, \{2, 3\}, 6)$ and $\tau_2(0, (\phi_2^1), 4, 4)$, $\phi_2^1 = (0, \{1, 2\}, 4)$ and the scheduler (RM,LSF) of the class (FTP,FSP). According to RM, τ_2 is the highest priority task. At sub-program level, LSF is applied and, consequently, $\tau_1^{1,2} \succ \tau_1^{1,1}$ and $\tau_2^{1,2} \succ \tau_2^{1,1}$.

4.4.1.2 Global thread schedulers

As Global Thread schedulers, the FSP, FThP and DThP schedulers can be applied to a set of sub-programs or threads regardless of the task that they belong to.

Notice that some Global Thread schedulers are identical to some hierarchical ones. For example, a total order between threads (i.e., a FThP scheduler) can "mimic" any hierarchical (FTP,FThP) scheduler.

An example of a Global Thread scheduler (LSF) is presented in Figure 4.4. The considered task set is the same as the one in Figure 4.3. The priority order at sub-program level according to LSF is the following: $\tau_1^{1,2} \succ \tau_2^{1,2} \succ \tau_1^{1,1} \succ \tau_2^{1,1}$ ($\tau_1^{1,1}$ and $\tau_2^{1,2}$ have the same execution time, but we choose to assign the highest priority to the sub-program belonging to the task with the smallest index).

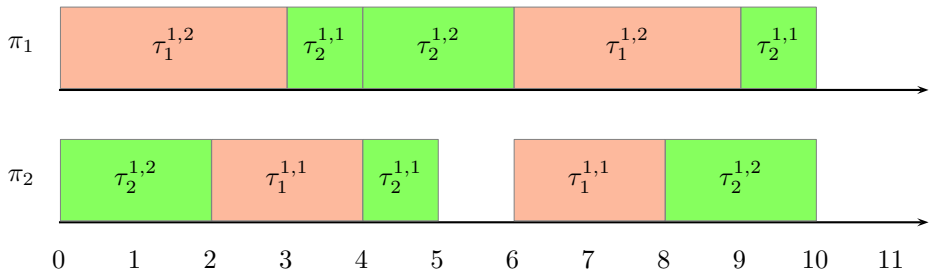


Figure 4.4 – Example of scheduler LSF

4.5 Schedulability analysis

4.5.1 MPMT tasks – schedulability NS-Test

In this section, we present two schedulability NS-Tests for our MPMT task model: one for FSP and one for (FTP,FSP) schedulers. We do not believe that these results are applicable to the other schedulers. This section is based on the work of Lupu and Goossens [LG11] where they proposed equivalent results for the mono-phase case. We extended their result to the multi-phase case in our joint publication with Courbin, Lupu, and Goossens [CLG13].

In the proposal, the schedulability NS-Test are based on *feasibility intervals* with the following definition.

Definition 4.16 (Feasibility interval)**.**
For any task set $\tau = \{\tau_1, \ldots, \tau_n\}$ and any multiprocessor platform, the *feasibility interval* is a finite interval such that if no deadline is missed while considering only the processes in this interval no deadline will ever be missed. ■

Our main contributions are schedulability NS-Tests for FSP scheduler (Subsection 4.5.1.1) and (FTP,FSP) scheduler (Subsection 4.5.1.2) used with parallel MPMT task set with C-Deadline. The two proofs follow the same logic: first we prove that the schedules are periodic, then we prove that the considered scheduler is predictable (Or, in other words, the considered scheduler is sustainable with respect to execution requirement. See Definition 4.17), finally we define the feasibility interval which gives the schedulability test.

4.5.1.1 FSP schedulability NS-Test

Since the scheduling of MPMT task are predictable (see Theorem 4.4) we know we have only to consider the worst-case scenario where the WCET is reached for each task/sub-program execution requirement. Consequently, in the following, we will assume that these execution requirements are constant.

The first step into defining the schedulability test for FSP schedulers is to prove that their schedules are periodic. The proof is based on the periodicity of

FTP schedules when the FTP is applied to task set τ' with the sequential task model (see Definition 4.9). The periodicity of FTP schedules for the sequential task model is stated in Theorem 4.1.

Theorem 4.1 (Periodicity of FTP schedules for sequential task set [CGG11])**.** *For any pre-emptive FTP scheduling algorithm $\mathcal{A}$, if an asynchronous C-Deadline periodic sequential task set $\tau' = \{\tau'_1, \ldots, \tau'_n\}$ with $\tau'_1 \succ \cdots \succ \tau'_n$ (task are ordered by decreasing priority) is $\mathcal{A}$-feasible, then the $\mathcal{A}$-schedule of τ' on a multiprocessor platform composed of m identical processors is periodic with a period of P starting from time instant S_n where S_i is defined as:*

$$\begin{cases} S_1 \stackrel{\text{def}}{=} O'_1, \\ S_i \stackrel{\text{def}}{=} \max\left\{O'_i, O'_i + \left\lceil \frac{S_{i-1} - O'_i}{T'_i} \right\rceil \times T'_i\right\}, \\ \qquad \forall i \in \{2, 3, \ldots, n\}. \end{cases} \tag{4.11}$$

(Assuming that the execution times of each task are constant.) ■

In the following, we consider the task set τ^* with $r \stackrel{\text{def}}{=} \sum_{i=1}^{n} \sum_{j=1}^{\ell_i} v_i^j$ sequential tasks (which correspond to parallel sub-programs) defined by Definition 4.9. A FSP scheduler is used to assign priorities to the r sub-programs. In the following we assume without loss of generality that sub-programs are ordered by FSP decreasing priority: $\tau_1^* \succ \cdots \succ \tau_r^*$.

Theorem 4.2.
For any pre-emptive FSP scheduling algorithm $\mathcal{A}$, if an asynchronous C-Deadline periodic MPMT task set $\tau = \{\tau_1, \ldots, \tau_n\}$ is $\mathcal{A}$-feasible, then the $\mathcal{A}$-schedule of τ on multiprocessor platform composed of m identical processors is periodic with a period of P starting from time instant S_r^*, with $r \stackrel{\text{def}}{=} \sum_{i=1}^{n} \sum_{j=1}^{\ell_i} v_i^j$ and $\forall s \in [\![1; r-1]\!], \tau_s^* \succ \tau_{s+1}^*$ (sub-programs are ordered by decreasing priority) where S_i^* is defined as follows:

$$\begin{cases} S_1^* \stackrel{\text{def}}{=} O_1^*, \\ S_s^* \stackrel{\text{def}}{=} \max\left\{O_s^*, O_s^* + \left\lceil \frac{S_{s-1}^* - O_s^*}{T_s^*} \right\rceil \times T_s^*\right\}, \\ \qquad \forall s \in \{2, 3, \ldots, r\}. \end{cases} \tag{4.12}$$

(Assuming that the execution times of each sub-program are constant.) ■

Proof. If we use FSP, a periodic parallel MPMT task set τ with n tasks and r sub-programs can be seen as the sequential task set τ^* which contains r periodic sequential tasks $\tau^* = \{\tau_1^*, \ldots, \tau_r^*\}$ given by Definition 4.9. From the FSP priority assignment on τ, a FTP priority assignment for τ^* can be defined: if $\tau_1^* \succ \cdots \succ \tau_r^*$ according to FSP, the corresponding sequential tasks have the same order according to FTP since a sub-program could be considered as a simple periodic sequential task.

By Theorem 4.1, we know that the schedule of FTP on τ^* is periodic with a period of P starting with S_r. We can observe that S_r has the same value as S_r^*. This means that the FSP schedule on τ is periodic with a period of P starting with S_r^*. □

Example We present an example for Theorem 4.2. We consider LSF as FSP scheduler, a multiprocessor platform composed of 2 processors and the task set $\tau = \{\tau_1, \tau_2\}$ with the following characteristics: $\tau_1(1, (\phi_1^1), 5, 5)$, $\phi_1^1 = (0, \{2\}, 5)$, $\tau_2(2, (\phi_2^1, \phi_2^2), 5, 5)$, $\phi_2^1 = (0, \{2, 1\}, 3)$ and $\phi_2^2 = (3, \{1\}, 2)$.

We define task set τ^* with $r = 4$ sequential tasks defined by Definition 4.9.

- $\tau_1^* = \tau_1^{1,1}(O_1^*, C_1^*, T_1^*, D_1^*) = \tau_1^{1,1}(O_1 + s_1^{1,1}, q_1^{1,1}, T_1, f_1^{1,1}) = \tau_1^{1,1}(1, 2, 5, 5)$,
- $\tau_2^* = \tau_2^{1,1}(O_2^*, C_2^*, T_2^*, D_2^*) = \tau_2^{1,1}(O_2 + s_2^{1,1}, q_2^{1,1}, T_2, f_2^{1,1}) = \tau_2^{1,1}(2, 2, 5, 3)$,
- $\tau_3^* = \tau_2^{1,2}(O_3^*, C_3^*, T_3^*, D_3^*) = \tau_2^{1,2}(O_2 + s_2^{1,2}, q_2^{1,2}, T_2, f_2^{1,2}) = \tau_2^{1,2}(2, 1, 5, 3)$,
- $\tau_4^* = \tau_2^{2,1}(O_4^*, C_4^*, T_4^*, D_4^*) = \tau_2^{2,1}(O_2 + s_2^{2,1}, q_2^{2,1}, T_2, f_2^{2,1}) = \tau_2^{2,1}(5, 1, 5, 2)$.

According to LSF, $\tau_1^* \succ \tau_2^* \succ \tau_3^* \succ \tau_4^*$ so $\tau_1^{1,1} \succ \tau_2^{1,1} \succ \tau_2^{1,2} \succ \tau_2^{2,1}$. We can now compute S_4^*:

- $S_1^* = O_1^* = 1$,
- $S_2^* = \max\left\{O_2^*, O_2^* + \left\lceil \frac{S_1^* - O_2^*}{T_2^*} \right\rceil \times T_2^*\right\} = \max\left\{2, 2 + \left\lceil \frac{1-2}{5} \right\rceil \times 5\right\} = 2$,
- $S_3^* = \max\left\{O_3^*, O_3^* + \left\lceil \frac{S_2^* - O_3^*}{T_3^*} \right\rceil \times T_3^*\right\} = \max\left\{2, 2 + \left\lceil \frac{2-2}{5} \right\rceil \times 5\right\} = 2$,
- $S_4^* = \max\left\{O_4^*, O_4^* + \left\lceil \frac{S_3^* - O_4^*}{T_4^*} \right\rceil \times T_4^*\right\} = \max\left\{5, 5 + \left\lceil \frac{2-5}{5} \right\rceil \times 5\right\} = 5$.

According to Theorem 4.2, we can conclude that the pre-emptive LSF schedule of τ on a multiprocessor platform composed of 2 processors is periodic with a period of $P \stackrel{\text{def}}{=} \text{lcm}\{T_1, T_2\} = 5$ starting from time instant $S_4^* = 5$. This conclusion is depicted in Figure 4.5.

Theorem 4.2 considers that execution times C_s^* of a sub-program τ_s^* $(1 \leqslant s \leqslant r)$ are *constant*. In order to define the schedulability test for the FSP schedulers, we have to prove that they are *predictable*.

Definition 4.17 (Predictability [HL94], or Sustainability with respect to execution requirement [BB06])**.**
Let us consider the thread sets J and J' which differ only with regards to their execution times: the threads in J have executions times less than or equal to the execution times of the corresponding threads in J'. A scheduling algorithm $\mathcal{A}$ is *predictable* if, when applied independently on J and J', a thread in J completes its execution before or at the same time instant as the corresponding thread in J'. ■

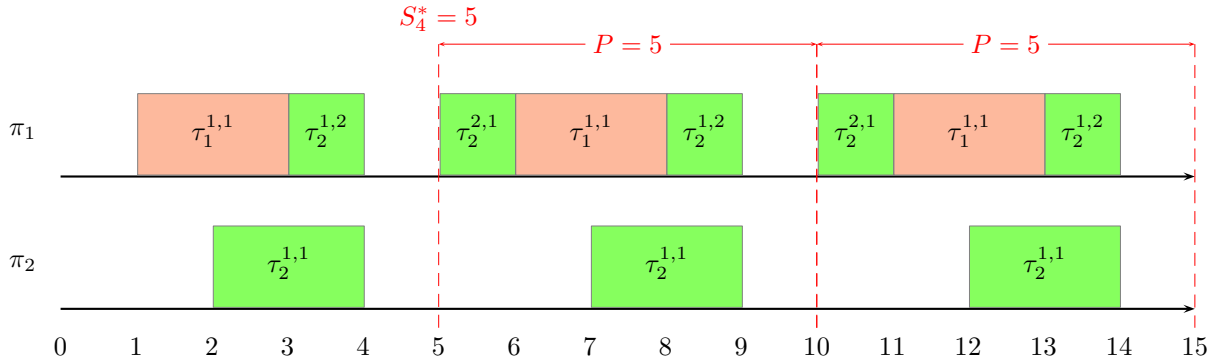


Figure 4.5 – Example of Theorem 4.2 with a LSF scheduler

Moreover, Ha and Liu [HL94] proved Theorem 4.3.

Theorem 4.3 ([HL94])**.**
Work-conserving and priority-driven schedulers are predictable for the sequential task model and identical multiprocessor platforms. ■

Using Theorem 4.3, we will prove that FSP schedulers are predictable.

Theorem 4.4.
FSP schedulers are predictable*.* ■

Proof. We mentioned in the proof of the Theorem 4.2 that the task set τ containing r sub-programs can be seen as a task set τ^* of r sequential tasks such that a task τ_s^* inherits the characteristics of the corresponding sub-program. A FTP priority assignment for τ^* can be built following the priorities assigned by FSP to the corresponding sub-programs in τ: $\tau_1^* \succ \cdots \succ \tau_r^*$.

By Theorem 4.3, FTP schedulers are predictable for sequential task sets like τ^* and on multiprocessor platforms composed of m processors. Since τ is equivalent to τ^* and the FTP scheduler assigns the same priorities to sequential tasks as FSP to the corresponding sub-programs, FSP schedulers are also predictable. □

Based on Theorems 4.2 and 4.4, we can define a schedulability NS-Test for FSP schedulers.

Schedulability Test 4.1.
For any pre-emptive FSP scheduler $\mathcal{A}$ and for any $\mathcal{A}$-feasible asynchronous C-Deadline periodic parallel MPMT task set $\tau = \{\tau_1, \ldots, \tau_n\}$ on a multiprocessor platform composed of m identical processors, $[0, S_r^* + P)$ is a feasibility interval, where S_r^* is defined by Equation 4.12. ■

Proof. This is a direct consequence of Theorems 4.2 and 4.4. □

4.5.1.2 (FTP,FSP) schedulability NS-Test

The first step in the definition of the schedulability NS-Test for the (FTP,FSP) schedulers is to prove the periodicity of the feasible schedules.

Theorem 4.5.
For any pre-emptive (FTP,FSP) scheduling algorithm $\mathcal{A}$, if an asynchronous C-Deadline periodic parallel MPMT task set $\tau = \{\tau_1, \ldots, \tau_n\}$ is $\mathcal{A}$-feasible, then the $\mathcal{A}$-schedule of τ on a multiprocessor platform composed of m identical processors is periodic with a period of P starting from time instant S_n, where S_n is defined by Equation 4.11 and tasks are ordered by decreasing priority: $\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n$. ■

Proof. Lets consider that the tasks in τ and their sub-programs are ordered by decreasing priority:

$$\tau_1 \succ \tau_2 \succ \cdots \succ \tau_n \text{ with } \forall i, 1 \leqslant i \leqslant n,$$
$$\tau_i^{1,1} \succ \cdots \succ \tau_i^{1,v_i^1} \succ \tau_i^{2,1} \succ \cdots \succ \tau_i^{2,v_i^2} \succ \cdots \succ \tau_i^{\ell_i,v_i^{\ell_i}}$$

Following these priority orders, we can define a FSP scheduler $\mathcal{A}'$ which assigns the following priorities to the $r \stackrel{\text{def}}{=} \sum_{i=1}^{n}\sum_{j=1}^{\ell_i} v_i^j$ sub-programs of τ:

$$\begin{aligned}
&\tau_1^{1,1} \succ \tau_1^{1,2} \succ \cdots \succ \tau_1^{1,v_1^1} \succ \tau_1^{2,1} \succ \tau_1^{2,2} \succ \cdots \succ \tau_1^{2,v_i^2} \succ \cdots \\
&\cdots \succ \tau_1^{\ell_1,1} \succ \tau_1^{\ell_i,2} \succ \cdots \succ \tau_1^{\ell_1,v_i^{\ell_1}} \succ \tau_2^{1,1} \succ \tau_2^{1,2} \succ \cdots \succ \tau_2^{1,v_2^1} \succ \cdots \\
&\cdots \succ \tau_{n-1}^{\ell_{n-1},1} \succ \cdots \succ \tau_{n-1}^{\ell_{n-1},v_{n-1}^{\ell_{n-1}}} \succ \tau_n^{1,1} \succ \cdots \succ \tau_n^{\ell_n,v_n^{\ell_n}}.
\end{aligned} \tag{4.13}$$

The FSP schedulers assign priorities to sub-programs regardless of the tasks they belong to. So we can rewrite Equation 4.13 regardless of the tasks $\tau_1, \ldots, \tau_n$:

$$\tau_1^* \succ \cdots \succ \tau_{v_1^1}^* \succ \cdots \succ \tau_{1+\sum_{j=1}^{\ell_1} v_1^j}^* \succ \cdots \succ \tau_{1+\sum_{i=1}^{n-1}\sum_{j=1}^{\ell_i} v_i^j}^* \succ \cdots \succ \tau_r^*.$$

By Theorem 4.2, the schedule generated by $\mathcal{A}'$ is periodic with a period of P from S_r^*. We can observe that the S_s^* quantity defined by Equation 4.12 represents the time instant of the first arrival of τ_s^* at or after time instant S_{s-1}^*. Since all the sub-programs belonging to the same phase of task τ_i $(1 \leqslant i \leqslant n)$ have the same activation times and the same periods and $\mathcal{A}'$ assigns consecutive priorities to the sub-programs of the same task (as seen in Equation 4.13):

$$\begin{aligned}
S_s^* = S_{s-1}^*, \text{ if } &\exists x \in [\![1; n]\!], y \in [\![1; \ell_x]\!] / \\
&\left(1 + \sum_{i=1}^{x}\sum_{j=1}^{y-1} v_i^j\right) < s \leqslant \sum_{i=1}^{x}\sum_{j=1}^{y} v_i^j
\end{aligned} \tag{4.14}$$

Furthermore, we can observe that $S_1^* = S_1 = O_1$. From this fact and Equation 4.14, we can conclude:

$$\begin{aligned} S_1^* = S_1 &\Rightarrow \\ S^*_{1+\sum_{j=1}^{\ell_1} v_1^j} = S_2 &\Rightarrow \\ &\vdots \\ S^*_{1+\sum_{i=1}^{n-1}\sum_{j=1}^{\ell_i} v_i^j} = S_n. & \end{aligned}$$

The $\mathcal{A}'$-schedule is then periodic with a period of P starting from S_n. Since the $\mathcal{A}'$-schedule is the same as the one generated by $\mathcal{A}$, the $\mathcal{A}$-schedule is also periodic with a period of P starting from S_n. □

We will now prove that the (FTP,FSP) schedulers are also predictable.

Theorem 4.6.
(FTP,FSP) schedulers are predictable. ■

Proof. Since based on any (FTP,FSP) scheduler we can define a FSP scheduler as shown in the proof of Theorem 4.5 and since, by Theorem 4.4, FSP schedulers are predictable, (FTP,FSP) schedulers are predictable as well. □

We will now define the schedulability NS-Test for (FTP,FSP) schedulers.

Schedulability Test 4.2.
For any pre-emptive (FTP,FSP) scheduler $\mathcal{A}$ and for any $\mathcal{A}$-feasible asynchronous C-Deadline periodic parallel MPMT task set $\tau = \{\tau_1, \ldots, \tau_n\}$ on a multiprocessor platform composed of m identical processors, $[0, S_n + P)$ is a feasibility interval, where S_n is defined by Equation 4.11. ■

Proof. This is a direct consequence of Theorem 4.5 and Theorem 4.6. □

4.5.2 MPMT tasks – WCRT computation

As presented in Algorithm 7, we have to compute the WCRT for each sub-program of each phase of each task. We recall that this algorithm has to be used with schedulers which assign fixed priorities without taking into account relative arrival offsets and relative deadlines of phases (e.g. FTP, FSP such as LSF, (RM,LSF), etc. See Subsection 4.4.1 for definitions). In the next section we present new results for the computation of WCRT of such subtasks.

Notation

We now define each specific notations used along this section.

As proposed by Guan et al. [Gua+09], we use the following notation to express that A as lower (respectively upper) bound B (respectively C) such as $[\![A]\!]_B = \max(A, B)$ (respectively $[\![A]\!]^C = \min(A, C)$). By extension, we have $[\![A]\!]_B^C = [\![[\![A]\!]_B]\!]^C$. This expression keeps the value of A if it is within the interval $[B, C]$, otherwise it returns B if $A < B$ or C if $A > C$.

In this section, we consider that tasks and subtasks are sorted in decreasing order of priority so the relation $\forall i < i',\ \tau_i \succ \tau_{i'}$ indicates that τ_i has a higher priority than $\tau_{i'}$. Moreover we use hierarchical schedulers so we have to define the priority relation between subtasks of the same task. If the relation $\tau_i^{j,k} \succ \tau_{i'}^{j',k'}$ indicates that $\tau_i^{j,k}$ has a higher priority than $\tau_{i'}^{j',k'}$, we defined priorities as follow:

$$
\begin{aligned}
&\tau_i^{j,k} \succ \tau_{i'}^{j',k'} \text{ if and only if} \\
&i < i' \\
&\textit{or } i = i' \text{ and } j < j' \\
&\textit{or } i = i' \text{ and } j = j' \text{ and } k < k'
\end{aligned}
$$

Some explanations Tasks τ_i with $i < i'$ have already been defined as higher priority than $\tau_{i'}$. Subtasks $\tau_i^{j,k}$ with $j < j'$ correspond to a subtask of phase which precede the phase of $\tau_i^{j',k}$ and $\tau_i^{j',k}$ can not execute while $\tau_i^{j,k}$ is not completed so it has to be lower priority. Finally, we consider that sub-programs of each tasks are sorted in decreasing order of priority so $\tau_i^{j,k} \succ \tau_i^{j,k'}\ \forall k < k'$.

4.5.2.1 The sporadic case - A new upper bound

In this section, we present our results to compute an *upper bound* of WCRT for sporadic parallel MPMT tasks based on the task model presented in Definition 4.8. Our results are based on the work of Guan et al. [Gua+09]. We summarize their results in Subsection 4.5.2.1.1 and present our adaptation in Subsection 4.5.2.1.2.

4.5.2.1.1 Previous work Guan et al. [Gua+09] propose an improvement of existing bound for WCRT of sequential mono-phase independent tasks on multiprocessor platforms.

They define sporadic tasks as given by Definition 2.5. They consider C-Deadline tasks.

Based on this model, they study the upper bound of the **workload** of a task in order to know the maximum possible **interference** produced by an higher-priority task within an interval.

Notice that, in order to compute the WCRT of a task τ_p, we have to study the maximum continuous time interval during which each processor executes

higher priority tasks until τ_p completes its job. This interval is also known as the *level-p busy period*. For uniprocessor case, we know the worst case activation scenario such as this interval is maximum. Indeed, the maximal interference is produced when all higher priority tasks and τ_p are activated at the same time instant. Conversely, in the multiprocessor case, the worst case activation scenario is unknown and since we cannot test all possible activation scenarios, we have to compute an upper bound of the interference, giving an upper bound for the WCRT.

Workload The workload $W(\tau_i, [a;b])$ of a task τ_i in an interval $[a;b]$ is the length of the accumulated execution time of that task within the interval $[a;b]$. As presented by Guan et al. [Gua+09], the workload of a task could be of two types: with a **C**arry **I**n (CI) job or without a carry-in job (**N**on **C**arry-in (NC)). A carry-in task refers to a task with one job with arrival instant earlier than the interval $[a;b]$ and deadline in the interval $[a;b]$. For both cases, they prove that the worst case scenarios are given by Figures 4.6 and 4.7 and computed using Lemma 4.1 where:

- $W^{\mathrm{NC}}(\tau_i, x)$ denotes the workload bound if τ_i does not have a carry-in job in the interval of length x.
- $W^{\mathrm{CI}}(\tau_i, x)$ denotes the workload bound if τ_i has a carry-in job in the interval of length x.

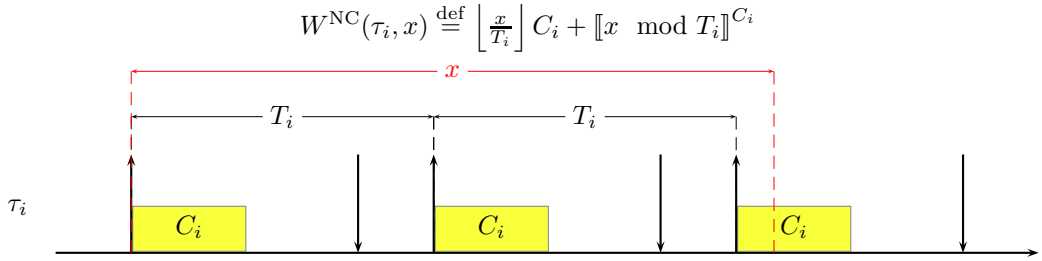


Figure 4.6 – Computing $W^{\mathrm{NC}}(\tau_i, x)$

Lemma 4.1 ([Gua+09])**.**
The workload bounds can be computed with

$$W^{\mathrm{NC}}(\tau_i, x) \stackrel{\mathrm{def}}{=} \left\lfloor \frac{x}{T_i} \right\rfloor C_i + [\![x \mod T_i]\!]^{C_i}$$

$$W^{\mathrm{CI}}(\tau_i, x) \stackrel{\mathrm{def}}{=} \left\lfloor \frac{[\![x - C_i]\!]_0}{T_i} \right\rfloor C_i + [\![C_i]\!]^x + \alpha$$

where $\alpha = [\![[\![x - C_i]\!]_0 \mod T_i - (T_i - R_i)]\!]_0^{C_i - 1}$
and R_i *is the WCRT of* τ_i*.* ■

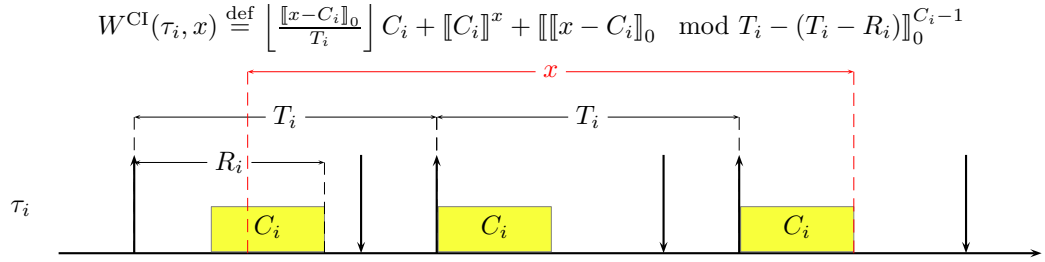


Figure 4.7 – Computing $W^{\mathrm{CI}}(\tau_i, x)$

Knowing an upper bound of the workload of each task, then they study the maximum possible **interference** suffered by a task in an interval of length x.

Interference The interference $I_p(x)$ on a task τ_p over an interval of length x is the total time during which τ_p is ready but blocked by the execution of at least m higher priority tasks on the platform. $I_p(\tau_i, x)$ is the total time during which task τ_p is ready but could not be scheduled on any processor while the higher priority task τ_i is executing.

Since we consider pre-emptive fixed priority schedulers, we have to notice that $\forall \tau_j \prec \tau_p,\ I_p(\tau_j, x) = 0$, i.e., all lower priority tasks does not produce interference on a higher priority task.

We now need to derive a computation of the interference of an higher priority task on τ_p. First of all, according to Bertogna and Cirinei [BC07], a task can interfere only when it is executing, which gives Theorem 4.7.

Theorem 4.7 ([BC07]).
The interference $I_p(\tau_i, x)$ of a task τ_i on a task τ_p in an interval of length x cannot be higher than the workload $W(\tau_i, x)$. ■

In the same paper, Bertogna and Cirinei [BC07] demonstrate an improvement of this upper bound. Since R_p is the response time of τ_p, nothing can interfere on τ_p for more than $R_p - C_p$. Using this assertion with Lemma 4.2 they proved Theorem 4.8

Lemma 4.2 ([BCL05]).
For any global scheduling algorithm it is

$$I_p(x) \geqslant y \iff \sum_{i \neq k} \min\left(I_p(\tau_i, x), y\right) \geqslant m \times y$$

■

Theorem 4.8 ([BC07]).
A task τ_p has a response time upper bounded by R_p^{ub} if

$$\sum_{i \neq k} \min\left(I_p\left(R_p^{ub}\right), R_p^{ub} - C_p + 1\right) < m \times \left(R_p^{ub} + 1\right)$$

■

Indeed, if Theorem 4.8 is verified for task τ_p then, according to Lemma 4.2, we have:

$$I_p\left(R_p^{ub}\right) < \left(R_p^{ub} - C_p + 1\right)$$

and task τ_p will be interfered for at most $R_p^{ub} - C_p$ time units so τ_p will complete its execution at most at time instant R_p^{ub}.

Finally, according to Bertogna and Cirinei [BC07], using Theorems 4.7 and 4.8 we could get the improved Equation 4.15 for the interference of τ_i on τ_p in an interval of length x.

$$I_p(\tau_i, x) \stackrel{\text{def}}{=} [\![W(\tau_i, x)]\!]_0^{x-C_p+1} \tag{4.15}$$

If we consider the computation of the workload presented in the previous part, we have to define two different interferences, one for a non carry-in task (Equation 4.16) and one for a task with a carry-in job (Equation 4.17).

$$I_p^{\text{NC}}(\tau_i, x) \stackrel{\text{def}}{=} [\![W^{\text{NC}}(\tau_i, x)]\!]_0^{x-C_p+1} \tag{4.16}$$

$$I_p^{\text{CI}}(\tau_i, x) \stackrel{\text{def}}{=} [\![W^{\text{CI}}(\tau_i, x)]\!]_0^{x-C_p+1} \tag{4.17}$$

We are now able to estimate the interference of one specific higher priority task on τ_p. We need to merge these results to get the total interference produced by all higher priority tasks on τ_p. A naive response would be to compute the sum of the interference of all higher priority tasks τ_i and taking for each one the maximum value between $I_p^{\text{NC}}(\tau_i, x)$ and $I_p^{\text{CI}}(\tau_i, x)$. However Guan et al. [Gua+09], based on a work from Baruah [Bar07], prove that there are at most $m-1$ tasks having a carry-in job, and for each task τ_i, the carry-in is at most $C_i - 1$. Therefore if we consider all higher priority tasks of τ_p and select from them at most $m-1$ carry-in tasks for which $\left(I_p^{\text{CI}}(\tau_i, x) - I_p^{\text{NC}}(\tau_i, x)\right)$ is positive and maximum, the remaining tasks will be non carry-in (NC). We then obtain Lemma 4.3.

Lemma 4.3 ([Gua+09])**.**
If τ^{CI} is the subset of at most $m-1$ higher priority tasks τ_i with respect to τ_p such as $\left(I_p^{\text{CI}}(\tau_i, x) - I_p^{\text{NC}}(\tau_i, x)\right)$ is positive and maximum and if τ^{NC} is the subset of the remaining higher priority tasks with respect to τ_p, we define the total interference $I_p(x)$ as

$$I_p(x) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau^{\text{NC}}} I_p^{\text{NC}}(\tau_i, x) + \sum_{\tau_i \in \tau^{\text{CI}}} I_p^{\text{CI}}(\tau_i, x) \tag{4.18}$$

■

Upper bound of WCRT Since they are able to compute an upper bound of the total interference produced by all higher priority tasks on τ_p, GUAN et al. [Gua+09] prove Theorem 4.9

Theorem 4.9 (OUR-RTA [Gua+09])**.**
Let R_p^{ub} be the minimal solution of the following Equation 4.19 by doing an iterative fixed point search of the right hand side starting with $x = C_p$.

$$x = \left\lfloor \frac{I_p(x)}{m} \right\rfloor + C_p \tag{4.19}$$

Then R_p^{ub} is an upper bound of τ_p's WCRT. ■

4.5.2.1.2 Adaptation to MPMT tasks A naive approach would be to get all subtasks as independent tasks and use Theorem 4.9 without further reflections. The result would be valid and we would obtain a real upper bound of the WCRT of each subtask. However, we propose to refine this result taking into account the precedence relation between subtasks. Indeed, if we analyse the workload of a subtask using a specific activation (carry-in or non carry-in), the activation of all other subtasks of the same task is accordingly defined.

In this section, we define the workload of an individual subtask and we deduce from it the workload of the entire associated task.

Computation of the workload of a subtask The workload bound of a subtask $\tau_i^{j,k}$ over an interval of length x can be computed according to Lemma 4.4, with $q_i^{j,k}$ the WCET of the subtask and $R_i^{j,k}$ its WCRT.

Lemma 4.4.
The workload bounds can be computed with

$$W^{\mathrm{NC}}\left(\tau_i^{j,k}, x\right) \stackrel{\text{def}}{=} \left\lfloor \frac{x}{T_i} \right\rfloor q_i^{j,k} + [\![x \mod T_i]\!]^{q_i^{j,k}}$$

$$W^{\mathrm{CI}}\left(\tau_i^{j,k}, x\right) \stackrel{\text{def}}{=} \left\lfloor \frac{[\![x - q_i^{j,k}]\!]_0}{T_i} \right\rfloor q_i^{j,k} + [\![q_i^{j,k}]\!]^x + \left[\!\left[[\![x - q_i^{j,k}]\!]_0 \mod T_i - \left(T_i - R_i^{j,k}\right) \right]\!\right]_0^{q_i^{j,k}-1}$$

■

In the following, we will use the same equations for the computation of workload of other subtasks. The precedence relation will be taken into account in the length of the interval considered. Let us take an example: if the subtask $\tau_i^{j,k}$ starts at the beginning of the interval x (non carry-in job), any subtask of the same phase are activated at the same time instant and any subtask $\tau_i^{j+1,k'}$ of the next phase will start $s_i^{j+1} - s_i^j$ later. Therefore, if we consider that the workload of $\tau_i^{j,k}$ must be computed as $W^{\mathrm{NC}}\left(\tau_i^{j,k}, x\right)$ then the workload of $\tau_i^{j+1,k'}$ is easily $W^{\mathrm{NC}}\left(\tau_i^{j,k}, x - (s_i^{j+1} - s_i^j)\right)$.

In the next paragraphs we study the workload of a task considering one specific phase as the reference of activation. Let us define:

- $W^{\mathrm{J,NC}}(\tau_p, x)$ the workload of the task τ_p if ϕ_p^J the J^{th} phase is activated as a non carry-in task.
- $W^{\mathrm{J,CI}}(\tau_p, x)$ the workload of the task τ_p if ϕ_p^J the J^{th} phase is activated as a carry-in task.

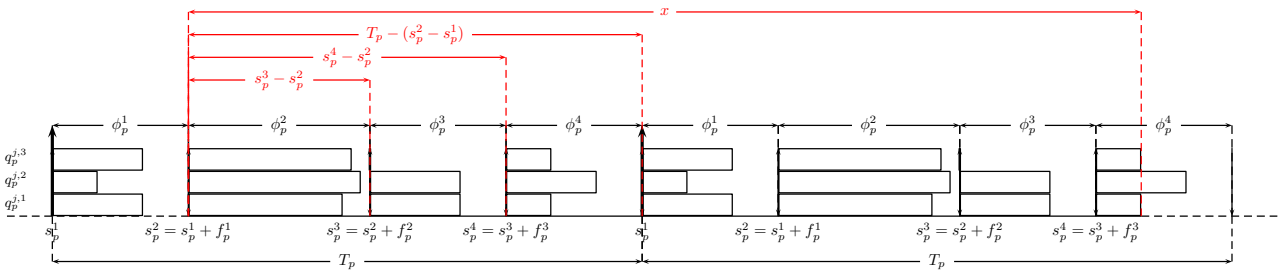


Figure 4.8 – Example of computation for $W^{2,\mathrm{NC}}(\tau_p, x)$, phase ϕ_p^2 is a non carry-in task, so it is activated at the beginning of the interval of length x

Computation of $W^{\mathrm{J,NC}}(\tau_p, x)$ In this paragraph we determine the length of the studied interval for each phase (so, each subtask) of a non carry-in task τ_p considering that ϕ_p^J the J^{th} phase is activated at the beginning of the interval. See Figure 4.8 for an example with $J = 2$.

If the J^{th} phase is activated at the beginning of the interval of length x, then:

- the next phases ($j > J$) are activated $\left(s_p^j - s_p^J\right)$ later, so the considered interval is $x - \left(s_p^j - s_p^J\right)$.
- the previous phases ($j < J$) are activated $T_p - \left(s_p^J - s_p^j\right)$ later, so the considered interval is $x - \left(T_p - (s_p^J - s_p^j)\right)$.

We then deduce Lemma 4.5.

Lemma 4.5.
The workload bound of the non carry-in task τ_p considering ϕ_p^J as the first activated phase in the interval of length x can be computed with

$$W^{\mathrm{J,NC}}(\tau_p, x) \stackrel{\mathrm{def}}{=} \sum_{j=1}^{J-1} \sum_{k=1}^{v_p^j} W^{\mathrm{NC}}\left(\tau_p^{j,k}, \left[\!\left[x - \left(T_p - (s_p^J - s_p^j)\right)\right]\!\right]_0\right) + \\ + \sum_{j=J}^{\ell_p} \sum_{k=1}^{v_p^j} W^{\mathrm{NC}}\left(\tau_p^{j,k}, \left[\!\left[x - (s_p^j - s_p^J)\right]\!\right]_0\right)$$

■

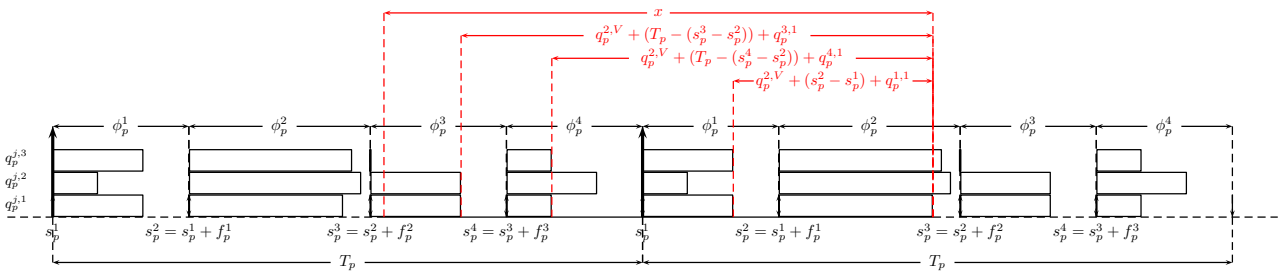


Figure 4.9 – Example of computation for $W^{2,\mathrm{CI}}(\tau_p, x)$, phase ϕ_p^2 is a carry-in task, so its last activation is $q_p^{2,V} = \min_{v=1,\ldots,v_p^{2,v}} q_p^{2,v}$ before the end of interval x

Computation of $W^{\mathrm{J,CI}}(\tau_p, x)$ In this paragraph we determine the length of the studied interval for each phase (so each subtask) of a carry-in task τ_p considering that ϕ_p^J, the J^{th} phase, has a carry-in job.

In this case, the idea is slightly more complicated. Indeed, according to Figure 4.7, ϕ_p^J does not start at the beginning of the interval of length x but its last activation completes exactly at the end of this interval. For example the last activation of subtask $\tau_p^{J,k}$ must be exactly $q_p^{J,k}$ before the end of the interval. The problem is that sub-programs of a phase could have different values of WCET. If we arbitrary choose the sub-program in order to determine the last activation of the phase, we may be pessimist or do an error. The best approach would be to test all possibilities but it would be time consuming. Due to the complexity of this approach, we coose to explore a sub-optimal approach: the sub-program with the minimal value of WCET will be selected to determine the scenario of activation fixing the last activation of its phase, so $q_p^{J,V} = \min_{v=1,\ldots,v_p^{J,v}} q_p^{J,v}$. By doing so we allow a maximum time to the other phases to run. See Figure 4.9 for an example with $J = 2$.

If the J^{th} phase has a carry-in job in the interval of length x, then:

- for the next phases ($j > J$), each subtask $\tau_p^{j,v}$ has to be considered on an interval of length $\left[\!\left[x - \left(\left[\!\left[q_p^{J,V}\right]\!\right]^x + \left(T_p - (s_p^j - s_p^J)\right) + q_p^{j,v}\right)\right]\!\right]_0$ with $q_p^{J,V} = \min_{v=1,\ldots,v_p^{J,v}} q_p^{J,v}$.
- for the previous phases ($j < J$), each subtask $\tau_p^{j,v}$ has to be considered on an interval of length $\left[\!\left[x - \left(\left[\!\left[q_p^{J,V}\right]\!\right]^x + (s_p^J - s_p^j) + q_p^{j,v}\right)\right]\!\right]_0$ with $q_p^{J,V} = \min_{v=1,\ldots,v_p^{J,v}} q_p^{J,v}$.

We then deduce Lemma 4.6.

Lemma 4.6.
The workload bound of the carry-in task τ_p considering that ϕ_p^J has a carry-in job in the interval of length x can be computed with

$$W^{\mathrm{J,CI}}(\tau_p, x) \stackrel{\text{def}}{=} \sum_{j=1}^{J-1} \sum_{k=1}^{v_p^j} W^{\mathrm{CI}}\left(\tau_p^{j,k}, \left[\!\left[x - \left(\left[\!\left[q_p^{J,V}\right]\!\right]^x + (s_p^J - s_p^j) + q_p^{j,v}\right)\right]\!\right]_0\right) +$$
$$+ \sum_{j=J}^{\ell_p} \sum_{k=1}^{v_p^j} W^{\mathrm{CI}}\left(\tau_p^{j,k}, \left[\!\left[x - \left(\left[\!\left[q_p^{J,V}\right]\!\right]^x + \left(T_p - (s_p^j - s_p^J)\right) + q_p^{j,v}\right)\right]\!\right]_0\right)$$

with $q_p^{J,V} = \min_{v=1,\ldots,v_p^{J,v}} q_p^{J,v}$. ■

Computation of $W^{\mathrm{NC}}(\tau_p, x)$ and $W^{\mathrm{CI}}(\tau_p, x)$ Finally, a bound of the workload generated by a task τ_p is given by Theorem 4.10.

Theorem 4.10.
The workload bounds of a sporadic parallel MPMT task given by Definition 4.8 can be computed with

$$W^{\mathrm{NC}}(\tau_i, x) \stackrel{\text{def}}{=} \max_{J=1,\ldots,l_p} W^{\mathrm{J,NC}}(\tau_i, x)$$

$$W^{\mathrm{CI}}(\tau_i, x) \stackrel{\text{def}}{=} \max_{J=1,\ldots,l_p} W^{\mathrm{J,CI}}(\tau_i, x)$$

■

Proof. This a direct consequence of Lemma 4.4 and Lemma 4.5 for $W^{\mathrm{NC}}(\tau_i, x)$ and Lemma 4.6 for $W^{\mathrm{CI}}(\tau_i, x)$. □

The WCRT is then computed using Theorem 4.9, Equations 4.16, 4.17 and Lemma 4.3 from Guan et al. [Gua+09] to get the total interference.

4.5.2.2 The periodic case - An exact value

Since each phase has to receive its own deadline and offset (the period is the same as the one of the original task), we can consider that the task set τ is composed of a number of mono-phase tasks (τ_i^j is the corresponding mono-phase task of the ϕ_i^j phase) for which we have to establish the values of the deadline and offset parameters.

We know from Goossens and Berten [GB10] that for mono-phase parallel RT tasks, $[0, S_n + P)$ is a feasibility interval . We can determine the WCRT of each phase by building the schedule of τ for the given time interval. The maximum response time obtained for a given phase in $[0, S_n + P]$ becomes its local deadline and the offset of the next phase of the same task.
The schedule for $[0, S_n + P)$ is build as follows:

- the first phase of highest priority task (τ_1) is assigned the needed processors at arrival ($O_1 + \alpha T_i$, $\alpha \geqslant 0$) in $[0, S_n + P)$; its maximum response time in this time interval becomes the local deadline ($f_1^1 = R_1^1$) of the phase and the offset ($s_1^2 = f_1^1$) of the second phase of the task.

- the second phase of τ_1 is assigned the needed processors at the time instant $O_1 + s_1^2 + \alpha T_i$ ($\alpha \geqslant 0$); the maximum response time of the phase in the time interval becomes its local deadline f_1^2 and the offset $s_1^3 = f_1^2 + s_1^2$ of the next phase, etc.

- when the assignation of the first task is completed, we start assigning the phases of the second one, etc.

4.6 Scheduling Gang tasks versus Multi-Thread tasks

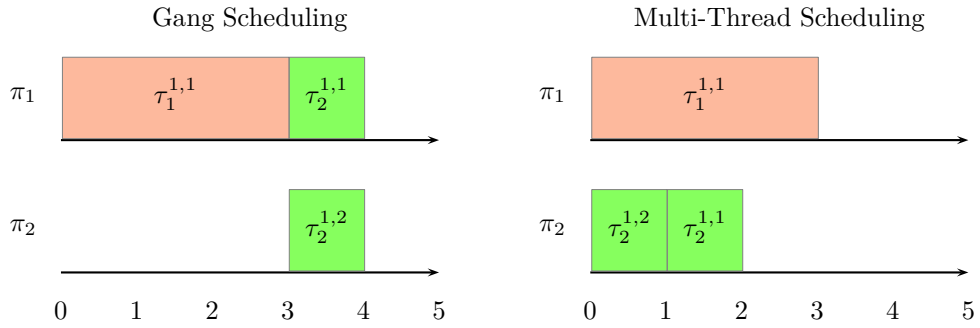


Figure 4.10 – Gang scheduling versus Multi-Thread scheduling

Figure 4.10 illustrates a Gang and a Multi-Thread scheduling for the "same" task set $\tau = \{\tau_1, \tau_2\}$: $\tau_1(0, (\phi_1^1), T_1, D_1)$, $\phi_1^1 = (0, \{3\}, D_1)$ (i.e., $q_1^{1,1} = 3$), $\tau_2 = (0, (\phi_2^1), T_2, D_2)$, $\phi_2^1 = (0, \{1, 1\}, D_2)$ (i.e., $q_2^{1,1} = 1$ and $q_2^{1,2} = 1$). In our case, $\tau_1 \succ \tau_2$. Focusing on τ_2, Gang scheduling has to manage the rectangle $\max\left(q_2^{1,1}, q_2^{1,2}\right) \times v_2^1 = 1 \times 2$ while Multi-Thread scheduling has to manage two 1-unit length threads.

From our point of view, we present the respective advantages of Multi-Thread and Gang scheduling seen from the schedulability angle.
Advantages of Gang scheduling:

1. The scheduling seems to be easiest to understand since we need to schedule rectangles in a two dimensions space (time and processors).

2. For tasks with a frequent need of communications between its threads, it seems to be easiest to consider threads by groups instead of decomposing the task in a large number of phases.

Advantages of Multi-Thread scheduling:

1. Multi-Thread scheduling does not suffer from *priority inversion*. As shown by Goossens and Berten [GB10], Gang scheduling suffer from priority inversion, i.e., a lower priority task can progress while an higher priority active task cannot.

2. The number of processors required by a task can be larger than the platform size.

3. An idle processor can always be used if a thread is ready. With Gang scheduling, because of the requirement that the task must execute on exactly v processors simultaneously very often many processors may be left idle while there is active tasks.

4. Last but not least, Multi-Thread FTP schedulers are proven *predictable* in Subsection 4.5.1. On the other hand, as shown by Goossens and Berten [GB10], Gang FTP schedulers are not predictable.

4.6.1 Gang DM and (DM,IM) scheduling are incomparable

In this section we show that Gang FTP and Multi-Thread hierarchical (FTP,FSP) schedulers may be incomparable — in the sense that there are task sets which are schedulable using Gang scheduling approaches and not by Multi-Thread scheduling approaches, and conversely. The result described in this section has been originally presented by Lupu and Goossens [LG11] and redeveloped in our joint publication with Courbin, Lupu, and Goossens [CLG13].

The considered FTP scheduler is DM [Aud+91]: the priorities assigned to tasks by DM are inversely proportional to the relative deadlines. The FSP scheduler is called **I**ndex **M**onotonic (IM) and it assigns priorities as follows: *the lower the index of the sub-program within the task, the higher the priority.*

In the following examples, the task offsets are equal to 0 and the feasible schedules are periodic from 0 with a period of P (according to Theorem 4.5 and the work of Goossens and Berten [GB10]).

First example This first example presents a task set that is unschedulable by Gang DM, but schedulable by (DM,IM) on a multiprocessor platform composed of 2 processors. The tasks is the set $\tau = \{\tau_1, \tau_2, \tau_3\}$ have the following characteristics: $\tau_1(0, (\phi_1^1), 3, 3)$, $\phi_1^1 = (0, \{2\}, 3)$, $\tau_2(0, (\phi_2^1), 4, 4)$, $\phi_2^1 = (0, \{3\}, 4)$ and $\tau_3(0, (\phi_3^1), 12, 12)$, $\phi_3^1 = (0, \{2, 2\}, 12)$. According to DM $\tau_1 \succ \tau_2 \succ \tau_3$.

We can observe (see Figure 4.11) that according to Gang DM, task τ_3 has to wait for 2 available processors simultaneously in order to execute. This is the

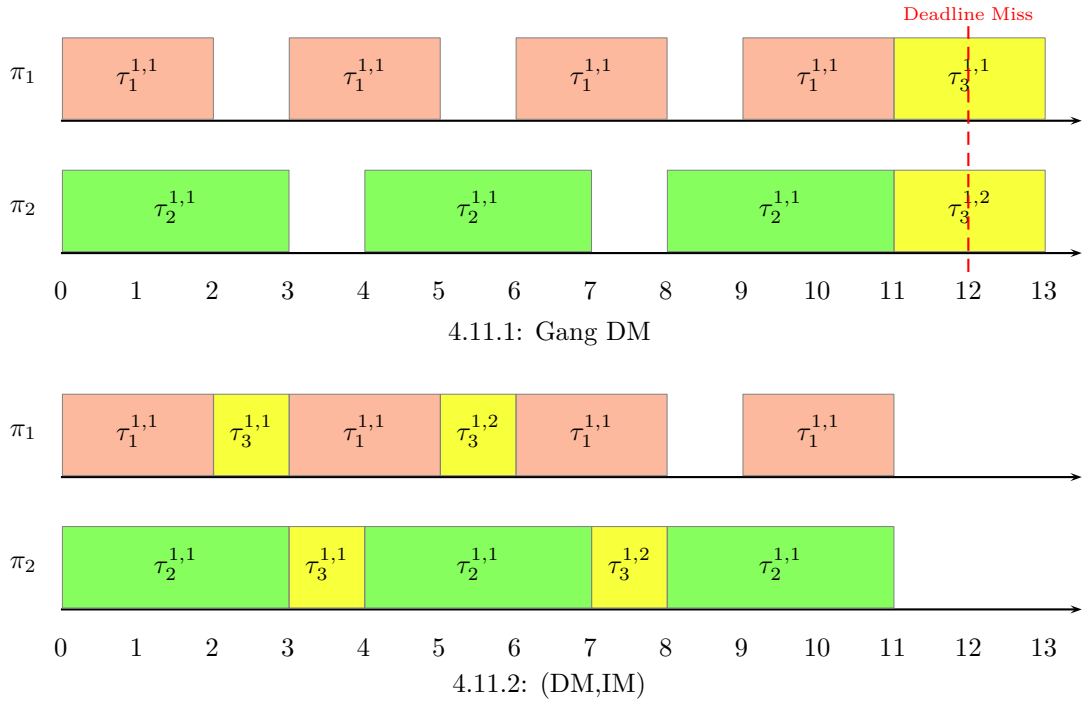


Figure 4.11 – Gang DM unschedulable, (DM,IM) schedulable

case at time instant 11; though, at time instant 12 the task has uncompleted execution demand and it misses its deadline.

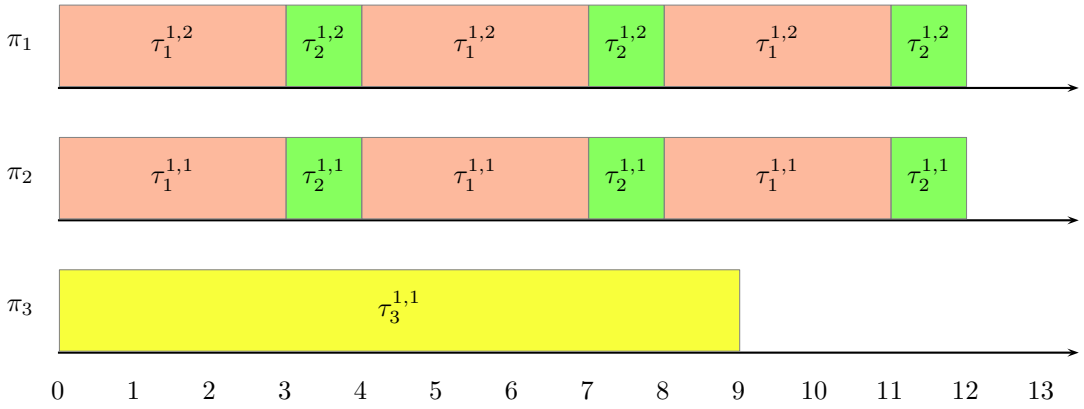
In the case of (DM,IM), τ_1 and τ_2 execute at the same time instants and on the same processors as in Gang DM. The difference is that τ_3 can start executing its first process at time instant 2 since one processor is available. Taking advantage of the fact that the processors are left idle by τ_1 and τ_2 at some time instants, the first process of τ_3 (which is the only τ_3 process in the interval $[0, 12)$) completes execution at time instant 8. No deadline is missed, therefore, the system is schedulable by (DM,IM).

Second example The second example presents a task set $\tau = \{\tau_1, \tau_2, \tau_3\}$ which is schedulable with Gang DM, but unschedulable with (DM,IM) on a multiprocessor platform composed of 3 processors. The tasks in τ have the following characteristics: $\tau_1(0, (\phi_1^1), 4, 4)$, $\phi_1^1 = (0, \{3, 3\}, 4)$, $\tau_2(0, (\phi_2^1), 5, 5)$, $\phi_2^1 = (0, \{1, 1\}, 5)$ and $\tau_3(0, (\phi_3^1), 10, 10)$, $\phi_3^1 = (0, \{9\}, 10)$. According to DM $\tau_1 \succ \tau_2 \succ \tau_3$.

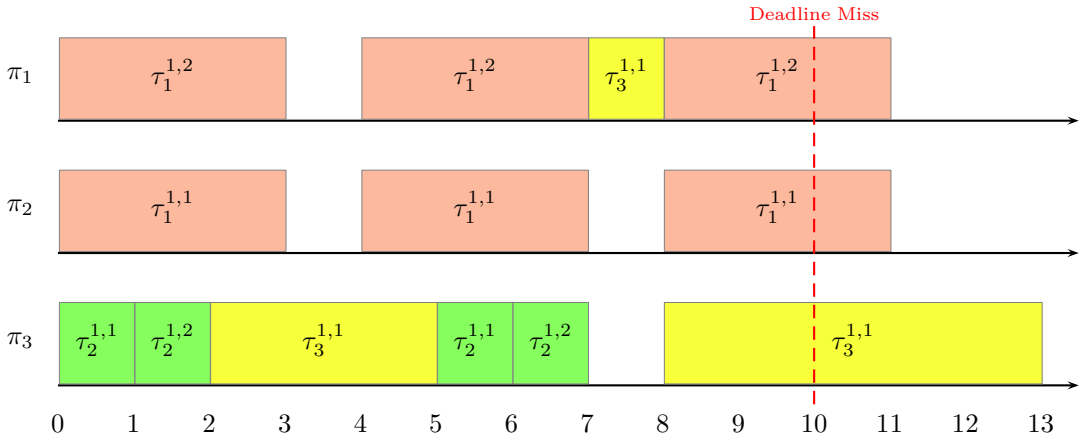
In Figure 4.12, we can observe that according to Gang DM, at time instant 0, τ_1 is assigned to 2 of the 3 processors in the platform. Since there is only one processor left, τ_2 cannot execute, therefore τ_3 starts its execution on the third processor. At time instant 3, 2 processors are available and, consequently, τ_2 may start executing, etc. No deadline is missed in the time interval $[0, 12)$, therefore

the system is schedulable with Gang DM.

According to (DM,IM), even if τ_1 occupies 2 processors of the 3 in the platform, τ_2 may start executing on the third a first thread from time instant 0 to 1. The second thread of its first process will execute on the third processor from time instant 1 to 2. We can conclude that τ_3 misses its deadline at time instant 10 since it has 9 units of execution demand and only 6 time units available *until* its deadline.



4.12.1: Gang DM



4.12.2: (DM,IM)

Figure 4.12 – Gang DM schedulable, (DM,IM) unschedulable

Therefore, Gang DM and (DM,IM) allow scheduling of different task sets but we see in our empirical study that this Multi-Thread scheduler appears to successfully schedule more task sets than this Gang scheduler.

4.7 Gang versus Multi-Thread task models evaluation

The purpose of this empirical study is to evaluate the performance of Multi-Thread schedulers compared with the one of Gang schedulers. More specifically, the chosen Multi-Thread scheduler is (DM,IM) from the (FTP,FSP) scheduler type. Among the Gang schedulers, we consider Gang DM from the Gang FTP scheduler type. The work described in this section has been originally presented by Lupu and Goossens [LG11]. It has been redeveloped in our joint publication with Courbin, Lupu, and Goossens [CLG13].

Since Gang FTP schedulers are not predictable (see Section 4.6), in this study we consider constant execution times. From the work of Goossens and Berten [GB10] and the Schedulability Test 4.2, we know that we have to simulate both Gang FTP and (FTP,FSP) schedulers in the time interval $[0, S_n + P)$ in order to conclude if the task set is schedulable with one of them, with both of them or unschedulable.

Since Gang schedulers consider that the execution requirement of processes corresponds to a "$C_i \times V_i$" rectangle, we will consider MPMT tasks composed of only one phase. Moreover, the execution times of all the sub-programs of a task are considered to be equal. Notice that in this context, a MPMT task is equivalent to a Fork-Join task.

4.7.1 Conditions of the evaluation

We present in this section the conditions of our evaluation. First of all, we make explicit the criteria used to compare the solutions and we explain the methodology applied to generate the task sets so that anyone could reproduce our results. About the platform, we considered identical multiprocessor platform containing 4, 8 and 16 processors.

4.7.1.1 Evaluation criteria

Gang DM and (DM,IM) are evaluated according to the following criteria:

- *Success Ratio* is defined with Equation 4.20. For instance, it allows us to determine which algorithm schedules the largest number of task sets.

$$\frac{\text{number of task sets successfully scheduled}}{\text{total number of task sets}} \tag{4.20}$$

- The WCRT of the lowest priority task in the system. The WCRT shows how a lower priority task is impacted by higher priority tasks. This value is used to measure how the scheduler influences the impact of higher priority

tasks on the other. We therefore chose to look only at the lowest priority task to measure the total impact of all other tasks.

In practical terms, if task set τ is schedulable, the WCRT of a task $\tau_i \in \tau$ for Gang DM and (DM,IM) are calculated according to its processes within the time interval $[0, S_n + P)$. For each schedulable task set with both Gang DM and (DM,IM) we compare the WCRT of the lowest priority task ($WCRT_{GangDM}$ and $WCRT_{(DM,IM)}$ respectively). For a given system utilization, we count separately the task sets where $WCRT_{GangDM}$ is *strictly* inferior to $WCRT_{(DM,IM)}$ and conversely. Consequently, the uncounted task sets are those where the computed WCRT are equal for the two schedulers. For example, the value of this criterion for the case $WCRT_{GangDM} < WCRT_{(DM,IM)}$ is computed with Equation 4.21.

$$\frac{\text{number of scheduled task sets with } WCRT_{GangDM} < WCRT_{(DM,IM)}}{\text{total number of task sets scheduled by } GangDM \text{ and } (DM, IM)} \tag{4.21}$$

Each criterion is presented in two ways. Firstly using a graph of the values as a function of the utilization of task sets. Secondly using a table with an aggregate performance metric known as *Weighted criterion* (Definition 4.18) derived from the *Weighted schedulability* proposed by Bastoni, Brandenburg, and Anderson [BBA10]. This metric reduces the obtained results to a single number which sums up the comparison.

Definition 4.18 (Weighted criterion [BBA10])**.**
Let $S(U) \in [0, 100]$ denote the considered criterion for a given U and let Q denote a set of evenly-spaced utilization gaps (*e.g.*, $Q = \{1.0, 1.2, 1.4, ..., m\}$). Then *weigthted criterion* W is defined as

$$W \stackrel{\text{def}}{=} \frac{\sum_{U \in Q} \left(U \times \frac{S(U)}{100} \right)}{\sum_{U \in Q} U}$$

■

4.7.1.2 Task set generation methodology

The procedure for task set generation is the following: individual tasks are generated and added to the task set until the total system utilization exceeds the platform capacity (m).

The characteristics of a task τ_i are integers and they are generated as follows:

1. the period T_i is uniformly chosen within the interval $[1; 250]$,
2. the offset O_i is uniformly chosen within the interval $[1; T_i]$,

3. the utilization U_{τ_i} is inferior to m and it is generated using the following distributions:

   - uniform distribution within the interval $[1/T_i; m]$,
   - bimodal distribution: light tasks have an uniform distribution within the interval $[1/T_i; m/2]$, heavy tasks have an uniform distribution within the interval $[m/2; m]$; the probability of a task being heavy is of $1/3$,
   - exponential distribution of mean $m/4$,
   - exponential distribution of mean $m/2$,
   - exponential distribution of mean $3 \times m/4$.

4. the number of thread $V_i = v_i^1$ is uniformly chosen within the interval $[\![1; m]\!]$. We only care about v_i^1 since we consider mono-phase tasks,

5. since we consider that all the sub-programs of a task τ_i have equal execution times, it is sufficient to compute a single execution time value: $C_i = q_i^{1,k} = U_{\tau_i} \times T_i / v_i^1, \ \forall k \in [\![1; v_i^1]\!]$,

6. the deadline D_i is uniformly chosen within the interval $[C_i; T_i]$.

We decided to reduce the time granularity (the minimum possible value of each parameter) to 1. Thus, for each task τ_i, its parameters C_i, T_i and D_i are considered as integers. Considering that the values are discretized according to the clock tick, it is always possible to modify all the parameters to integer values by multiplying them by an appropriate factor. To simplify testing, we used this approach and all the parameters are limited to integer values. This does not imply, however, that the algorithms used and presented in this evaluation cannot be applied to non-integer values.

We use several distributions (with different means) in order to generate a wide variety of task sets and, consequently, to have more accurate simulation results.

The generated task sets have a least common multiple of the task periods bounded by 5×10^6 (each task set with a larger value is deleted and replaced by an other until this constraint is respected). A total of 450×10^3 task sets were generated.

4.7.2 Results

4.7.2.1 Success Ratio

Figures 4.13.1–4.13.3 contain 3 plots: one represents the percent of task sets scheduled by (DM,IM) multi-thread scheduler, a second one the percent of task sets scheduled by Gang DM and a third one expresses the percent of task sets

scheduled by both of them. Table 4.2 gives the weighted criterion values for schedulability study.

Figures 4.13.1–4.13.3 show that the performance gap between the two schedulers is growing as the number of processors grows. We observe the same behaviour in Table 4.2 where the difference between the weighted criteria of the two schedulers constantly increases with the number of processors; this difference is equal to 0.03 on a 4 processors platform, 0.04 on 8 processors and 0.05 on 16 processors.

We can also verify that (DM,IM) and Gang DM are incomparable since the plot representing the task sets successfully scheduled by the two schedulers is below the others. Moreover the amount of *additional* task sets that Multi-Thread scheduling can manage is quite higher (the difference between plots "(DM,IM)" and "both" is higher than the difference between plots "Gang DM" and "both"). For example, in the case of 4 processors platform, 50% of the task sets are unschedulable according to Gang DM at a utilization level of 2.4 ($= 1.67m$); however, using (DM,IM), approximatively 50% of the task sets are schedulable at a utilization level of 2.5 ($= 1.60m$). Hence, in this case, (DM,IM) enables 4.2% better utilization of the processing resource than Gang DM. In the case of 16 processors platforms, Gang DM schedules 50% of the tasks set at a utilization level of 8.2 ($= 1.95m$) while (DM,IM) schedules the same amount at a utilization level of 9.1 ($= 1.76m$). This difference corresponds to an increase in usable processing capacity of around 11%.

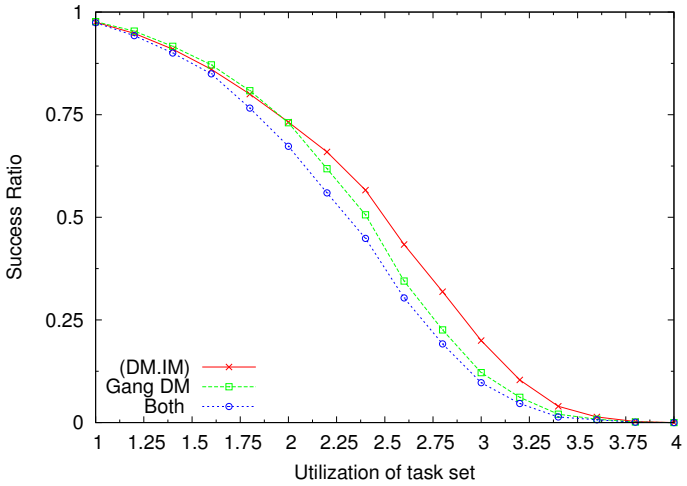
Notice that we generate our tasks with a number of threads which can be equal to the number of processors on the platform since v_i^1 is uniformly chosen within the interval $[\![1; m]\!]$. Therefore, the results show the capacity of the scheduler to take advantage of the whole platform. Our results confirm that (DM,IM) has an advantage versus Gang DM in this context. As presented in the advantages of Multi-Thread scheduling in Section 4.6, it can be explained by the fact that Gang schedulers require v_i^1 processors to be simultaneously idle to start task τ_i while Multi-Thread schedulers can always use an idle processor if a thread is ready.

	(DM,IM)	**Gang DM**
4 **processors**	0.34	0.31
8 **processors**	0.29	0.25
16 **processors**	0.28	0.23

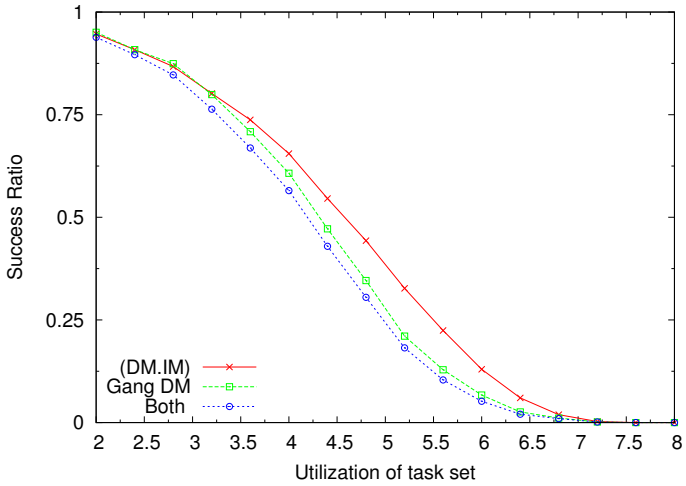
Table 4.2 – Weighted criterion for schedulability study from Figures 4.13.1–4.13.3

4.7.2.2 WCRT of the lowest priority task

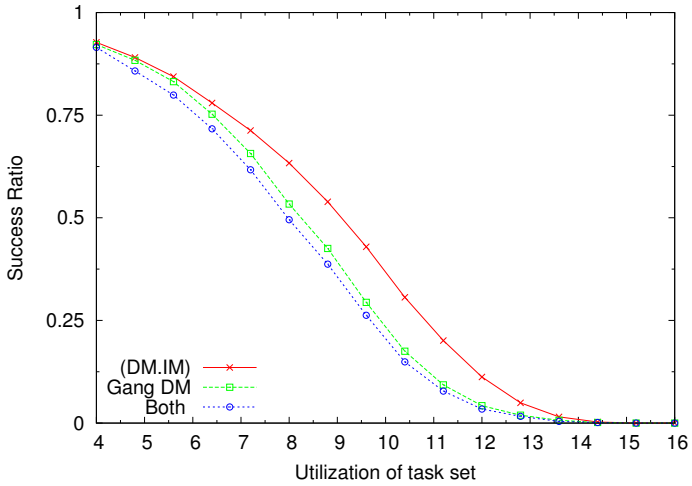
In the following we will reference the Figures 4.14.1–4.14.3. The utilization of the considered systems in this part of the study is greater than 25% and less to 90%



4.13.1: 4 processors



4.13.2: 8 processors



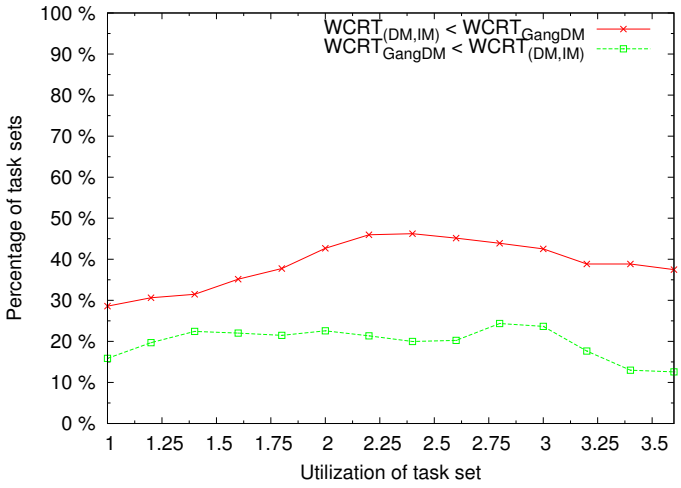
4.13.3: 16 processors

Figure 4.13 – Success Ratio analysis

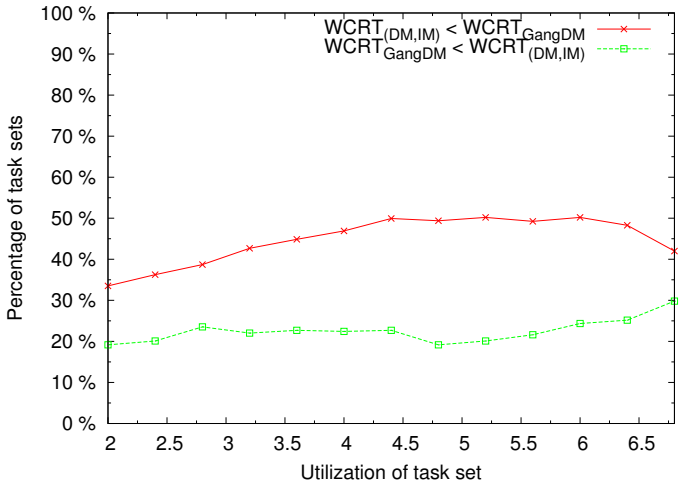
of the platform capacity since we focus only on task sets schedulable by both Gang DM and (DM,IM) schedulers. In each figure, there are two plots: one that marks the portion of task sets where $WCRT_{(DM,IM)}$, the (DM,IM) WCRT of the lowest priority task, is *strictly* inferior to $WCRT_{GangDM}$, the one computed under Gang DM; a second plot marks the contrary behaviour. Table 4.3 gives the weighted criterion values for WCRT study.

It is clear from Figures 4.14.1–4.14.3 that (DM,IM) outperforms Gang DM on 4, 8 and 16 identical multiprocessor platforms in this context. Table 4.3 shows the same results with values which are at least two times higher using the (DM,IM) scheduler.

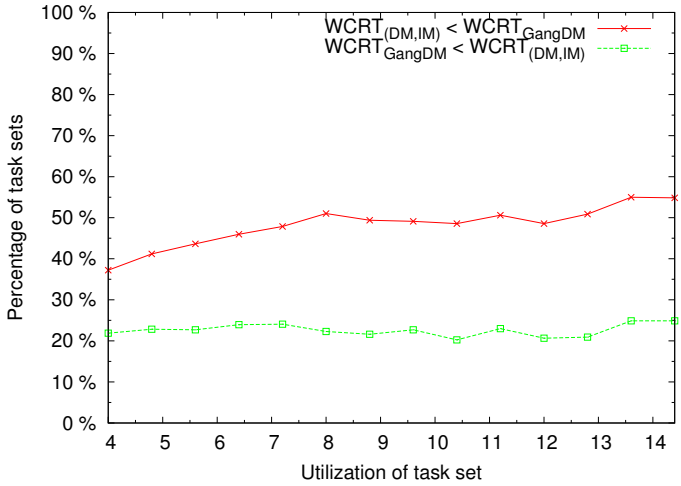
As previously, we observe that Multi-Thread schedulers is at an advantage compared with Gang schedulers since it can start a thread without waiting for v_i^1 processors to be idle. This clearly allows reducing the WCRT of the tasks.



4.14.1: 4 processors



4.14.2: 8 processors



4.14.3: 16 processors

Figure 4.14 – WCRT analysis

	$WCRT_{(DM,IM)} < WCRT_{GangDM}$	$WCRT_{GangDM} < WCRT_{(DM,IM)}$
4 processors	0.40	0.19
8 processors	0.46	0.23
16 processors	0.50	0.23

Table 4.3 – Weighted criterion for WCRT study from Figures 4.14.1–4.14.3

4.8 Summary

In this chapter we considered the Multi-Thread scheduling for parallel **R**eal-**T**ime (RT) systems. We introduce a new task model, **M**ulti-**P**hase **M**ulti-**T**hread (MPMT) task model, which belongs to the Multi-Thread class. The main advantage of this class is that it does not require *all* threads of a same task to execute simultaneously as Gang scheduling does.

We defined in this chapter several types of priority-driven schedulers dedicated to our parallel task model and scheduling method. We distinguished between Hierarchical schedulers (that firstly assign distinct priorities at task set level and secondly, within each task) and Global Thread schedulers (that do not take into account the original tasks when priorities are assigned at thread level).

We proposed the MPMT task model in order to rectify the negative result revealed by Lupu and Goossens [LG11] which stated that "multi-phase multi-thread Hierarchical schedulers are not predictable". With relative arrival offsets and relative deadlines assigned to each phase, we were able to define predictable Hierarchical scheduler and Global Thread scheduler for this task model. Indeed, we showed that, contrary to Gang **F**ixed **T**ask **P**riority (FTP), the Hierarchical and Global Thread schedulers based on FTP and **F**ixed **S**ub-program **P**riority (FSP) are predictable. Based on this property and the periodicity of their schedules, we defined two exact schedulability tests. We also explained how adapt a task set defined by the well known Fork-Join task model into MPMT task model in order to take advantage of our results.

Finally, even though the Gang and Multi-Thread schedulers are, as we have shown, incomparable, the empirical study confirmed the intuition that Multi-Thread scheduling outperforms Gang scheduling. In terms of success ratio, the performance gap increases as the number of processors grows.

Part III

Tools for real-time scheduling analysis

Chapter 5

Framework fOr Real-Time Analysis and Simulation

Codez toujours en pensant que celui qui maintiendra votre code est un psychopathe qui connait votre adresse.

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

John F. Woods [Woo91]

Contents

5.1 Introduction

We observe a growing importance of multiprocessors architectures including multi-core systems addressed by the field of **R**eal-**T**ime (RT) scheduling. These architectures have brought a lot of questions to this area, and its own set of answers: algorithms, techniques, optimizations etc.

Many solutions have been proposed by the community to meet this challenge: either for pre-emptive or for non-pre-emptive scheduling, with fixed or dynamic priority scheduling, based on a **G**lobal **Scheduling** (G-Scheduling), **P**artitioned **Scheduling** (P-Scheduling) or **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approaches.

Everyone develops his idea, discovers many advantages and would like to share with the community so that it can use, understand the tricks and possibly be inspired in order to improve the original idea. But when comes the time to test the solution and present it, we are often faced to a problem: on what basis can we compare? Too few common ways of generating sets of tasks used for simulations or common way to implement other solutions are existing.

The tool presented in this chapter does not respond to each questions, but it proposes a perfectible, extensible, open and scalable solution for these concerns. A minimalist **G**raphical **U**ser **I**nterface (GUI) is available for those who want easy access and basic usage. The code is open and offered to those who want complete control and specific results. This chapter presents some functionalities of this tool.

Section 5.2 presents some other tools and our motivation are given in Section 5.3. Then, we present the four main options identified as the most common: **test** a scheduling algorithm (Section 5.4), **observe** a scheduling (Section 5.5), **generate** tasks and sets of tasks (Section 5.6) and **edit** and run an evaluation of performance comparison (Section 5.7). A final section is devoted to summary and future work related to this tool (Section 5.8).

5.2 Existing tools

Several tools, commercial or free, are already available to study RT systems. On the commercial side, the goal is usually an analysis and a complete design of a particular system. Examples are TimeWiz (TimeSys Corp.) or RapidRM (Tri-Pacific Software Inc.) based on the **R**ate **M**onotonic **A**nalysis (RMA) methodology. On the other hand, free projects proposed by the academic community generally respond to specific needs and are not always flexible or even maintained. For projects still in development, we can cite MAST [Har+01] which proposes a set of tools to analyse and represent the temporal and logical elements of RT systems. Cheddar [Sin+04] mainly focuses on theoretical methods of RT scheduling and

proposes a simulator and the majority of existing schedulability tests. STORM [UDT10] defines the hardware platform and software as an XML file and then conduct simulations on scheduling. Others tools like RESCH [KRI09], Grasp [HBL10] or Litmus^{RT} [Cal+06] can analyse the practical operation of a RT scheduling on a real system such as μC/OS-II and Linux. Finally, YARTISS [Cha+12] provide an interesting modular tool in Java with the special feature of considering the energy state as a scheduling constraint in the same manner as the **W**orst **C**ase **E**xecution **T**ime (WCET).

Each tool provides a valuable aid for the analysis of RT systems. However, it seemed that almost all of them focus on the analysis or design of a given scheduling: given my platform, or even my task set, what will be the performance or how do I have to change my system to ensure its schedulability?

Framework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation (FORTAS) implements some of these elements but often remains far less advanced than existing tools. However, it focuses on the possibility to compare and evaluate scheduling algorithms, whether based on a theoretical analysis of feasibility/schedulability or on the simulation of scheduling, without necessarily focusing on a given platform or a specific task set.

5.3 Motivation for FORTAS

The tools proposed by the community do not exactly correspond to our needs. Especially, we needed to evaluate and compare algorithms based on analytical tests and not simulations. We also needed to have a simple GUI to use this tool for teaching purposes. We certainly do not meet all the needs in this area but we simply want to offer and make our work available.

Thus, this tool was first developed for analytical testing and evaluation. The Java programming language was chosen for its development efficiency and proven interoperability. Each part of the program is conceived as a module and an effort of abstraction was given to each element. Thus, the GUI is completely interchangeable and can be redeveloped by anyone without coming to interfere with the core program. Similarly, a new algorithm, a new scheduling policy, a new way to sort tasks or processors, a new placement heuristic, a new criterion of comparison for graphs etc can be made by adding a simple Java file to the project without further changes.

To introduce the current possibilities of the tool, we identified four main axis which will be explained in the following sections. A basic GUI has been developed to quickly test some features and understand the possibilities.

5.4 Test a Uni/Multiprocessor scheduling

The first option is to test a scheduling algorithm. This part corresponds to analytically test if a task set associated to a processor set will be schedulable or not (see Figure 5.1). Whatever the number of processors contained in the set, the tool should be able to act upon an algorithm and give a solution.

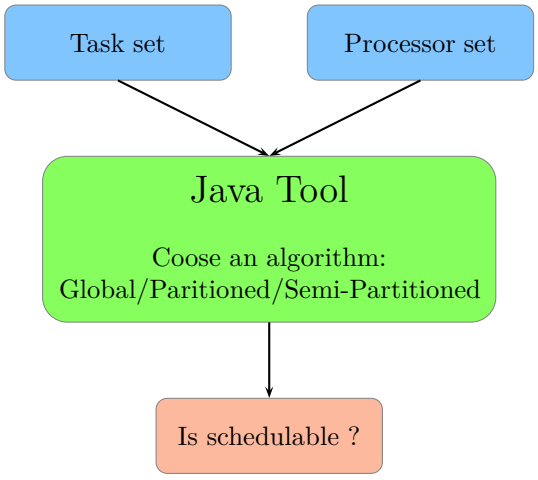


Figure 5.1 – Test a Scheduling

Based on the current state of the solutions, we have coded the three different approaches for multiprocessors:

- The G-Scheduling, which consists in scheduling tasks in a single queue and allow jobs to migrate between processors, requires a global schedulability test.

- With P-Scheduling, we need to find a placement heuristic to assign tasks to processors and then to use a uniprocessor feasibility/schedulability condition on each processor to decide on the schedulability of the task assigned to it. A sorting criterion for tasks and processors can be added to improve the performance of the approach. Notice that it corresponds exactly to our generalized P-Scheduling algorithm given in Section 3.2.

- The SP-Scheduling consists in partitioning the majority of the tasks, and allow a few to migrate between processors. In addition to P-Scheduling, this approach needs a uniprocessor feasibility/schedulability condition which takes into account the migrant tasks. In particular, we have coded our generic SP-Scheduling algorithm for **M**igration at **L**ocal **D**eadline (MLD) approaches defined in Section 3.3.

In order to obtain a modular and scalable tool, a scheduling algorithm has been split into several parts:

- A feasibility/schedulability test is an interface which has to answer if a task added to a given processor is schedulable.
- A placement heuristic that defines how the processors should be checked in order to assign tasks.
- A criterion for sorting tasks or processors that defines the order in which they must be addressed.

5.4.1 Placement Heuristics

Used for P-Scheduling and SP-Scheduling approaches, placement heuristic was defined as an abstract class. This abstract object needs one function: according to a feasibility/schedulability test, a processor and a task sets, it must return the processor able to schedule this task, if any.
Currently, the four placement heuristics given in Subsection 3.2.2.2.2 are coded: First-Fit, Next-Fit, Best-Fit and Worst-Fit.

Modularity A new placement heuristic can be added by deriving the abstract class. For information, the First-Fit heuristic is coded in about 10 lines.

5.4.2 Algorithm/Schedulability test

A scheduling algorithm is defined according to the multiprocessor approach used: G-Scheduling, P-Scheduling or SP-Scheduling. An abstract class defines the generic procedure for each approach:

- The G-Scheduling requires only a feasibility/schedulability test on all tasks and processors.
- The P-Scheduling sorts the tasks / processors based on criteria, then assigns them on processors according to the selected placement heuristic and to the uniprocessor feasibility/schedulability test defined in the algorithm.
- The SP-Scheduling offers several methods presented in the state-of-the-art, which includes different ways to determine when and how to split tasks between processors.

Modularity For example, about 10 lines in a Java file are sufficient to define the P-Scheduling algorithm which allows us to test any sort criterion of tasks and processors, any placement heuristic and which uses the uniprocessor feasibility/schedulability test for pre-emptive **E**arliest **D**eadline **F**irst (EDF) scheduler based on the computation of the *Load* function (See Subsection 2.4.3.1).

If we consider $\tau = \{\tau_1, \ldots, \tau_n\}$ a set of n sporadic sequential tasks, $\tau_i(C_i, T_i, D_i)$ the i^{st} task where C_i is its WCET, T_i is its minimum inter-arrival time and D_i is its relative deadline, here are some feasibility/schedulability tests currently available in the tool:

- $EDF{-}LL$ [LL73]: the total utilization of the set $U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1$.
- $EDF{-}BHR$ [BRH90]: $Load(\tau) \stackrel{\text{def}}{=} \sup_{t \geqslant 0} \frac{DBF(\tau,t)}{t} \leqslant 1$ with **D**emand **B**ound **F**unction (DBF) represents the upper bound of the work load generated by all tasks with activation times and absolute deadlines within the interval $[0; t]$. The tool implements some optimizations to accelerate the calculation of the *Load* function such as the computation of the C-Space using the simplex algorithm proposed by George and Hermant [GH09b] or the QPA algorithm of Zhang and Burns [ZB09].
- $DM{-}ABRTW$ [Aud+93]: **D**eadline **M**onotonic (DM) test based on the response time analysis: $\forall \tau_i \in \tau,\ r_i \leqslant D_i$, where r_i is τ_i's **W**orst **C**ase **R**esponse **T**ime (WCRT).
- $RM{-}LL$ [LL73]: **R**ate **M**onotonic (RM) test based on the total utilization of the set $U_\tau \leqslant n\left(\sqrt[n]{2} - 1\right)$.

Here are some G-Scheduling and SP-Scheduling algorithms currently available in the tool:

- RTA (G-Scheduling) proposed by Bertogna and Cirinei [BC07]. It is a global feasibility/schedulability test based on an iterative estimation of the WCRT of each task for Global EDF scheduler.
- $EDF{-}WM$ (SP-Scheduling) proposed by Kato, Yamasaki, and Ishikawa [KYI09]. It splits migrants tasks in subtasks and defines a window during which a subtask should be executed on a processor.
- $C{=}D$ (SP-Scheduling) proposed by Burns et al. [Bur+10]. It splits migrants tasks in two parts: one with a $C{=}D$ ($\tau_i^1(C, T_i, C)$), WCET equal to its deadline) and a second part with the remaining values ($\tau_i^2(C_i - C, T_i, D_i - C)$).

- *EDF−RRJM* (SP-Scheduling) proposed by George, Courbin, and Sorel [GCS11]. It uses **R**ound-**R**obin **J**ob **M**igration (RRJM) to split migrants tasks and reduces the number of migration by using job migrations at job boundaries. Notice that it is the algorithm proposed in Subsection 3.3.2.

5.5 View a scheduling

View Scheduling

The second option proposed is to allow the user to view the sequence of scheduling with respect to time. This part is performed by an abstract object *Scheduler* which will proceed according to the rules defined by the scheduling, check deadline misses and record the jobs scheduled. A GUI proposes a graphical representation of the scheduling (see Figure 5.2).

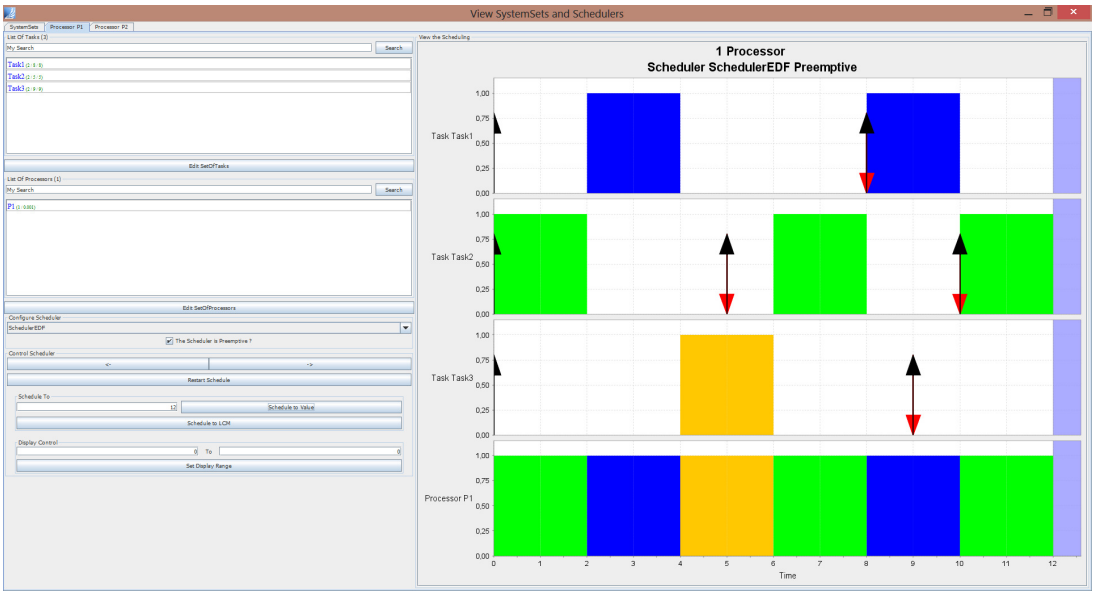


Figure 5.2 – GUI to display a scheduling

5.5.1 Available schedulers

Schedulers currently implemented are:

- PFair family (PF, PD^2) represents the global scheduling presented by Baruah, Gehrke, and Plaxton [BGP95],

- Arbitrary Priority Assignment chooses the active job with the highest predefined priority,

- **D**eadline **M**onotonic (DM) chooses the active job with the minimal relative deadline,
- **R**ate **M**onotonic (RM) chooses the active job with the minimal period,
- **E**arliest **D**eadline **F**irst (EDF) chooses the active job with the minimal absolute deadline,
- **L**east **L**axity **F**irst (LLF) chooses the active job with the minimal laxity.

Each of these scheduler can then be used as mono or multiprocessors schedulers: one Java object EDF can represent the uniprocessor EDF scheduler or the global EDF scheduler according to the number of processors available.

Modularity Add a new scheduling policy to the tool consists of adding an object that derives from the abstract class and only defines the function which chooses the job to be scheduled in the list of active jobs. The EDF scheduling, pre-emptive and non-pre-emptive, for uni and multiprocessor, is thus a Java file of about 10 lines.

5.6 Generate tasks and task sets

One of the challenges of a test tool for RT scheduling is to offer a method of generating sets of tasks which give representative and reusable results for the most honest and consistent possible comparison.

We based our methods of generation of tasks and sets according to the work of Baker [Bak06] and the $UUnifast$ algorithm proposed by Bini and Buttazzo [BB04]. With a modular and abstract code, it is possible to use various methods of generation and various parameters such as type of task deadline or a specific probability distribution for the utilization of tasks. Sets are saved in an XML file to be loaded for others options of the tool.

5.6.1 Generating a Task

Here we present the procedure derived from the work of Baker [Bak06]. To generate a task, several parameters are needed:

- The type of deadline, **I**mplicit **Deadline** (I-Deadline) (the deadline of each task equal its period), **C**onstrained **Deadline** (C-Deadline) (the deadline of each task is less than or equal to its period) or **A**rbitraty **Deadline** (A-Deadline) (the deadline of each task can be lower, equal or greater than its period),

- The probability distribution of the utilization of each task (such as uniform within the interval $[0; 1]$ or exponential of mean 0.5),
- The interval used to generate the values of periods and deadlines.

The generation procedure is as follows:

1. The period is generated following a uniform distribution in the defined interval,
2. The utilization of the task is generated according to the distribution selected,
3. The value of WCET is calculated based on the period and utilization of the task,
4. The value of the deadline is set to the period (I-Deadline), uniformly selected between the WCET and period (C-Deadline) or uniformly selected between the WCET and the maximum value of the defined interval (A-Deadline).

5.6.2 Generating Sets Of Tasks

To generate sets of tasks, several functions are available but the main procedure is also extracted from the work of Baker [Bak06].

The following procedure needs a task generator (see Subsection 5.6.1), a minimum number of tasks, a maximum utilization of task set and a number of sets to produce:

1. The minimum number of tasks is created based on the task generator; utilization of the set must not exceed the maximum utilization defined. This is the first task set.
2. A new task is generated according to the same task generator. If it can be added to the previous set without exceeding the maximum defined utilization, it is added to create a new set. If not, return to the previous step.

These steps are repeated until the number of sets expected is reached.

5.7 Edit/Run an evaluation

This option uses all the previous options defined in Sections 5.4, 5.5 and 5.6 to automate the generation of results in order to compare various algorithms (see Figure 5.3).

It can save results for reuse and share them and extract values for graphs.

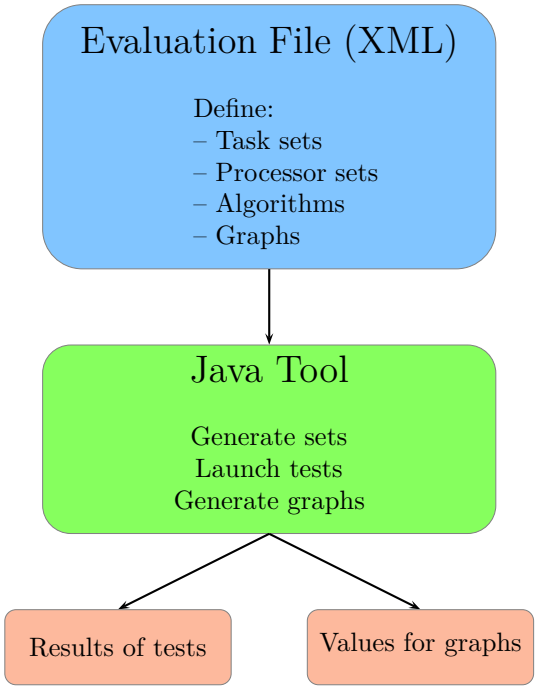


Figure 5.3 – Edit/Run an Evaluation

The definition of an evaluation is done in an XML file containing:

1. A list of types of task sets. These task sets can be defined by generation parameters according to Subsection 5.6.2 (see Subsection 5.7.1),

2. An equivalent list for processor sets (see Subsection 5.7.1),

3. A list of algorithms. For each one, we can define some settings: placement heuristics, criteria for sorting tasks, type of task sets (previously defined in the XML at point 1) and the processor sets to be considered (previously defined in the XML at point 2) (see Subsection 5.7.2),

4. A list of graphs to be produced according to the results (see Subsection 5.7.3).

5.7.1 Defining the sets

You could choose to use pre-existing sets of tasks or define generation parameters (see Section 5.6) and let the generator create the sets.

```
<EvaluationSetOfTasks>
  <Name value="Deadline_IMPLICIT___Distrib_UNIFORM" />
  <NbSetOfTasks value="10000" type="Integer" />
  <AutoPath value="true" type="Boolean" />
  <Path value="./SetOfTasks/" />
  <FileName value="setOfTasks.xml" />
  <CriterionDeadline value="IMPLICIT" type="CriterionDeadline" />
  <CriterionDistribution value="UNIFORM" type="CriterionDistribution" />
  <GeneratorSetOfTasks
      parameterClassName="TaskModel.SetOfTasksGenerator.GeneratorSetOfTasksETFA">
    <Name value="mySetOfTasksGenerator" />
    <NbMinTask value="5" type="Integer" />
    <MinUtilization value="2" type="BigDecimal" />
    <MaxUtilization value="4" type="BigDecimal" />
    <GeneratorTask parameterClassName="TaskModel.SetOfTasksGenerator.GeneratorTaskETFA">
      <Name value="myTaskGenerator" />
      <CriterionUtilizationDistribution value="UNIFORM" type="CriterionDistribution" />
      <CriterionDeadline value="IMPLICIT" type="CriterionDeadline" />
      <PeriodDeadlineMin value="1" type="BigDecimal" />
      <PeriodDeadlineMax value="100" type="BigDecimal" />
      <Precision value="15" type="Integer" />
      <TaskActivationName value="TaskActivationPeriodic" />
      <MaxU value="1" type="BigDecimal" />
    </GeneratorTask>
  </GeneratorSetOfTasks>
</EvaluationSetOfTasks>
```

Figure 5.4 – Example to define a type of task sets in the XML Evaluation file

Figure 5.4 defines that in the folder "./SetOfTasks/", a file "setOfTasks.xml" will be placed in a sub-folder "./SetOfTasks/IMPLICIT_UNIFORM/" auto generated and will contain 10000 sets of tasks with a total utilization between 2 and 4, a minimum of 5 tasks for each set and each task will be generated with an "IMPLICIT" deadline (I-Deadline) and an "UNIFORM" distribution of utilization within the interval $[0; 1]$.

```
<EvaluationSetOfProcessors name="4_Processors_HOMOGENEOUS" autoPath="false"
    path=".\SetOfProcessors" fileName="setOfProcessors4.xml" nbProcessors="4"
    type="HOMOGENEOUS" />
```

Figure 5.5 – Example to define a type of processor set in the XML Evaluation file

Figure 5.5 defines that in a folder "./SetOfProcessors/", a file "setOfProcessors4.xml" contains the definition of a processor set with 4 homogeneous processors.

5.7.2 Defining the scheduling algorithms

Then, you define algorithms to be tested. For each, indicate the name of the scheduling algorithm (corresponding to its class name), a file path defining the location where results will be stored and parameters such as the placement heuristics to consider, task and processor sets to test and criteria for sorting tasks and processors.

```
<EvaluationAlgorithm name="4_EDFPartitioned" algoName="EDF_Load_P" path=".\Results"
      fileName="results.xml" >
  <Heuristic>FIRST_FIT</Heuristic>
  <Heuristic>WORST_FIT</Heuristic>
  <CriterionSortSetOfProcessors>PROCESSOR_NONE_ORDER</CriterionSortSetOfProcessors>
  <CriterionSortSetOfTasks>TASK_DENSITY_DECREASING_ORDER</CriterionSortSetOfTasks>
  <EvaluationSetOfTasks>Deadline_IMPLICIT___Distrib_UNIFORM</EvaluationSetOfTasks>
  <EvaluationSetOfProcessors>4_Processors_HOMOGENEOUS</EvaluationSetOfProcessors>
</EvaluationAlgorithm>
```

Figure 5.6 – Example to define an algorithm in the XML Evaluation file

Figure 5.6 defines that the algorithm "EDF_Load_P" (which correspond to the P-Scheduling algorithm based on the schedulability test using the computation of the *Load* to EDF pre-emptive scheduler) will be tested on the previously defined task set "Deadline_IMPLICIT___Distrib_UNIFORM", without sorting processors and sorting tasks according to decreasing density. Placement heuristics "FIRST_FIT" and "WORST_FIT" will be tested following all possible combinations between all previous parameters.

The results will be stored automatically in files named "results.xml", in separate sub-folders for each parameter in the main folder "./Results/".

5.7.3 Defining a graph result

Finally, parameters for graphs can be defined. X-axis and Y-axis have to be selected according to a class name. For example, "GetUtilizationValue" returns the utilization of the task set, "GetSuccessValue" retrieves in the result files if the set has been successfully scheduled by the algorithm.

Modularity A new class placed in the correct package will automatically add a new possible value for axis in graphs.

By defining a curve name, it indicates what each curve must represent. For example, "GetAlgorithmCurveName" will generate a curve for each algorithm, while "GetHeuristicCurveName" will generate a curve for each placement heuristic found in the result files.

Modularity To add a new type of curve, just add a Java file with a class derived from the abstract object "GetCurveName".

It is also possible to filter the results in order to focus only on some of the data. For example, the graph can concentrate on a particular type of deadline or on results for a 4-processor platform. It can consider only some algorithms, some heuristics or only sets of tasks in a particular range of utilization.

Modularity Each of these parameters corresponds to "filter", it is possible to add a new filter to the tool by filing a Java file derived from the abstract class in the correct package.

```
<Graphs path="./Graphs/">
  <Graph name="MyGraph " Scale="1">
    <GetValueX name="GetUtilizationValue" />
    <GetValueY name="GetSuccessValue" />
    <GetCurveName name="GetAlgorithmCurveName" />
    <Deadlines>
      <Deadline>IMPLICIT</Deadline>
      <Deadline>CONSTRAINED</Deadline>
    </Deadlines>
    <Distributions>
      <Distribution>UNIFORM</Distribution>
    </Distributions>
    <NumbersOfProcessors>
      <NumberOfProcessors>4</NumberOfProcessors>
    </NumbersOfProcessors>
    <Filters>
      <Filter name="StatisticsHeuristicFilter">
        <ToKeep>FIRST_FIT</ToKeep>
      </Filter>
      <Filter name="StatisticsAlgorithmFilter">
        <ToKeep>EDF_Load_P</ToKeep>
        <ToKeep>DM_RT_P</ToKeep>
      </Filter>
      <Filter name="StatisticsUtilizationRangeFilter">
        <ToKeep>2</ToKeep>
        <ToKeep>4</ToKeep>
      </Filter>
    </Filters>
  </Graph>
</Graphs>
```

Figure 5.7 – Example to define a graph in the XML Evaluation file

Figure 5.7 creates a text file "MyGraph.txt" in the folder "./Graphs/" containing data which describe a graph with a X-axis representing the utilization of sets of tasks, Y-axis the success ratio. Each curve will be a different algorithm. We will focus on sets of tasks with "IMPLICIT" or "CONSTRAINED" deadlines, with a utilization generated with a "UNIFORM" distribution of probability. Only 4-processor platform will be checked and results from the "FIRST_FIT" heuristic. Both algorithms "EDF_Load_P" and "DM_RT_P" (P-Scheduling algorithm

based on the schedulability **N**ecessary and **S**ufficient **Test** (NS-Test) on response time for a DM pre-emptive scheduler) will be taken into account. Finally, we are interesting only in sets of tasks with utilization in the range $[2; 4]$.

The graph produced with the example given in this chapter is shown in Figure 5.7. This figure is created using Gnuplot (`http://www.gnuplot.info/`) to interpret "MyGraph.txt".

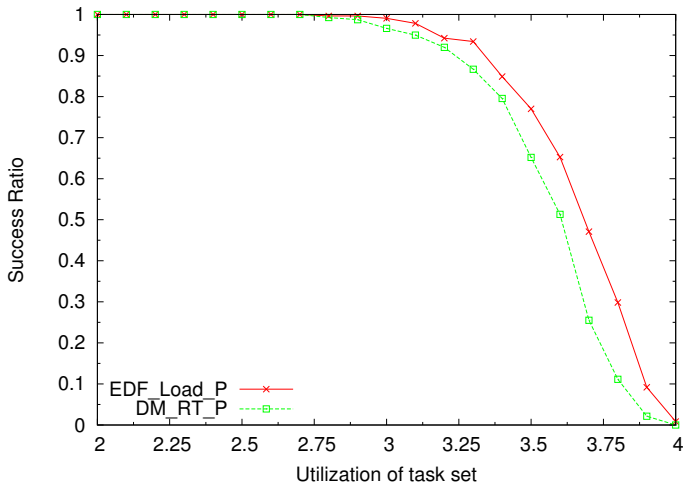


Figure 5.8 – Example of graph produced according to the example

5.7.4 Generating the evaluations

The evaluation file allows us to automate the whole procedure: the generation of sets of tasks and the generation of graphs. Filters allows us to reuse some of the results and thus to resume the evaluations conducted previously.

However, this process can be time-consuming. Since the tool can also be used with a command-line, it allows us to run the computation, stop them at a predefined times and resume them later. It can then be used to spread the workload over multiple computers: the tool will generate a list of parameters corresponding to an XML Evaluation file; each parameter can be run on different computers and then assembled without recoveries problems.

5.8 Summary

Research in **R**eal-**T**ime (RT) scheduling has produced a large number of algorithms with their associated feasibility/schedulability conditions to respond to the increasing complexity of multiprocessors architectures. However, it is difficult to find tools able to evaluate and compare these algorithms based on simulations or on analytical tests. Our tool named **F**ramework f**O**r **R**eal-**T**ime **A**nalysis

and Simulation (FORTAS) offers to facilitate the comparison between different algorithms for uniprocessor and multiprocessors RT scheduling. Developed in Java with a programming paradigm oriented to modules and abstraction, it gives the user the opportunity to develop their own extensions. Moreover, it proposes to automate the process of comparing different algorithms: generation of task sets, computation of results for each algorithm and generation of graphs for comparison.

To sum up, FORTAS allows the user to test if a task set is schedulable on a processor set according to a specific algorithm. You may view the sequence of scheduling in time to check that no deadline is missed. A procedure is also proposed to generate sets of tasks according to various parameters. Furthermore, the tool offers to automate the creation of evaluations of algorithms from beginning to end: generation of sets to test, computation of the results for all algorithms desired with a distribution of work on different computers and finally creation of graphs associated. We give an overview of currently available functionalities in Tables 5.1-5.4. All these options can be improved by the user by defining itself new parameters, new algorithms, new axes for graphs etc. This is facilitated by a programming paradigm oriented to modules and abstract classes.

Notice that FORTAS has already been used effectively for various published papers ([Lup+10; DGC10; GC11; GCS11]).

Test a Uni/Multiprocessor scheduling		
P-Scheduling		
Placement heuristics	**Schedulability/Feasibility tests**	**Sorting criteria**
First-Fit	EDF-LL [LL73]	Increasing/Decreasing order of *relative deadline*
Best-Fit	EDF-BHR [BRH90]	Increasing/Decreasing order of *period*
Worst-Fit	EDF-BF [BF06]	Increasing/Decreasing order of *density*
Next-Fit	DM-ABRTW [Aud+93]	Increasing/Decreasing order of *utilization*
	RM-LL [DG00]	
	RM-BBB [BBB03]	
	RM-LMM [LMM98]	
SP-Scheduling		
Placement heuristics	**Split techniques**	**Sorting criteria**
First-Fit	C=D [Bur+10]	Increasing/Decreasing order of *relative deadline*
Best-Fit	EDF-WM [KYI09]	Increasing/Decreasing order of *period*
Worst-Fit	EDF-MLD-Dfair-Cfair [GCS11]	Increasing/Decreasing order of *density*
Next-Fit	EDF-MLD-Dmin-Cexact [GCS11]	Increasing/Decreasing order of *utilization*
	EDF-RRJM [GCS11]	
G-Scheduling		
Global technique and Schedulability/Feasibility tests		
EDF-Load [BB09]		
EDF-RTA [BC07]		
U-EDF [Nel+11; Nel+12]		
RUN [Reg+11]		

Table 5.1 – Available functionalities for the "test" part of FORTAS

View a scheduling
Schedulers, pre-emptive or non-pre-emptive
Deadline **M**onotonic (DM)
Rate **M**onotonic (RM)
Arbitrary Priority Assignement
Earliest **D**eadline **F**irst (EDF)
Least **L**axity **F**irst (LLF)
PFair family (PF, PD^2) [BGP95]
U-EDF [Nel+11; Nel+12]
RUN [Reg+11]

Table 5.2 – Available functionalities for the "view" part of FORTAS

Generate tasks and task sets			
Generation techniques	**Types of task deadline**	**Probability distribution of utilization**	**Options**
UUnifast [BB04] Baker [Bak06]	I-Deadline C-Deadline A-Deadline	UNIFORM BIMODAL EXPONENTIAL	Number of task sets Minimum number of tasks Limit task set utilization Limit lcm of task periods

Table 5.3 – Available functionalities for the "generate" part of FORTAS

Edit/Run an evaluation	
Parameters for the evaluation	
Define or generate the sets (See Table 5.3) Define the scheduling algorithms (See Table 5.1) Define the graphs result parameters	
Graph result parameters	
Options for axis values	**Comparison criterion (curve type)**
Number of task set scheduled (Success Ratio) Number of task per task set Density or utilization of task set Number of processors used Average remaining density or utilization on processors	Scheduling algorithm Criterion for sorting tasks Placement heuristic

Table 5.4 – Available functionalities for the "evaluation" part of FORTAS

Part IV

Conclusion and perspectives

Chapter 6

Conclusion

Une des maximes favorites de mon père était la distinction entre les deux sortes de vérités, des vérités profondes reconnues par le fait que l'inverse est également une vérité profonde, contrairement aux banalités où les contraires sont clairement absurdes.

One of the favorite maxims of my father was the distinction between the two sorts of truths, profound truths recognized by the fact that the opposite is also a profound truth, in contrast to trivialities where opposites are obviously absurd.

Hans Henrik Bohr [Roz67]

Contents

In this thesis, we have addressed the problem of hard **R**eal-**T**ime (RT) scheduling upon identical multiprocessor platforms. A RT system is a system having time constraints (or timeliness constraints) such that the correctness of these systems depends on the correctness of results it provides, but also on the time instant the results are available. Thus, the problem of scheduling tasks on a hard RT system consist in finding a way to choose, at each time instant, which tasks should be executed on the processors so that each task succeeds to complete its work before its deadline. In the multiprocessor case, we are not only concerned by the respect of all deadlines but we also aim to efficiently use all the processors. Is the number of processors enough? Is there a method to better utilize these processors? A lot of research exists in the literature of the state-of-the-art to propose solutions to this problem.

6.1 Scheduling Sequential Task (S-Task)

First, we have studied **S**equential **Tasks** (S-Tasks) scheduling problem. We have investigated two of the mains approaches: **P**artitioned **Scheduling** (P-Scheduling) approach and **S**emi-**P**artitioned **Scheduling** (SP-Scheduling) approach.

6.1.1 P-Scheduling approach

For the P-Scheduling approach, we have studied different partitioning algorithms proposed in the literature of the state-of-the-art in order to elaborate a generic partitioning algorithm (Algorithm 3 on page 72). Especially, we have investigated four main placement heuristics (First-Fit, Best-Fit, Next-Fit and Worst-Fit), eight criteria for sorting tasks and seven schedulability tests for **E**arliest **D**eadline **F**irst (EDF), **D**eadline **M**onotonic (DM) and **R**ate **M**onotonic (RM) schedulers. It is equivalent to 224 potential P-Scheduling algorithms. Then, we have analysed each of the parameters of this algorithm to extract the best choices according to various objectives.

Our simulations allowed us to confirm a common assumption: the heuristics which have the best results in terms of success ratio are Best-Fit and First-Fit. Likewise, the sorting task criterion which maximizes the success ratio is Decreasing Density, similar to Decreasing Utilization. Moreover, this result has been recently confirmed through a *speedup factor* analysis by Baruah [Bar13] for EDF scheduler and **I**mplicit **Deadlines** (I-Deadlines) tasks.

Finally, we have put ourselves in a practical case where we had to choose the parameters of the algorithm according to the constraints of our problem. We have identified three main practical cases and we have summed up our results for each case in Table 3.2 on page 69.

6.1.2 SP-Scheduling approach

Afterwards, we have studied the SP-Scheduling approach for which we have proposed a solution for each of the two sub-categories: with **Rest**ricted **Migrations** (Rest-Migrations) where migrations are only allowed between two successive activations of the task (in other words, between two jobs of the task, thus only *task migration* is allowed), and with **UnRest**ricted **Migrations** (UnRest-Migrations) where migrations are not restricted to job boundaries (*job migration* is allowed). For the Rest-Migration case we have provided **R**ound-**R**obin **J**ob **M**igration (RRJM), a new job placement heuristic, and an associated schedulability **N**ecessary and **S**ufficient **Test** (NS-Test) for EDF scheduler. RRJM consists in assigning the jobs of a task to a set of processors and define a recurrent pattern of successive migrations using a Round-Robin pattern of migration. For the UnRest-Migration case we have studied the **M**igration at **L**ocal **D**eadline (MLD)

approach which consists in using local deadlines to specify migration points. We have provided a generic SP-Scheduling algorithm for MLD approaches and an associated schedulability NS-Test for EDF scheduler. We have used an evaluation to compare the performances of our Rest-Migration approach compared to the UnRest-Migration with MLD approach. In particular, we have observed that the approach with UnRest-Migration gives the best results in terms of number of task sets successfully scheduled. However, we have noticed a limit on the ability of this approach to split tasks between many processors: if the execution time of the task is too small compared to the time granularity of processor execution, it will be impossible to split the execution time. Thus, the Rest-Migration approach is still interesting, especially as its implementation seems to be easier to achieve on real systems.

6.2 Scheduling Parallel Task (P-Task)

Regarding **P**arallel **Tasks** (P-Tasks) scheduling problem, we have proposed the **M**ulti-**P**hase **M**ulti-**T**hread (MPMT) task model which is a new model for Multi-Thread tasks to facilitate scheduling and analysis. We have also provided schedulability NS-Tests and a method for transcribing Fork-Join tasks to our new task model. An exact computation of the **W**orst **C**ase **R**esponse **T**ime (WCRT) of a periodic MPMT task has been given as well as a WCRT bound for the sporadic case. Finally, we have proposed an evaluation to compare Gang and Multi-Thread approaches in order to analyse their advantages and disadvantages. In particular, even if we have showed that both approaches may be incomparable (there are task sets which are schedulable using Gang approach and not by using Multi-Thread approach, and conversely.), the Multi-Thread model allows us to schedule a larger number of task sets and it reduces the WCRT of tasks. Thus, if the tasks do not require too much communication between concurrent threads, it seems interesting to model them with a Multi-Thread approach.

6.3 Our tool: FORTAS

Finally, we have developed the framework called **F**ramework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation (FORTAS) to facilitate evaluations and tests of multi-processor scheduling algorithms. Its particularity is to provide a programming library to accelerate the development and testing of RT scheduling algorithms. It is developed with a modular approach to facilitate the addition of new schedulers, P-Scheduling algorithms, **G**lobal **Scheduling** (G-Scheduling) or SP-Scheduling algorithms, schedulability tests, etc. This framework will be proposed as an open source library for the research community.

6.4 Perspectives

A lot of interesting questions and improvements are opened up for further researches. Here we draw up a non-exhaustive list:

- For the scheduling of S-Tasks:
  - In the P-Scheduling approach, we focused on simulations to evaluate the parameters of our generic algorithm. Following the work of Baruah [Bar13], it may be interesting to confirm the other results of evaluation by theoretical analysis.
  - We think that other SP-Scheduling algorithms should be further investigated by define a more precise taxonomy of different algorithms to facilitate their study and comparison.
  - We conjectured that SP-Scheduling approaches with Rest-Migration would be easier to implement than approaches with UnRest-Migration. It would be interesting to check this proposal by implementing various SP-Scheduling approaches on actual RT systems.
- For the scheduling of P-Tasks:
  - During the comparison of scheduling Gang tasks versus Multi-Thread tasks, we have constrained our tasks to have only one phase since Gang schedulers consider that the execution requirement of processes corresponds to a "$C_i \times V_i$" rectangle. Further research could be conducted to assess how evolves the comparison according to the complexity introduced by our MPMT task model.
  - Our MPMT task model allows us to define different number of threads for each phase of a task. In our study, we considered that this number was previously given during the task definition. Following our work with Bado et al. [Bad+12], it would be interesting to explore different way to compute this value in order to maximize the success ratio and the total utilization of the platform.
  - Our MPMT task model should be studied more deeply and possibly extended to handle different cases or find its limits. A comparison with others P-Task models could be an interesting research direction. The representation using precedence constraints as presented in several publications and by Nelissen [Nel13] seems to be a important research direction.
- Considering the development of FORTAS, with Frederic Fauberteau recently arrived in our research group, we aim to improve this framework and work with groups from other laboratories in order to combine the expertise and benefits of tools that each one has created.

Concerning more personal perspectives, we plan to expand the theories and practices of research developed during this thesis to other application areas:

- We want to continue the collaboration initiated with Vincent Sciandra [SCG12] on the application of RT scheduling theory to public transport systems and especially the **E**uropean **B**us **S**ystem of the **F**uture (EBSF) European project. The approach using a representation of the constraints with mixed criticality tasks seems promising.
- The fruitful discussions with Clément Duhart and Rafik Zitouni (colleagues and PhD students) seem promising to apply the RT scheduling theories to problems encountered in the field of sensor networks and especially for the **E**nvironment **M**onitoring and **M**anagement **A**gents (EMMA) project that aims to improve energy management at home. The thoughts that we have conducted on how to schedule home appliances in order to reduce overall electricity consumption seems promising, especially for comparison with the approach proposed by EMMA which is to decentralize all scheduling choices.
- Finally, very interested for years by parallel programming, we want to consolidate our knowledge in this field to better integrate its specificities in our research on RT scheduling.

List of symbols

$\mathbb{Z}$	Integers numbers: $\ldots, -2, -1, 0, 1, 2, \ldots$
$\mathbb{N}$	Natural numbers: $0, 1, 2, \ldots$
$\mathbb{R}$	Real numbers
$\lvert x \rvert$	Absolute value of x
$[x; y]$	Interval of real values: $\{a \in \mathbb{R} \lvert x \leqslant a \leqslant y\}$
$[x; y)$	Half-open interval of real values: $\{a \in \mathbb{R} \lvert x \leqslant a < y\}$
$[\![x; y]\!]$	Interval of integers values: $\{a \in \mathbb{Z} \lvert x \leqslant a \leqslant y\}$
$[\![x; y)\!]$	Half-open interval of integers values: $\{a \in \mathbb{Z} \lvert x \leqslant a < y\}$
$\lceil x \rceil$	Ceil of x
$\lfloor x \rfloor$	Floor of x
max	Maximum
min	Minimum
$[\![A]\!]_B$	A has lower bound B such that $[\![A]\!]_B = \max(A, B)$
$[\![A]\!]^C$	A has upper bound C such that $[\![A]\!]^C = \min(A, C)$
$[\![A]\!]_B^C$	$[\![A]\!]_B^C = [\![[\![A]\!]_B]\!]^C$
mod	Modulo
lcm	Least common multiple
π	A processor set
π_k	A processor
τ	A task set
τ^i	A task set
τ_i	A task
τ^{π_k}	A task set associated to processor π_k
$\tau_i^{\pi_k}$	A task associated to processor π_k
$\tau_i \succ \tau_j$	τ_i has a higher priority than τ_j
$\tau_i \prec \tau_j$	τ_i has a lower priority than τ_j
$\tau^{hp(\tau,\tau_i)}$	The task set composed of the tasks in τ which have a priority higher than τ_i. $\tau_j \in \tau^{hp(\tau,\tau_i)}$ if $\tau_j \in \tau$ and $\tau_j \succ \tau_i$.
$\tau^{lp(\tau,\tau_i)}$	The task set composed of the tasks in τ which have a priority lower than τ_i. $\tau_j \in \tau^{lp(\tau,\tau_i)}$ if $\tau_j \in \tau$ and $\tau_j \prec \tau_i$.
n	The number of tasks
m	The number of processors
P	The least common multiple of all task period, $P \stackrel{\text{def}}{=} \text{lcm}\{T_1, \ldots, T_n\}$

Glossaries

Acronyms

RT

***R**eal-**T**ime*. xiii–xv, xvii–xix, 3–6, 8, 10, 15, 19–21, 35, 38, 41, 44, 45, 50, 94, 95, 104, 121, 132, 136, 137, 142, 148, 149, 155, 157–159

A-Deadline

A**rbitraty **Deadline. 11, 20, 21, 25, 31, 32, 58, 82, 142, 143, 151,
— Glossary: A-Deadline

C-Deadline

C**onstrained **Deadline. 10, 20, 24, 27, 38, 41, 58–60, 63, 64, 69, 86, 87, 96, 99, 108, 109, 111–114, 142, 143, 151,
— Glossary: C-Deadline

CI

***C**arry **I**n*. 115, 117–121, 168, 169,
— Glossary: CI

DBF

***D**emand **B**ound **F**unction*. 15, 25–29, 31–35, 37, 39, 41, 58, 140, 168,
— Glossary: DBF

DJP

***D**ynamic **J**ob **P**riority*. 21

DM

***D**eadline **M**onotonic*. xiv, xviii, 20, 24, 57, 58, 100, 106, 123–130, 140, 142, 148, 150, 156

DPP

***D**ynamic **P**rocess **P**riority*. 106, 107

DThP

***D**ynamic **Th**read **P**riority*. 106, 107

DTP

***D**ynamic **T**ask **P**riority*. 19, 21, 24, 43, 79

EBSF

***E**uropean **B**us **S**ystem of the **F**uture*. 159, 168,
— Glossary: EBSF

EDF

***E**arliest **D**eadline **F**irst*. xiv, xviii, xxxiii, 21, 23–25, 31–33, 37–40, 43–45, 51, 57, 59, 61–64, 69, 72–74, 76, 79, 82, 83, 85, 87, 88, 91, 98, 106, 140, 142, 146, 150, 156, 157

EMMA

***E**nvironment **M**onitoring and **M**anagement **A**gents*. 159

FJP

***F**ixed **J**ob **P**riority*. 21

FORTAS

***F**ramework f**O**r **R**eal-**T**ime **A**nalysis and **S**imulation*. xv, xix, 6, 137, 148, 149, 157, 158

FPP

***F**ixed **P**rocess **P**riority*. 100, 106, 107

FSP

***F**ixed **S**ub-program **P**riority*. 103–113, 123, 126, 132

FThP

***F**ixed **Th**read **P**riority*. 106, 107

FTP

***F**ixed **T**ask **P**riority*. 19, 20, 23, 42, 43, 45, 59, 61–64, 69, 79, 82, 103–113, 123, 126, 132

G-Scheduling

G**lobal **Scheduling. xiii, xviii, 35, 38, 40, 44, 70, 136, 138–140, 150, 157

GUI

***G**raphical **U**ser **I**nterface*. 136, 137, 141

I-Deadline

I**mplicit **Deadline. 10, 20, 23, 24, 28, 38–40, 43, 51, 54, 57, 58, 60, 63, 64, 69, 86, 87, 142, 143, 145, 151, 156,
— Glossary: I-Deadline

IM

***I**ndex **M**onotonic*. 123–130

LLF

***L**east **L**axity **F**irst*. 21, 106, 142, 150

LSF

***L**ongest **S**ub-program **F**irst*. 103, 106, 107, 110, 113

MLD

***M**igration at **L**ocal **D**eadline*. 76, 78, 79, 82–84, 89, 91, 138, 156, 157, 168, —
Glossary: MLD

MPI

***M**essage **P**assing **I**nterface*. 5

MPMT

***M**ulti-**P**hase **M**ulti-**T**hread*. xiv, xv, xix, 94, 98, 100–103, 105, 108, 109, 111–114, 121, 126, 132, 157, 158, 169, —
Glossary: MPMT

N-Test

N**ecessary **Test. 22–24, —
Glossary: N-Test

NC

***N**on **C**arry-in*. 115, 117–119, 121, 169, —
Glossary: NC

NS-Test

N**ecessary and **S**ufficient **Test. 23–25, 45, 57–59, 62, 63, 72–76, 78, 79, 82, 84, 85, 94, 108, 111–113, 148, 156, 157, —
Glossary: NS-Test

OPA

***O**ptimal **P**riority **A**ssignment*. 20

OpenMP

***O**pen **M**ulti-**P**rocessing*. 5, 16, 17

P-Scheduling

P**artitioned **Scheduling. xiii, xiv, xvii, xviii, 6, 35–38, 40–42, 44, 45, 50–53, 57, 59, 60, 64, 67, 68, 70–72, 74, 75, 78, 79, 82, 84, 85, 87–91, 136, 138–140, 146, 147, 150, 156–158

P-Task

P**arallel **Task. xiii, xiv, xvii–xix, 5, 6, 9, 15, 35, 45, 94, 157, 158, 168, —
Glossary: P-Task

Pthread

P**OSIX **thread. 16, 17

RBF

***R**equest **B**ound **F**unction*. 15, 24, 169, —
Glossary: RBF

Rest-Migration

Rest**ricted **Migration. xiv, xviii, xix, 6, 41, 71, 73, 74, 83, 84, 86, 89, 91, 156–158, 169, —
Glossary: Rest-Migration

RM

***R**ate **M**onotonic*. xiv, xviii, 20, 23, 24, 57, 103, 106, 107, 113, 140, 142, 150, 156

RMA

***R**ate **M**onotonic **A**nalysis*. 136

RRJM

***R**ound-**R**obin **J**ob **M**igration*. 73, 74, 91, 141, 156, —
Glossary: RRJM

RTSJ

***R**eal-**T**ime **S**pecification for **J**ava*. 44

S-Task

S**equential **Task. xiii, xiv, xvii, xviii, 5, 6, 9, 14, 15, 20, 21, 35, 37, 39, 41, 45, 50, 156, 158, —
Glossary: S-Task

S-Test

***S**ufficient **Test**.* 23, 24, 37, 39, 45, 57, 58, 64,
— Glossary: S-Test

SP-Scheduling

***S**emi-**P**artitioned **Scheduling**.* xiv, xviii, xxxi, 6, 35, 40, 42, 44, 45, 50, 70–72, 74, 75, 78, 79, 82, 83, 87–91, 136, 138–141, 150, 156–158, 168–170

TBB

***T**hreading **B**uilding **B**locks.* 5

UML

***U**nified **M**odeling **L**anguage.* 94

UnRest-Migration

***UnRest**ricted **Migration**.* xiv, xviii, 6, 41, 44, 71, 72, 76, 78, 83, 84, 86, 89, 91, 156–158, 168, 170,
— Glossary: UnRest-Migration

WCET

***W**orst **C**ase **E**xecution **T**ime.* 13, 14, 17, 18, 26–31, 41–44, 50, 74, 76, 79–82, 84, 91, 96, 99, 101–103, 108, 118, 120, 137, 140, 143, 170,
— Glossary: WCET

WCRT

***W**orst **C**ase **R**esponse **T**ime.* xv, xix, 15, 24, 39, 43, 58, 77, 78, 94, 103–105, 113–115, 118, 121, 126, 127, 130, 131, 140, 157, 170,
— Glossary: WCRT

Glossary

A-Deadline

A task is said to have A-Deadline when there is no link between its deadline and its period, so $D_i \leqslant T_i$ or $D_i \geqslant T_i$. 163

C-Deadline

A task is said to have C-Deadline when its deadline is lower or equal to its period, so $D_i \leqslant T_i$. 163

CI

A **C**arry **I**n (CI) task refers to a task with one job with arrival instant earlier than the interval $[a; b]$ and deadline in the interval $[a; b]$. See Figure 4.7 on page 116. 163

DBF

The **D**emand **B**ound **F**unction (DBF) represents the upper bound of the work load generated by all tasks with activation instants and absolute deadlines within the interval $[0; t]$. See example on page 15. 163

EBSF

EBSF is an initiative of the European Commission under the Seventh Framework Programme for Research and Technological Development. Starting in September 2008; EBSF is a four-year project with an overall budget of 26 million Euros (16 millions cofunded) and is coordinated by UITP, the International Association of Public Transport. See `http://www.ebsf.eu/`. 164

I-Deadline

A task is said to have I-Deadline when its deadline is equal to its period, so $D_i = T_i$. 164

MLD

In the SP-Scheduling approach with UnRest-Migration, MLD refers to the solution of using local deadlines to specify migration points. See Definition 3.3 on page 76. 165

MPMT

P-Task model given by Definition 4.8 on page 98. 165

N-Test

A test is said to be necessary if a negative result allows us to reject the proposition but a positive result does not allow us to accept the proposition. See Definition 2.12 and example on page 22. 165

NC

A **N**on **C**arry-in (NC) task is the opposite of a CI task. It refers to a task with one job with arrival instant and deadline in the interval $[a; b]$. See Figure 4.6 on page 115. 165

NS-Test

A test is said to be necessary and sufficient if a positive result allows us to accept the proposition and a negative result allows us to reject the proposition. See Definition 2.14 and example on page 23. 165

P-Task

Task model presented in Definition 2.6 (See page 16) for the Gang model, Definition 2.7 (See page 18) for the Fork-Join model and Definition 4.8 (See page 98) for the MPMT model. 166

RBF

The **R**equest **B**ound **F**unction (RBF) represents the upper bound of the work load generated by all tasks with activation instants included within the interval $[0; t)$. See example on page 15. 166

Rest-Migration

In the SP-Scheduling approach, Rest-Migration refers to the case where migration is allowed, but only at job boundaries. A job is executed on one processor but successive jobs of a task can be executed on different processors. See Figure 3.13.1 on page 71. 166

RRJM

Job placement heuristic used for the SP-Scheduling approach with Rest-Migration. See Definition 3.1 on page 73. 166

S-Task

Task model presented in Definition 2.4 for the periodic case and Definition 2.5 for the sporadic case. (See page 13). 166

S-Test

A test is said to be sufficient if a positive result allows us to accept the proposition but a negative result does not allow us to reject the proposition. See Definition 2.13 and example on page 23. 167

UnRest-Migration

In the SP-Scheduling approach, UnRest-Migration refers to the case where migration is allowed, and a job can be portioned between multiple processors. A job can start its execution on one processor and complete on an other processor. See Figure 3.13.2 on page 71. 167

WCET

The **W**orst **C**ase **E**xecution **T**ime (WCET) of a task is the maximum execution time required by the task to complete. 167

WCRT

The WCRT of a task is the maximum duration between the activation of the task and the moment it finishes its execution. 167

Bibliography

[AS04] Karsten Albers and Frank Slomka. "An Event Stream Driven Approximation for the Analysis of Real-Time Systems". In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Catania, Italy: IEEE Computer Society, June 2004, pages 187–195. ISBN: 0-7695-2176-2. DOI: 10.1109/ECRTS.2004.4.
(Cited on page 37).

[ABD05] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. "An EDF-based scheduling algorithm for multiprocessor soft real-time systems". In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Balearic Islands, Spain: IEEE Computer Society, July 2005, pages 199–208. ISBN: 0-7695-2400-1. DOI: 10.1109/ECRTS.2005.6.
(Cited on pages 40, 41, 70).

[AB08] Björn Andersson and Konstantinos Bletsas. "Sporadic Multiprocessor Scheduling with Few Preemptions". In: *Proceedings of the 20th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Prague, Czech Republic: IEEE Computer Society, July 2008, pages 243–252. ISBN: 978-0-7695-3298-1. DOI: 10.1109/ECRTS.2008.9.
(Cited on page 43).

[ABB08] Björn Andersson, Konstantinos Bletsas, and Sanjoy K. Baruah. "Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors". In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Barcelona, Spain: IEEE Computer Society, Dec. 2008, pages 385–394. ISBN: 978-0-7695-3477-0. DOI: 10.1109/RTSS.2008.44.
(Cited on page 43).

[AT06] Björn Andersson and Eduardo Tovar. "Multiprocessor Scheduling with Few Preemptions". In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Embedded and Real-Time Computing Systems and Applications (RTCSA). Sydney, Australia, Aug. 2006, pages 322–334. ISBN: 0-7695-2676-4. DOI: 10.1109/RTCSA.2006.45.
(Cited on page 41).

[Aud01] Neil C. Audsley. "On priority asignment in fixed priority scheduling". In: *Information Processing Letters* 79.1 (May 2001), pages 39–44. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(00)00165-4.
(Cited on page 20).

[Aud91] Neil C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. Technical report. University of York, Nov. 1991.
(Cited on page 20).

[Aud+93] Neil C. Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software Engineering Journal* 8.5 (Sept. 1993), pages 284–292. ISSN: 0268-6961.
(Cited on pages 24, 58, 140, 150).

[Aud+91] Neil C. Audsley, Alan Burns, Mike Richardson, and Andy Wellings. "Hard Real-Time Scheduling: The Deadline-Monotonic Approach". In: *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems*. IEEE Workshop on Real-Time Operating Systems (RTOS). May 1991, pages 133–137.
(Cited on pages 20, 106, 123).

[Bad+12] Benjamin Bado, Laurent George, Pierre Courbin, and Joël Goossens. "A semi-partitioned approach for parallel real-time scheduling". In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Pont à Mousson, France: ACM, Nov. 2012, pages 151–160. ISBN: 978-1-4503-1409-1. DOI: 10.1145/2392987.2393006.
(Cited on pages xx, 158).

[Bak06] Theodore P. Baker. "A comparison of global and partitioned EDF schedulability tests for multiprocessors". In: *Proceedings of the 14th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Poitiers, France, May 2006, pages 119–127.
(Cited on pages 37, 53, 60, 86, 142, 143, 151).

[Bak05a] Theodore P. Baker. "An Analysis of EDF Schedulability on a Multiprocessor". In: *IEEE Transactions on Parallel and Distributed Systems* 16.8 (Aug. 2005), pages 760–768. ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.88.
(Cited on page 38).

[Bak05b] Theodore P. Baker. *Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time*. Technical report. Florida State University, 2005.
(Cited on page 53).

[Bak03] Theodore P. Baker. "Multiprocessor EDF and Deadline Monotonic Schedulability Analysis". In: *Proceedings of the 24th IEEE International Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Cancun, Mexico: IEEE Computer Society, Dec. 2003, pages 120–129. ISBN: 0-7695-2044-8. DOI: 10.1109/REAL.2003.1253260.
(Cited on page 38).

[BB09] Theodore P. Baker and Sanjoy K. Baruah. "An analysis of global edf schedulability for arbitrary-deadline sporadic task systems". In: *Real-Time Systems* 43.1 (Sept. 2009), pages 3–24. ISSN: 0922-6443. DOI: 10.1007/s11241-009-9075-8.
(Cited on pages 39, 150).

[BRC06] Patricia Balbastre, Ismael Ripoll, and Alfons Crespo. "Optimal deadline assignment for periodic real-time tasks in dynamic priority systems". In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dresden, Germany: IEEE Computer Society, July 2006, pages 65–74. ISBN: 0-7695-2619-5. DOI: 10.1109/ECRTS.2006.17.
(Cited on pages 32, 77).

[BRC02] Patricia Balbastre, Ismael Ripoll, and Alfons Crespo. "Schedulability analysis of window-constrained execution time tasks for real-time control". In: *Proceedings of the 14th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Vienna, Austria, June 2002, pages 11–18. ISBN: 0-7695-1665-3. DOI: 10.1109/EMRTS.2002.1019181.
(Cited on page 32).

[Bar07] Sanjoy K. Baruah. "Techniques for Multiprocessor Global Schedulability Analysis". In: *Proceedings of the 28th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Tucson, Arizona, USA: IEEE Computer Society, Dec. 2007, pages 119–128. ISBN: 0-7695-3062-1. DOI: 10.1109/RTSS.2007.35.
(Cited on pages 38, 39, 117).

[BF06] Sanjoy K. Baruah and Nathan W. Fisher. "The Partitioned Multiprocessor Scheduling of Deadline-Constrained Sporadic Task Systems". In: *IEEE Transactions on Computers* 55.7 (July 2006), pages 918–923. ISSN: 0018-9340. DOI: 10.1109/TC.2006.113.
(Cited on pages 37, 58, 150).

[BF07] Sanjoy K. Baruah and Nathan W. Fisher. "The partitioned dynamic-priority scheduling of sporadic task systems". In: *Real-Time Systems* 36.3 (Aug. 2007), pages 199–226. ISSN: 0922-6443. DOI: 10.1007/s11241-007-9022-5.
(Cited on page 37).

[BF05] Sanjoy K. Baruah and Nathan W. Fisher. "The partitioned multiprocessor scheduling of sporadic task systems". In: *Proceedings of the 26th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Miami, Florida, USA: IEEE Computer Society, Dec. 2005, pages 321–329. ISBN: 0-7695-2490-7. DOI: 10.1109/RTSS.2005.40.
(Cited on page 37).

[BGP95] Sanjoy K. Baruah, Johannes Gehrke, and Greg C. Plaxton. "Fast scheduling of periodic tasks on multiple resources". In: *Proceedings of the 9th International Parallel Processing Symposium*. International Parallel Processing Symposium (IPPS). Santa Barbara, California, USA: IEEE Computer Society, Apr. 1995, pages 280–288. ISBN: 0-8186-7074-6.
(Cited on pages 35, 37, 141, 150).

[BRH90] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor". In: *Real-Time Systems* 2.4 (Oct. 1990), pages 301–324. ISSN: 0922-6443. DOI: 10.1007/BF01995675.
(Cited on pages 15, 25, 58, 140, 150).

[Bar+09] Sanjoy K. Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. "Implementation of a Speedup Optimal Global EDF Schedulability Test". In: *Proceedings of the 21th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dublin, Ireland, July 2009, pages 259–268. ISBN: 978-0-7695-3724-5. DOI: 10.1109/ECRTS.2009.31.

(Cited on page 39).

[Bar+96] Sanjoy K. Baruah, Neil K. Cohen, Greg C. Plaxton, and Donald A. Varvel. "Proportionate progress: A notion of fairness in resource allocation". In: *Algorithmica* 15.6 (June 1996), pages 600–625. ISSN: 0178-4617. DOI: 10.1007/BF01940883.

(Cited on pages 35, 37).

[Bar13] Sanjoy Baruah. "Partitioned EDF scheduling: a closer look". In: *Real-Time Systems* 49.6 (Nov. 2013), pages 715–729. ISSN: 0922-6443. DOI: 10.1007/s11241-013-9186-0.

(Cited on pages 64, 156, 158).

[BB06] Sanjoy Baruah and Alan Burns. "Sustainable Scheduling Analysis". In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Rio de Janeiro, Brazil: IEEE Computer Society, Dec. 2006, pages 159–168. ISBN: 0-7695-2761-2. DOI: 10.1109/RTSS.2006.47.

(Cited on page 110).

[BBA10] Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson. "Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability". In: *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT). Brussels, Belgium, July 2010.

(Cited on pages 38, 127).

[BCG11] Vandy Berten, Pierre Courbin, and Joël Goossens. "Gang fixed priority scheduling of periodic moldable real-time tasks". In: *Proceedings of the Junior Researcher Workshop Session of the 19th International Conference on Real-Time and Network Systems*. Edited by Alan Burns and Laurent George. Real-Time and Network Systems (RTNS). Nantes, France, Sept. 2011, pages 9–12.
(Cited on pages xxi, 95, 98).

[Ber09] Marco Bertogna. "Evaluation of Existing Schedulability Tests for Global EDF". In: *Proceedings of the 38th IEEE International Conference on Parallel Processing Workshops*. Edited by Leonard Barolli and Wu chun Feng. IEEE International Conference on Parallel Processing Workshops (ICPPW). Vienna, Austria: IEEE Computer Society, Sept. 2009, pages 11–18. isbn: 978-0-7695-3803-7. doi: 10.1109/ICPPW.2009.12.
(Cited on pages 38, 39).

[BC07] Marco Bertogna and Michele Cirinei. "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms". In: *Proceedings of the 28th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Tucson, Arizona, USA: IEEE Computer Society, Dec. 2007, pages 149–160. isbn: 0-7695-3062-1. doi: 10.1109/RTSS.2007.41.
(Cited on pages 39, 116, 117, 140, 150).

[BCL05] Marco Bertogna, Michele Cirinei, and Giuseppe Lipari. "Improved Schedulability Analysis of EDF on Multiprocessor Platforms". In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Balearic Islands, Spain: IEEE Computer Society, July 2005, pages 209–218. isbn: 0-7695-2400-1. doi: 10.1109/ECRTS.2005.18.
(Cited on pages 39, 116).

[BB04] Enrico Bini and Giorgio C. Buttazzo. "Biasing Effects in Schedulability Measures". In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Catania, Italy: IEEE Computer Society, June 2004, pages 196–203. isbn: 0-7695-2176-2. doi: 10.1109/ECRTS.2004.7.
(Cited on pages 142, 151).

[BBB03] Enrico Bini, Giorgio C. Buttazzo, and Giuseppe M. Buttazzo. "Rate monotonic analysis: the hyperbolic bound". In: *IEEE Transactions on Computers* 52.7 (July 2003), pages 933–942. ISSN: 0018-9340. DOI: 10.1109/TC.2003.1214341.
(Cited on pages 24, 58, 64, 150).

[BGM07] Lamine Bougueroua, Laurent George, and Serge Midonnet. "Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF". In: *Proceedings of the 2nd IEEE International Conference on Systems*. International Conference on Systems (ICONS). Sainte-Luce, Martinique, France, Apr. 2007, pages 52–52. ISBN: 978-0-7695-2807-6. DOI: 10.1109/ICONS.2007.18.
(Cited on pages 31, 44).

[Bur+10] Alan Burns, Robert I. Davis, P. Wang, and Fengxiang Zhang. "Partitioned EDF Scheduling for Multiprocessors using a C=D Scheme". In: *Proceedings of the 18th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Toulouse, France, Nov. 2010, pages 169–178.
(Cited on pages 43, 140, 150).

[Cal+06] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. "$LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers". In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Rio de Janeiro, Brazil, Dec. 2006, pages 111–126. ISBN: 0-7695-2761-2. DOI: 10.1109/RTSS.2006.27.
(Cited on page 137).

[Cha+12] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms". In: *Proceedings of 3th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). Pisa, Italy, July 2012, pages 21–26.
(Cited on page 137).

[Cha+00] Robit Chandra et al. *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Oct. 2000. isbn: 1-55860-671-8.

(Cited on pages 5, 17).

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and Douglas E. Jensen. "An Optimal Real-Time Scheduling Algorithm for Multiprocessors". In: *Proceedings of the 27th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Rio de Janeiro, Brazil: IEEE Computer Society, Dec. 2006, pages 101–110. isbn: 0-7695-2761-2. doi: 10.1109/RTSS.2006.10.

(Cited on pages 35, 38).

[Cio86] Emile M. Cioran. *The Trouble With Being Born*. Edited by Seaver Books. First published in 1973 by Gallimard with title "De l'inconvénient d'être né". 1986. isbn: 1611454433.

(Cited on page 7).

[CCGG08] Sébastien Collette, Liliana Cucu-Grosjean, and Joël Goossens. "Integrating job parallelism in real-time scheduling theory". In: *Information Processing Letters* 106.5 (May 2008), pages 180–187. issn: 0020-0190. doi: 10.1016/j.ipl.2007.11.014.

(Cited on page 45).

[Cor92] Thinking Machines Corporation. *The Connection Machine CM-5: Technical Summary*. Thinking Machines Corporation, Jan. 1992.

(Cited on page 16).

[CKR09] Ayse K. Coskun, Andrew B. Kahng, and Tajana Simunic Rosing. "Temperature- and Cost-Aware Design of 3D Multiprocessor Architectures". In: *Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD). Patras, Greece, Aug. 2009, pages 183–190. isbn: 978-0-7695-3782-5. doi: 10.1109/DSD.2009.233.

(Cited on page 38).

[CG11] Pierre Courbin and Laurent George. "FORTAS : Framework fOr Real-Time Analysis and Simulation". In: *Proceedings of 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). Porto, Portugal, July 2011.
(Cited on page xxi).

[CLG13] Pierre Courbin, Irina Lupu, and Joël Goossens. "Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks". In: *Real-Time Systems* 49.2 (2013), pages 239–266. ISSN: 0922-6443. DOI: 10.1007/s11241-012-9173-x.
(Cited on pages xx, 94, 98, 104, 108, 123, 126).

[CGG11] Liliana Cucu-Grosjean and Joël Goossens. "Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms". In: *Journal of Systems Architecture* 57.5 (May 2011), pages 561–569. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2011.02.007.
(Cited on pages 12, 109).

[Dar58] Charles Robert Darwin. *Selected Letters on Evolution and Origin of Species*. In a letter from Erasmus Darwin, Charles Darwin's brother, page 227. Dover Publications, 1958. ISBN: 978-1-2580-3864-9.
(Cited on page 3).

[DB11] Robert I. Davis and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems". In: *ACM Computing Survey* 43.4 (Oct. 2011), 35:1–35:44. ISSN: 0360-0300. DOI: 10.1145/1978802.1978814.
(Cited on page 22).

[DGC10] Robert I. Davis, Laurent George, and Pierre Courbin. "Quantifying the Sub-optimality of Uniprocessor Fixed Priority Non-Pre-emptive Scheduling". In: *Proceedings of the 18th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Toulouse, France, Nov. 2010, pages 1–10.
(Cited on pages xx, 149).

[Dav+09] Robert I. Davis, Thomas Rothvoβ, Sanjoy K. Baruah, and Alan Burns. "Exact quantification of the sub-optimality of uniprocessor fixed priority pre-emptive scheduling". In: *Real-Time Systems* 43.3 (Nov. 2009), pages 211–258. ISSN: 0922-6443. DOI: 10.1007/s11241-009-9079-4.

(Cited on page 61).

[Der74] Michael L. Dertouzos. "Control Robotics: The Procedural Control of Physical Processes." In: *Proceedings of the International Federation for Information Processing*. International Federation for Information Processing (IFIP). Stockholm, Sweden: American Elsevier, Aug. 1974, pages 807–813. ISBN: 0-7204-2803-3.

(Cited on page 21).

[DM89] Michael L. Dertouzos and Aloysius K Mok. "Multiprocessor Online Scheduling of Hard-Real-Time Tasks". In: *IEEE Transactions on Software Engineering* 15.12 (Dec. 1989), pages 1497–1506. ISSN: 0098-5589. DOI: 10.1109/32.58762.

(Cited on page 106).

[DG00] Raymond Devillers and Joël Goossens. "Liu and Layland's schedulability test revisited". In: *Information Processing Letters* 73.5-6 (Mar. 2000), pages 157–161. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(00)00016-8.

(Cited on pages 58, 150).

[Dor+10] François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. "Semi-Partitioned Hard Real-Time Scheduling with Restricted Migrations upon Identical Multiprocessor Platforms". In: *Proceedings of the 18th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Toulouse, France, Nov. 2010, pages 207–216.

(Cited on page 41).

[Fei96] Dror G. Feitelson. "Packing Schemes for Gang Scheduling". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing of the 10th International Parallel Processing Symposium*. International Parallel Processing Symposium (IPPS). Honolulu, Hawaii, USA: Springer-Verlag, Apr. 1996, pages 89–110. ISBN: 3-540-61864-3.

(Cited on page 16).

[FBB06a] Nathan W. Fisher, Theodore P. Baker, and Sanjoy K. Baruah. "Algorithms for Determining the Demand-Based Load of a Sporadic Task System". In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Embedded and Real-Time Computing Systems and Applications (RTCSA). Sydney, Australia, Aug. 2006, pages 135–146. ISBN: 0-7695-2676-4. DOI: 10.1109/RTCSA.2006.12.
(Cited on page 25).

[FBB06b] Nathan W. Fisher, Sanjoy K. Baruah, and Theodore P. Baker. "The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities". In: *Proceedings of the 18th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dresden, Germany: IEEE Computer Society, July 2006, pages 118–127. ISBN: 0-7695-2619-5. DOI: 10.1109/ECRTS.2006.30.
(Cited on page 53).

[For87] Maxime Le Forestier. *Né quelque part*. Song composed and performed by Maxime Le Forestier and music composed with Jean-Pierre Sabard. Title of the album "Né quelque part". 1987.
(Cited on page xi).

[GC11] Laurent George and Pierre Courbin. "IGI Global". In: edited by Mohamed Khalgui and Hans-Michael Hanisch. IGI Global, 2011. Chapter Reconfiguration of Uniprocessor Sporadic Real-Time Systems: The Sensitivity Approach, pages 167–189. ISBN: 978-1-5990-4988-5. DOI: 10.4018/978-1-60960-086-0.ch007.
(Cited on pages xx, 149).

[GCS11] Laurent George, Pierre Courbin, and Yves Sorel. "Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling". In: *Journal of Systems Architecture* 57.5 (May 2011), pages 518–535. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2011.02.008.
(Cited on pages xx, 72, 83, 141, 149, 150).

[GH09a] Laurent George and Jean-François Hermant. "A Norm Approach for the Partitioned EDF Scheduling of Sporadic Task Systems". In: *Proceedings of the 21th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dublin, Ireland, July 2009, pages 161–169. ISBN: 978-0-7695-3724-5. DOI: 10.1109/ECRTS.2009.29.
(Cited on pages 25, 29, 36).

[GH09b] Laurent George and Jean-François Hermant. "Characterization of the Space of Feasible Worst-Case Execution Times for Earliest-Deadline-First Scheduling". In: *Journal of Aerospace Computing, Information and Communication (JACIC)* 6 (Nov. 2009), pages 604–623. ISSN: 2327-3097. DOI: 0.2514/1.44721.
(Cited on pages 26, 28, 37, 140).

[GRS96] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling*. Rapport de recherche RR-2966. Projet REFLECS. INRIA, Sept. 1996.
(Cited on page 20).

[GB10] Joël Goossens and Vandy Berten. "Gang FTP scheduling of periodic and parallel rigid real-time tasks". In: *Proceedings of the 18th International Conference on Real-Time and Network Systems*. Real-Time and Network Systems (RTNS). Toulouse, France, Nov. 2010, pages 189–196.
(Cited on pages 45, 97, 121, 123, 126).

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy K. Baruah. "Priority -Driven Scheduling of Periodic Task Systems on Multiprocessors". In: *Real-Time Systems* 25.2-3 (Sept. 2003), pages 187–205. ISSN: 0922-6443. DOI: 10.1023/A:1025120124771.
(Cited on page 38).

[GB98] Sergei Gorlatch and Holger Bischof. "A Generic MPI Implementation for a Data-Parallel Skeleton: Formal Derivation and Application to FFT". In: *Parallel Processing Letters* 8.4 (Mar. 1998), pages 447–458. DOI: 10.1142/S0129626498000456.
(Cited on page 5).

[GKP88] Ronald L. Graham, Donald Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison Wesley Publisher, Sept. 1988, page 638. ISBN: 0201142368.
(Cited on page 54).

[GLS00] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. 2nd. MIT Press, Jan. 2000. ISBN: 0-26257-132-3.
(Cited on page 5).

[Gua+09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. "New Response Time Bounds for Fixed Priority Multiprocessor Scheduling". In: *Proceedings of the 30th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Washington, DC, USA: IEEE Computer Society, Dec. 2009, pages 387–397. ISBN: 978-0-7695-3875-4. DOI: 10.1109/RTSS.2009.11.
(Cited on pages 114, 115, 117, 118, 121).

[HL94] Rhan Ha and Jane Win Shih Liu. "Validating timing constraints in multiprocessor and distributed real-time systems". In: *Proceedings of the 14th International Conference on Distributed Computing Systems*. International Conference on Distributed Computing Systems (ICDCS). Poznan, Poland, June 1994, pages 162–171. DOI: 10.1109/ICDCS.1994.302407.
(Cited on pages 110, 111).

[HP06] Sangchul Han and Minkyu Park. "Predictability of least laxity first scheduling algorithm on multiprocessor real-time systems". In: *Proceedings of the 2006 International Conference on Emerging Directions in Embedded and Ubiquitous Computing*. International Conference on Emerging Directions in Embedded and Ubiquitous Computing (EUC). Seoul, Korea: Springer-Verlag, Aug. 2006, pages 755–764. ISBN: 3-540-36850-7, 978-3-540-36850-2. DOI: 10.1007/11807964_76.
(Cited on page 44).

[Har+01] Michael González Harbour, José Javier Gutiérrez, José Carlos Palencia, and José María Drake. "MAST: Modeling and Analysis Suite for Real Time Applications". In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Delft, The Netherlands: IEEE Computer Society, June 2001, pages 125–134. ISBN: 0-7695-1221-6.

(Cited on page 136).

[Hla+07] Pierre-Emmanuel Hladik, Anne-Marie Déplanche, Sébastien Faucou, and Yvon Trinquet. "Adequacy between AUTOSAR OS specification and real-time scheduling theory". In: *Proceedings of the 2nd IEEE International Symposium on Industrial Embedded Systems*. IEEE International Symposium on Industrial Embedded Systems (SIES). Lisbon, Portugal, July 2007, pages 225–233. ISBN: 1-4244-0840-7. DOI: 10.1109/SIES.2007.4297339.

(Cited on page 44).

[HBL10] Martijn M.H.P. Holenderski Mike andvan den Heuvel, Reinder J. Bril, and Johan J. Lukkien. "GRASP: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems". In: *Proceedings of 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). Brussels, Belgium, July 2010.

(Cited on page 137).

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. "On non-preemptive scheduling of period and sporadic tasks". In: *Proceedings of the 12th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). San Antonio, Texas, USA, Dec. 1991, pages 129–139. ISBN: 0-8186-2450-7. DOI: 10.1109/REAL.1991.160366.

(Cited on page 21).

[Joh74] David S. Johnson. "Fast algorithms for bin packing". In: *Journal of Computer and System Sciences* 8.3 (June 1974), pages 272–314. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(74)80026-7.

(Cited on pages 36, 50).

[Jol12] Alexandre Jollien. *Petit traité de l'abandon : Pensées pour accueillir la vie telle qu'elle se propose*. Edited by Seuil. Quote on page 115 or or on the track "22–Zen" of the audio CD from 7'43. 2012. isbn: 978-2-0210-7941-8.

(Cited on page 49).

[JP86] Mathai Joseph and Paritosh K. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29 (5 1986), pages 390–395. doi: 10.1093/comjnl/29.5.390.

(Cited on pages 24, 58).

[KI09] Shinpei Kato and Yutaka Ishikawa. "Gang EDF Scheduling of Parallel Task Systems". In: *Proceedings of the 30th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Washington, DC, USA: IEEE Computer Society, Dec. 2009, pages 459–468. isbn: 978-0-7695-3875-4. doi: 10.1109/RTSS.2009.42.

(Cited on pages 16, 17, 45, 95).

[KRI09] Shinpei Kato, Ragunathan Rajkumar, and Yutaka Ishikawa. *A Loadable Real-Time Scheduler Suite for Multicore Platforms*. Technical Report CMUECE-TR09-12. University of Tokyo and Carnegie Mellon University, Dec. 2009.

(Cited on page 137).

[KY08a] Shinpei Kato and Nobuyuki Yamasaki. "Portioned EDF-based scheduling on multiprocessors". In: *Proceedings of the 8th ACM International Conference on Embedded Software*. ACM International Conference on Embedded Software (EMSOFT). Atlanta, Georgia, USA: ACM, Oct. 2008, pages 139–148. isbn: 978-1-60558-468-3. doi: 10.1145/1450058.1450078.

(Cited on page 42).

[KY08b] Shinpei Kato and Nobuyuki Yamasaki. "Portioned static priority scheduling on multiprocessors". In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IEEE International Symposium on Parallel and Distributed Processing (IPDPS). Miami, Florida, USA, Apr. 2008, pages 1–12. doi: 10.1109/IPDPS.2008.4536299.

(Cited on page 42).

[KY07] Shinpei Kato and Nobuyuki Yamasaki. "Real-Time Scheduling with Task Splitting on Multiprocessors". In: *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). Daegu, Korea: IEEE Computer Society, Aug. 2007, pages 441–450. ISBN: 0-7695-2975-5. DOI: 10.1109/RTCSA.2007.61. (Cited on page 42).

[KY08c] Shinpei Kato and Nobuyuki Yamasaki. "Semi-Partitioning Technique for Multiprocessor Real-Time Scheduling". In: *Proceedings of the WIP Session of the 29th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Barcelona, Spain: IEEE Computer Society, Dec. 2008, page 4. (Cited on pages 41, 42).

[KY09] Shinpei Kato and Nobuyuki Yamasaki. "Semi-partitioned Fixed Priority Scheduling on Multiprocessors". In: *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). San Francisco, California, USA, Apr. 2009, pages 23–32. ISBN: 978-0-7695-3636-1. DOI: 10.1109/RTAS.2009.9. (Cited on page 43).

[KYI09] Shinpei Kato, Nobuyuki Yamasaki, and Yutaka Ishikawa. "Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors". In: *Proceedings of the 21th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dublin, Ireland, July 2009, pages 249–258. ISBN: 978-0-7695-3724-5. DOI: 10.1109/ECRTS.2009.22. (Cited on pages 36, 43, 76, 79, 84, 140, 150).

[LKR10] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. "Scheduling Parallel Real-Time Tasks on Multi-core Processors". In: *Proceedings of the 31th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). San Diego, California, USA, Dec. 2010, pages 259–268. ISBN: 978-0-7695-4298-0. DOI: 10.1109/RTSS.2010.42. (Cited on pages 17, 18, 45).

[LRL09] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. "Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors". In: *Proceedings of the 21th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Dublin, Ireland: IEEE Computer Society, July 2009, pages 239–248. ISBN: 978-0-7695-3724-5. DOI: 10.1109/ECRTS.2009.33.

(Cited on page 43).

[LMM98] Sylvain Lauzac, Rami Melhem, and Daniel Mossé. "An efficient RMS admission control and its application to multiprocessor scheduling". In: *Proceedings of the 12th International Parallel Processing Symposium*. International Parallel Processing Symposium (IPPS). Orlando, Florida, USA, Mar. 1998, pages 511–518. DOI: 10.1109/IPPS.1998.669964.

(Cited on pages 58, 150).

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. "The Rate Monotonic scheduling algorithm: exact characterization and average case behavior". In: *Proceedings of the 10th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Santa Monica, California, USA: IEEE Computer Society, Dec. 1989, pages 166–171. ISBN: 0-8186-2004-8. DOI: 10.1109/REAL.1989.63567.

(Cited on page 15).

[Leu89] Joseph Y. Leung. "A new algorithm for scheduling periodic real-time tasks". In: *Algorithmica* 4.1-4 (June 1989), pages 209–219. ISSN: 0178-4617. DOI: 10.1007/BF01553887.

(Cited on pages 21, 106).

[LM80] Joseph Y. Leung and Maggie L. Merrill. "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks". In: *Information Processing Letters* 11.3 (1980), pages 115–118. DOI: 10.1016/0020-0190(80)90123-4.

(Cited on page 20).

[LL73] Chung Laung Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *Journal of ACM* 20.1 (Jan. 1973), pages 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743.

(Cited on pages 12, 20, 21, 23, 24, 57, 58, 106, 140, 150).

[Liu00] Jane Win Shih Liu. *Real-Time Systems*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, Apr. 2000. ISBN: 0130996513.
(Cited on pages 24, 37).

[LDG04] José M. López, José L. Díaz, and Daniel F. García. "Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems". In: *Real-Time Systems* 28.1 (Oct. 2004), pages 39–68. ISSN: 0922-6443. DOI: 10.1023/B:TIME.0000033378.56741.14.
(Cited on page 36).

[LG11] Irina Lupu and Joël Goossens. "Scheduling of Hard Real-Time Multi-Thread Periodic Tasks". In: *Proceedings of the 19th International Conference on Real-Time and Network Systems*. Edited by Sébastien Faucou, Alan Burns, and Laurent George. Real-Time and Network Systems (RTNS). Nantes, France, Sept. 2011, pages 35–44.
(Cited on pages 98, 104, 108, 123, 126, 132).

[Lup+10] Irina Lupu, Pierre Courbin, Laurent George, and Joël Goossens. "Multi-criteria evaluation of partitioning schemes for real-time systems". In: *Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*. Emerging Technologies and Factory Automation (ETFA). Bilbao, Spain: IEEE Computer Society, Sept. 2010, pages 1–8. ISBN: 978-1-4244-6848-5. DOI: 10.1109/ETFA.2010.564121.
(Cited on pages xx, 59, 149).

[MMR98] Govindarasu Manimaran, C. Siva Ram Murthy, and Krithi Ramamritham. "A New Approach for Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems". In: *Real-Time Systems* 15.1 (July 1998), pages 39–60. ISSN: 0922-6443. DOI: 10.1023/A:1008022923184.
(Cited on page 44).

[MSD10] Thomas Megel, Renaud Sirdey, and Vincent David. "Minimizing Task Preemptions and Migrations in Multiprocessor Optimal Real-Time Schedules". In: *Proceedings of the 31th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). San Diego, California, USA, Dec. 2010, pages 37–46. ISBN: 978-0-7695-4298-0. DOI: 10.1109/RTSS.2010.22.
(Cited on page 39).

[Mok83] Aloysius Ka-Lau Mok. "Fundamental design problems of distributed systems for the hard-real-time environment". PhD thesis. Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, May 1983.
(Cited on page 21).

[Moo03] Gordon Earle Moore. "No exponential is forever: but "Forever" can be delayed!" In: *Proceedings of the 50th IEEE International Solid-State Circuits Conference*. Volume 1. IEEE International Solid-State Circuits Conference (ISSCC). San Francisco, California, USA, Feb. 2003, pages 20–23. isbn: 0-7803-7707-9. doi: 10.1109/ISSCC.2003.1234194.
(Cited on page 5).

[Nel13] Geoffrey Nelissen. "Efficient Optimal Multiprocessor Scheduling Algorithms for Real-Time Systems". PhD thesis. Université Libre de Bruxelles, Aug. 2013.
(Cited on page 158).

[Nel+11] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. "Reducing Preemptions and Migrations in Real-Time Multiprocessor Scheduling Algorithms by Releasing the Fairness". In: *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Embedded and Real-Time Computing Systems and Applications (RTCSA). Toyama, Japan: IEEE Computer Society, Aug. 2011, pages 15–24. isbn: 978-0-7695-4502-8. doi: 10.1109/RTCSA.2011.57.
(Cited on pages 39, 150).

[Nel+12] Geoffrey Nelissen, Vandy Berten, Vincent Nelis, Joël Goossens, and Dragomir Milojevic. "U-EDF: An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks". In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems*. EuroMicro Conference on Real-Time Systems (ECRTS). Pisa, Italy: IEEE Computer Society, July 2012, pages 13–23. isbn: 978-1-4673-2032-0. doi: 10.1109/ECRTS.2012.36.
(Cited on pages 39, 150).

[Reg+11] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. "RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor". In: *Proceedings of the 32th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Vienna, Austria: IEEE Computer Society, Nov. 2011, pages 104–115. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.17.

(Cited on pages 40, 150).

[Rei07] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, July 2007, pages I–XXV, 1–303. ISBN: 978-0-596-51480-8.

(Cited on page 5).

[Roz67] Stefan Rozental. *Niels Bohr: His Life and Work as Seen by His Friends and Colleagues*. Edited by North Holland Publishing Co. Quote from Hans Henrik Bohr writing about his father Niels Bohr in the "My father" section of the book. 1967.

(Cited on page 155).

[Sai+11] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. "Multi-core Real-Time Scheduling for Generalized Parallel Task Models". In: *Proceedings of the 32th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). Vienna, Austria: IEEE Computer Society, Nov. 2011, pages 217–226. ISBN: 978-0-7695-4591-2. DOI: 10.1109/RTSS.2011.27.

(Cited on page 45).

[SE48] Antoine de Saint-Exupéry. *Citadelle*. Collection Folio. Quote from Section LXXXVII. Editions Gallimard, 1948. ISBN: 978-2-0704-0747-7.

(Cited on page vii).

[SE39] Antoine de Saint-Exupéry. *Terre des hommes*. Quote on page 59 of the original version. Le Livre de Poche, 1939. ISBN: 978-2-0703-6021-5.

(Cited on page xii).

[SCG12] Vincent Sciandra, Pierre Courbin, and Laurent George. "Application of mixed-criticality scheduling model to intelligent transportation systems architectures". In: *Proceedings of the WIP Session of the 33th IEEE Real-Time Systems Symposium*. IEEE Real-Time Systems Symposium (RTSS). San Juan, Puerto Rico: ACM, Dec. 2012, pages 22–22. DOI: 10.1145/2518148.2518160.
(Cited on pages xxi, 159).

[Sin+04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. "Cheddar: a flexible real time scheduling framework". In: *ACM SIGAda Ada Letters* XXIV (4 Nov. 2004), pages 1–8. ISSN: 1094-3641. DOI: 10.1145/1046191.1032298.
(Cited on page 136).

[Sut05] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (Mar. 2005), pages 202–210.
(Cited on page 5).

[TC94] Ken Tindell and John Clark. "Holistic schedulability analysis for distributed hard real-time systems". In: *Microprocessors and Microprogramming* 40.2-3 (Apr. 1994), pages 117–134. ISSN: 0165-6074. DOI: 10.1016/0165-6074(94)90080-9.
(Cited on page 77).

[UDT10] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. "STORM : A Simulation Tool for Real-time Multiprocessor Scheduling Evaluation". In: *Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*. Emerging Technologies and Factory Automation (ETFA) MF-000477. Bilbao, Spain: IEEE Computer Society, Sept. 2010. ISBN: 978-1-4244-6848-5. DOI: 10.1109/ETFA.2010.5641179.
(Cited on page 137).

[Woo91] John F. Woods. *Usage of comma operator*. 1991. URL: https://groups.google.com/d/msg/comp.lang.c++/rYCO5yn4lXw/oITtSkZOtoUJ.
(Cited on page 135).

[ZB09] Fengxiang Zhang and Alan Burns. "Schedulability Analysis for Real-Time Systems with EDF Scheduling". In: *IEEE Transactions on Computers* 58.9 (Sept. 2009), pages 1250–1258. ISSN: 0018-9340. DOI: 10.1109/TC.2009.58.

(Cited on page 140).

Scheduling Sequential or Parallel Hard Real-Time Pre-emptive Tasks upon Identical Multiprocessor platforms

Abstract

The scheduling of tasks on a hard real-time system consists in finding a way to choose, at each time instant, which task should be executed on the processor so that each succeed to complete its work before its deadline.

In the uniprocessor case, this problem is already well studied and enables us to do practical applications on real systems (aerospace, stock exchange etc.). Today, multiprocessor platforms are widespread and led to many issues such as the effective use of all processors.

In this thesis, we explore the existing approaches to solve this problem. We first study the partitioning approach that reduces this problem to several uniprocessor systems and leverage existing research. For this one, we propose a generic partitioning algorithm whose parameters can be adapted according to different goals. We then study the semi-partitioning approach that allows migrations for a limited number of tasks. We propose a solution with restricted migration that could be implemented rather simply on real systems. We then propose a solution with unrestricted migration which provides better results but is more difficult to implement.

Finally, programmers use more and more the concept of parallel tasks that can use multiple processors simultaneously. These tasks are still little studied and we propose a new model to represent them. We study the possible schedulers and define a way to ensure the schedulability of such tasks for two of them.

Keywords: real-time, scheduling, multiprocessor, parallel, fork-join, gang, thread, migration, partitioning, semi-partitioning, global

---

Ordonnancement de Tâches Temps Réel Dures Préemptives Séquentielles ou Parallèles sur plateformes multiprocesseur identique

Résumé

L'ordonnancement de tâches sur un système temps réel dur correspond à trouver une façon de choisir, à chaque instant, quelle tâche doit être exécutée sur le processeur pour que chacune ait le temps de terminer son travail avant son échéance.

Ce problème, dans le contexte monoprocesseur, est déjà bien étudié et permet des applications sur des systèmes en production (aérospatiale, bourse etc.). Aujourd'hui, les plate-formes multiprocesseur se sont généralisées et ont amené de nombreuses questions telle que l'utilisation efficace de tous les processeurs.

Dans cette thèse, nous explorons les approches existantes pour résoudre ce problème. Nous étudions tout d'abord l'approche par partitionnement qui consiste à utiliser les recherches existantes en ramenant ce problème à plusieurs systèmes monoprocesseur. Ici, nous proposons un algorithme générique dont les paramètres sont adaptables en fonction de l'objectif à atteindre. Nous étudions ensuite l'approche par semi-partitionnement qui permet la migration d'un nombre restreint de tâches. Nous proposons une solution avec des migrations restreintes qui pourrait être assez simplement implémentée sur des systèmes concrets. Nous proposons ensuite une solution avec des migrations non restreintes qui offre de meilleurs résultats mais est plus difficile à implémenter.

Enfin, les programmeurs utilisent de plus en plus le concept de tâches parallèles qui peuvent utiliser plusieurs processeurs en même temps. Ces tâches sont encore peu étudiées et nous proposons donc un nouveau modèle pour les représenter. Nous étudions les ordonnanceurs possibles et nous définissons une façon de garantir l'ordonnançabilité de ces tâches pour deux d'entre eux.

Mots-clés : temps réel, ordonnancement, multiprocesseur, parallèlle, fork-join, gang, thread, migration, partitionnement, semi-partitionnement, global