UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Implementing a Dynamic and Global Scheduling Algorithm in a Real-Time OS

Arabella Brayer

Promoteur : Joël Goossens

Travail préliminaire au mémoire
Master 1
Science de l'informatique

Academic year 2016 - 2017

Table des matières

Chapitre 1

Introduction

1.1 Présentation générale du sujet

Dans le paysage quotidien des appareils informatiques, il existe toujours plus de systèmes embarqués. Parmi eux, un nombre très important de systèmes temps réel qui nécessitent des ordonnanceurs adaptés. Ce champ de recherche a été largement nourri durant les vingt dernières années par de nombreuses publications scientifiques.

Les systèmes embarqués n'ont pas toujours de grands besoins en efficacité, ils doivent principalement être sûrs, et réactifs. Toutefois, un système dont l'efficacité n'est pas maximale n'utilise pas ses ressources de façon optimale. De nos jours où la gestion énergétique se doit d'être la plus économe possible, où l'on est de plus en plus exigeant concernant l'efficacité, il est envisageable que la solution évolue du côté des industries. En effet,

Par ailleurs, cela fait plusieurs années que la croissance de l'efficacité des appareils n'est plus principalement due à des améliorations physiques des composants.
En effet, la multiplication des processeurs dans les ordinateurs a permis de paralléliser les exécutions, et c'est ainsi que la quasi stagnation des technologies a pu être contournée pour continuer la progression.
Le monde des systèmes embarqués en temps réel a bénéficié de ces améliorations, et dispose également de processeurs multi-cœurs. Cependant, leur gestion n'est à l'heure actuelle pas optimale, la plupart des systèmes sont gérés soit comme des systèmes mono-processeurs, soit avec des algorithmes partitionnés [38]. Dans le premier cas, les ressources ne sont pas utilisées de façon optimale, et dans le second, des implémentations et tests ont montré empiriquement que les algorithmes globaux pouvaient présenter des avantages intéressants, comme une meilleure répartition de l'utilisation des processeurs [5].

Le décalage entre la connaissance scientifique et les implémentations réelles peut s'expliquer en partie par le fait qu'il soit compliqué de gérer le partage des ressources, et que cette complexité n'apporte pas suffisamment d'avantages à l'heure qu'il est.

Le fait que l'industrie n'implémente pas à ce jour de solutions plus "performantes" pour les systèmes embarqués pose plusieurs problèmes :

1. Le matériel n'est pas exploité de façon optimale.
2. Par conséquent, on utilise bien plus d'énergie que nécessaire. À l'heure actuelle, dans un monde où l'on cherche à consommer le moins possible, cela pose question.

Mais au delà, cela signifie également plus de maintenance sur ces appareils parfois sans source de renouvellement d'énergie.

3. Certains systèmes embarqués nécessitent une très basse consommation car ils sont difficiles d'accès ou non faciles à recharger.

Toutes ces raisons poussent à s'intéresser à une implémentation réelle et réaliste d'ordonnanceurs connus dans la littérature, mais moins dans la réalité.
C'est dans ce contexte que s'inscrit ce travail d'implémentation d'un ordonnanceur en temps réel global et multiprocesseur, dont un des objectifs est d'améliorer la connaissance pratique d'ordonnanceurs multiprocesseurs globaux, d'en apercevoir les limites. Cette implémentation pourrait amener plusieurs résultats intéressants :

— Une comparaison entre des résultats théoriques et l'effet de leur mise en œuvre
— L'origine de ces différences
— Vérifier les avantages, inconvénients ou obstacles à la commercialisation de telles solutions.

1.2 Contexte et objectifs

1.2.1 Ordonnanceurs globaux

Comme il sera expliqué plus tard dans ce document, il existe deux grandes familles d'ordonnanceurs multiprocesseur :

— Les ordonnanceurs partitionnés
— Les ordonnanceurs globaux

Si parmi ces deux familles d'ordonnanceurs, les systèmes mono-processeurs, voire partitionnés ont le plus de succès auprès de l'industrie, ce n'est pas la famille la plus efficace pour tout type de classe de tâches. Les algorithmes globaux répartissent habituellement mieux l'utilisation des processeurs, les migrations sont possibles et donc les temps de réalisation sont généralement moins grands.

En 1988, Hong and Leung [22] publient un article dans lequel est démontré que :
"For every $m > 1$, no optimal on-line scheduler can exist for task systems with two or more distinct deadlines" : *Il n'existe pas d'ordonnanceur multiprocesseur en ligne optimal pour un système de tâches avec plusieurs échéances distinctes*, donc un système de tâches dites "sporadiques".
Or, les systèmes embarqués doivent très souvent exécuter des systèmes de tâches sporadiques, cela tient de leur nature : la plupart d'entre eux attendent en effet de recevoir des signaux, qui arrivent de façon indéterminée.
Toutefois, des publications ultérieures viennent compléter cette preuve, et démontrent que des ordonnanceurs globaux optimaux existent, mais nécessitent de la clairvoyance.

1.2.2 HIPPEROS

HIPPEROS (High Performance Parallel Embedded Real-time Operating Systems) est un **RTOS** développé depuis plusieurs années par une spinoff de l'ULB. Il bénéficie des connaissances apportées par le monde de la recherche dans le domaine des systèmes critiques avec multicœurs. Une de ses particularités est sa modularité, qui permet d'adapter

ses possibilités en fonction du système lors de la compilation de l'OS, ainsi peut-on différencier deux installations en fonction des particularités.
HIPPEROS est un candidat idéal pour l'implémentation d'un ordonnanceur global, mais une partie du travail consistera à tirer parti de ses particularités. Par exemple, ce système d'exploitation gère les cœurs en leur attribuant des niveaux différents. L'un est considéré comme "maître" et les autres comme "esclaves". Ceci peut apporter un comportement particulier, ce auquel il convient d'apporter l'attention nécessaire. En résumé, une nouvelle implémentation sur un OS différent peut elle-aussi apporter à la connaissance générale des détails importants.

1.3 Problématique

L'ordonnanceur idéal – c'est à dire optimal pour toute classe de tâches et qui ne fasse pas de multiples migrations coûteuses n'existant pas, notre sujet est de sélectionner l'un d'eux parmi une liste d'ordonnanceurs connus (ne serait-ce que dans la littérature) afin que cela apporte à la connaissance générale, tant sur le plan théorique que pratique.
Par ailleurs, certains des ordonnanceurs présentés dans l'état de l'art ont déjà bénéficié d'une implémentation sur un **RTOS**, et une nouvelle implémentation sur un **OS** différent pourrait apporter un éclairage intéressant quant aux différences de comportements. Dans certaines présentations d'ordonnanceurs sont proposées des hypothèses à propos du comportement, comme faire globalement plus de préemption, de migrations. Le comportement observé dans des conditions différentes pourrait être vérifié, ou le contraire.
La question cruciale de cette première partie de travail consiste donc à faire un tour d'horizon de la littérature afin de pouvoir sélectionner un ordonnanceur pertinent à implémenter sur le **RTOS HIPPEROS**.

1.4 Vocabulaire général sur les systèmes temps réel

1.4.1 Système de tâches

Un système de tâches et un ensemble de tâches ayant les mêmes propriétés devant être exécutées.

1.4.2 Tâche (temps réel)

Une tâche correspond à une sorte de programme, c'est à dire une série d'instructions qui doivent être exécutées par un processeur. Il y a plusieurs caractéristiques et propriétés qui définissent une tâche, que nous allons définir ici :

Temps de réalisation : C'est le moment t où une tâche τ peut commencer à être exécutée.

Début, Fin : Le moment où la tâche commence effectivement à être exécutée, ou termine son exécution.

Temps de réponse : Temps maximum nécessaire à la réalisation de la tâche entre le temps de réalisation et Fin.

Échéance, temps limite : Correspond au temps limite au delà duquel la tâche doit avoir été exécutée. Il existe deux types d'échéances : une absolue, et une relative. L'échéances relative est une valeur fixe qui est une propriété de la tâche, elle dépend donc du temps de réalisation.

L'échéances absolue quant à elle est calculée avant l'exécution, et correspond à un temps t absolu.

Worst Case Execution Time Afin de faciliter les calculs et l'abstraction du problème, l'on pose habituellement que le temps considéré d'exécution de la tâche est le pire temps. Cela permet d'envisager le pire scénario, et ainsi de s'assurer que si celui-ci est possible, alors dans de meilleures conditions, la faisabilité est conservée. La notation utilisée dans le reste de ce document est **WCET** : **W**orst **C**ase **E**xecution **T**ime

Laxité La laxité d'une tâche est la durée entre sa réalisation et son échéance. Pour une tâche τ_i, la laxité est :

$$L_i = D_i - C_i$$

Tâches périodiques Une tâche périodique est une tâche qui génère régulièrement des jobs. Formellement, une tâche périodique est définie par un 4-uplet (O, T, D, C) où

- O est l'"offset", c'est à dire le temps que met la tâche à générer un premier job.
- T est la période, c'est à dire le temps qui sépare deux générations de job par la tâche. Comme le premier job est généré à l'instant O, alors $\forall i \in \{0, \infty\} t = O + i \times T$ où t est le temps où la tâche est générée la $i$ème fois.
- D est l'échéance relative, c'est à dire le temps qui sépare au maximum la génération d'un job et sa réalisation.
- C est le temps de réalisation. Dans les algorithmes, on utilise – comme expliqué précédemment – le **WCET**.

Dans le cas de tâches périodiques, on distingue trois cas différents :

1. si l'échéance est égale à la période $\forall \tau_i, D_i = T_i$: tâche à échéance sur requête
2. si l'échéance est inférieure ou égale à la période $\forall \tau_i, D_i \leq T_i$, on dit que la tâche est à échéance contrainte
3. s'il n'y a pas de contrainte particulière, la tâche est dite "à échéance arbitraire".

Les solutions d'ordonnancement pour ces trois types de tâches diffèrent donc. Globalement, on peut ordonner ces trois types de tâches :
échéance sur requêtes $\subset$ échéance contrainte $\subset$ échéance arbitraire.

Tâche sporadique : Une tâche sporadique est une tâche qui génère de nouveaux jobs, comme dans le cas de la tâche périodique. La différence entre ces deux types est que la tâche sporadique génère deux jobs avec un intervalle de temps au moins égal à la durée correspondant à sa période, et pas exactement égal. Les systèmes embarqués ont souvent des tâches sporadiques à gérer : en effet, un système qui est composé de capteur aura du mal à prévoir l'arrivée d'un signal de celui-ci.

Tâche apériodique : Pour ce type de tâches, on ignore la régularité de l'arrivée de nouveaux jobs. Les propriétés du job ne sont connues que lorsqu'un job est généré dans le système.

1.4.3 Job

Un job est une instance d'une tâche.

1.4.4 RTOS

(Real Time Operating System) Système d'Exploitation Temps Réel.
Un système d'exploitation Temps Réel un système d'exploitation implémenté pour les systèmes à temps réel, c'est à dire dont l'objectif est d'assurer le respect de certaines échéances.
Cela concerne pour beaucoup les systèmes dits "critiques", c'est à dire dont la sécurité est primordiale (avions, centrales nucléaires, pacemakers, etc.). Ce type de dispositifs est actuellement très répandu, notamment dans les voitures.
Les contraintes sont différentes de celles des systèmes d'exploitations traditionnels puisqu'on doit garantir le respect des échéances associées à chacune des tâches.
Cependant, si les systèmes d'exploitation temps réel ont des besoins de respect des échéances, cela ne signifie pas qu'ils aient forcément de grands besoins en efficacité ou en rapidité, ce qui est primordial est que les échéances soient respectées. Pour certains systèmes critiques, on comprend qu'il soit important de pouvoir prouver minutieusement que le système est correct, et ne risque pas de produire des pannes, ce qui peut s'avérer très grave.

1.4.5 Contraintes strictes, contraintes relatives

Dans le cas strict, le système doit impérativement respecter tous les temps limites. Aucun dépassement n'est toléré. Dans le cas des contraintes relatives, ce respect est moins impératif, et on pourra dépasser les délais occasionnellement.

1.4.6 Ordonnanceur

Un ordonnanceur (*Scheduler*) est la partie logicielle de l'*OS* chargée d'orchestrer l'ordre d'exécution des tâches du système selon des priorités fixées à l'avance ou durant l'exécution. On distingue trois types d'assignation de priorités des ordonnanceurs :

1. Priorité fixée au niveau des tâches : Ce type d'ordonnanceur fixe la priorité avant l'exécution, et celle-ci dépend donc des attributs de la tâche. Deux exemples d'ordonnanceurs de ce type seront présentés plus loin : Rate Monotonic et Deadline Monotonic.
2. Priorité fixe au niveau des jobs : La priorité est fixée par l'ordonnanceur à l'arrivée du job dans l'exécution, et elle ne peut pas changer jusqu'à la réalisation du job. Un exemple sera présenté plus loin : Earliest Deadline First.
3. Priorité dynamique : L'ordonnanceur peut recalculer à tout moment de l'exécution la priorité du job avec un exemple connu, Least Laxity First[21].

Selon les cas, l'ordonnanceur peut avoir à gérer un seul processeur. Dans ce document, on parlera dans ce cas d'ordonnanceur mono-processeur. Toutefois, de plus en plus de systèmes possèdent plusieurs processeurs, et il existe deux grandes familles d'ordonnanceurs associés à ces systèmes : les Partitionnés, ou les Globaux.

1.4.7 Work-Conservative

Un ordonnanceur peut-être "économe", ou "non-économe". Dans le premier cas, si le processeur est libre, et qu'une tâche est prête à être exécutée, l'ordonnanceur donnera l'instance : cela signifie qu'on évite les temps d'inactivité du processeur. Dans le second cas, une tâche ne sera pas toujours ordonnancée. Dans le reste de ce document, les ordonnanceurs présentés sont tous économes selon cette définition.

1.4.8 En ligne, hors ligne

Un ordonnanceur en ligne fait ses opérations d'ordonnancement durant le runtime. Un ordonnanceur hors ligne fait des opérations au préalable. Un exemple pour illustrer cela est un ordonnanceur partitionné, qui aura partagé le système en sous-systèmes durant une phase hors-ligne, et un ordonnanceur global non clairvoyant qui ordonnance au fur et à mesure durant l'exécution.

Ordonnanceur multi-processeur Partitionné

Un ordonnanceur partitionné composé de n tâches et de m processeurs divise le système en m sous-systèmes qui seront ensuite ordonnancés par autant d'ordonnanceurs mono-processeurs. Le problème principal consiste à découper efficacement le système de tâches.

Ordonnanceur multi-processeur Global

Un ordonnanceur global ne sépare pas le système en sous-systèmes mais gère une queue de tâches prêtes à être exécutées. Ainsi à chaque instant où il devra prendre une décision d'ordonnancement, il devra ordonnancer toute la liste des m processeurs.
Contrairement aux ordonnanceurs partitionnés, les migrations sont possibles. Le problème est donc de réduire leur nombre pour ne pas faire de mouvements inutilement.

1.4.9 Préemption et hypothèse

Un système est dit préemptif s'il a la capacité de mettre l'exécution d'un job en pause et d'exécuter un autre à la place. Une hypothèse pour simplifier la question théorique des ordonnanceur est très souvent faite dans la littérature : on considère le temps de préemption comme nul. Cette hypothèse est bien entendu fausse, et il faudra en tenir compte, particulièrement dans ce travail où l'on tente de faire le lien entre la théorie et la mise en pratique.

1.4.10 Utilisation

Le facteur d'utilisation U_i d'une tâche périodique τ_i est le rapport entre son temps d'exécution et sa période. Par exemple, pour une tâche τ_i tel que $WCET = 50ms$ et $T = 30ms$, le processeur doit allouer $\frac{30}{50}$ du temps total d'exécution à cette tâche.

L'utilisation (ou le facteur d'utilisation) d'un système périodique est la proportion de temps passée par le processeur à l'exécution de tâches d'un système donné. Le calcul de l'utilisation permet dans certains cas et certaines conditions de donner une indication à propos de la faisabilité du système par un certain ordonnanceur.

Liu et Layland [30] en donnent une définition formelle dans leur article décrivant l'ordonnanceur **Rate Monotonic** (présenté plus tard dans ce document).

Soit C_i le temps d'exécution d'une tâche τ_i, et soit T_i sa période, voici la définition de l'utilisation pour un système de m tâches :

$$U_{tot} = \sum_{i=1}^{m} (\frac{C_i}{T_i})$$

1.4.11 Laxité

La laxité d'un job est la durée entre son temps de réalisation et son échéance absolue.

Chapitre 2

État de l'art

La littérature sur le sujet des ordonnanceurs est assez vaste. Ceux-ci sont composés principalement de trois grandes familles :

— Les mono-processeurs
— Les multi-processeurs Partitionnés
— Les multiprocesseurs Globaux

Les ordonnanceurs mono-processeurs étant plus simples à appréhender, nous commencerons par présenter cette famille.

2.1 Ordonnanceurs mono-processeur

2.1.1 Rate Monotonic

L'ordonnanceur **R**ate **M**onotonic (RM) est décrit par Liu et Layland [30] en 1973. C'est donc un ordonnanceur classique, bien connu, ainsi que largement documenté [25]. L'idée de base est qu'une tâche avec une période courte évolue rapidement, et doit donc être prioritaire.
On considère un ensemble de tâches périodiques et indépendantes. Les priorités sont statiques et attribuées en fonction de la durée de la période, ainsi la tâche avec la période la plus faible sera de priorité plus élevée. Un des points fondamentaux de l'article est la preuve de l'optimalité de RM dans le cas préemptif, à tâches périodiques, à échéance sur requête, indépendantes, simultanées. Les auteurs énoncent même une condition suffisante de faisabilité pour un système à m tâches :

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m(2^{\frac{1}{m}} - 1)$$

Notons également que pour $n \to \infty$, $\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m(2^{\frac{1}{m}} - 1) \to \ln(2) \approx 0.69$. Une condition suffisante est donc que le facteur d'utilisation du système soit inférieur à 0.69. Dans ce cas, le système est ordonnançable par RM.

Des améliorations ont par la suite été apportées par Joseph et Pandya [23]. qui ont trouvé une condition nécessaire et suffisante. Ainsi, si $RT(t_i)$ est le temps de réponse (1.4.2) d'une tâche t_i, alors si un ensemble de tâches périodiques est trié par ordre décroissant, l'équation suivante définit la borne supérieure du temps de réalisation :

$$RT(t_i)^{q+1} = \sum_{j=1}^{i-1} \lceil \frac{RT(t_i)^q}{T(t_i)} \rceil \times C(t_j)) + C(t_i)$$

Le système est ordonnançable si et seulement si $\forall i_{(1 \leq i \leq m)} RT(t_i) \leq T_i$. On pourra résoudre récursivement cette équation afin de prouver sa faisabilité.

RM n'est cependant optimal que pour cette classe de tâches. Dès que les propriétés changent, il faudra se tourner vers un autre type d'ordonnanceur.

2.1.2 Deadline Monotonic

Deadline Monotonic (DM) est un ordonnanceur pour les systèmes à départ simultanés ($offset = 0$) et à échéance contrainte2($D_i \leq T_i$). Il a été décrit par Leung et Whitehead [27]. Cet article aborde le point de vue mono-processeur et multi-processeur partitionné, dont il sera question plus loin dans ce document.

Avec cet ordonnanceur, les priorités sont fixes, au niveau des tâches. Plus l'échéance est petite, plus la priorité est élevée. On peut considérer que *RM* est un cas particulier de *DM*, puisque pour *RM*, les tâches sont à échéance sur requête 1. Il n'existe pas de test de faisabilité basé sur l'utilisation du système. Pour vérifier celle-ci, il faut avoir recours à l'équation décrite plus haut. On peut la résoudre récursivement à l'aide de ce système d'équations :

1. $W_0 = C_i$
2. $W_{k+1} = C_i + \sum_{j=1}^{i-1} \lceil \frac{W_k}{T_j} \rceil \times C_j$

2.1.3 Earliest Deadline First

Earliest **D**eadline **F**irst (EDF) est un ordonnanceur qui a été introduit en 1973, à la même période que Rate Monotonic, également par Lui et Layland [30]. C'est un ordonnanceur à départ différé ($Offset \neq 0$) et à échéance arbitraire (pas de contrainte sur l'échéance par rapport à la période), qui fixe les priorités sur les jobs. L'assignation de la priorité se fait sur base de la proximité de l'échéance absolue : Plus cette échéance est proche et plus la priorité est élevée. Une façon déterministe et arbitraire de régler les cas d'égalité doit être décidée.

EDF est optimal pour toutes les tâches synchrones et asynchrones, avec et sans contraintes sur les échéances. Cela signifie que si un système est ordonnançable, *EDF* peut l'ordonnancer.
Cette propriété est importante, car les classes de tâches ordonnancées par *EDF* sont plus larges que *RM* et *DM*, par conséquent, *EDF* peut ordonnancer également les mêmes classes qu'*RM* et *DM* en conservant son optimalité.

Les tests de faisabilité avec *EDF* dépendent de la classe de tâche considérée. Pour un système synchrone et à échéance sur requête, une condition nécessaire et suffisante de faisabilité est donc que l'utilisation soit inférieure à 100%, c'est à dire :

$$\forall \tau_i \in \Gamma_m, \sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1$$

Dans le cas d'un système synchrone à échéance arbitraire, on doit considérer un intervalle de recherche $[0, L]$ avec L qui peut se calculer de façon itérative en cherchant un point

fixe avec cette formule :

$$\begin{cases} W_0 & = \sum_{i=1}^{m} C_i \\ W_{k+1} & = \sum_{i=1}^{m} \lceil \frac{W_k}{T_i} \rceil \times C_i \end{cases}$$

Dans le cas des systèmes de tâches asynchrones, l'intervalle considéré est plus grand : $[0, O_{max} + 2 \times P]$ avec O_{max} l'offset maximum du système et P le plus petit commun multiple (*ppcm*) de toutes les périodes du système, soit :

$$P = ppcm\{T_i | i \in \{1, ...m\}\}$$

Malgré le fait que la littérature montre la supériorité d'*EDF* sur *RM*, il semble que ce soit RM qui soit plus volontiers choisi pour implémentation. Sa réputation est qu'il est plus simple. On trouve un résumé du débat qui existe à ce sujet dans un article de Buzzato [11].

2.2 Multi-processeurs : les différentes familles d'ordonnanceurs

Dans la partie précédente, nous avons présenté certains algorithmes mono-processeur, ainsi que quelques conditions d'ordonnançabilité. De nos jours, une grande partie des architectures employées dans les systèmes embarqués est multiprocesseur. Les stratégies mises en place pour l'ordonnancement de tels systèmes sont différentes. Dans la littérature, il est habituel de présenter deux familles principales d'ordonnanceurs multiprocesseurs :

— Les ordonnanceurs partitionnés
— Les ordonnanceurs globaux

Nous verrons qu'une troisième famille est souvent présentée également. Elle représente un mélange des deux précédentes. On parle d'ordonnanceurs *hybrides*, ou *semi-globaux*.

2.3 Les ordonnanceurs partitionnés

L'idée principale derrière la stratégie du partitionnement est que pour tout ensemble Γ de n tâches, si l'utilisation de chaque tâche est inférieure à 1, alors il existe un ordonnanceur de $m <= n$ processeurs capable d'ordonnancer cet ensemble. Il y a principalement deux optiques différentes pour envisager ce problème :

— Partir de l'ensemble maximum et appliquer un algorithme de recherche afin de trouver la valeur de m la plus petite telle que l'ordonnancement soit possible
— Résoudre ce problème connu dans la littérature comme celui du Bin-Packing, et rejoindre un domaine bien connu et largement documenté.

Dans la suite du document, on ne s'intéressera qu'aux heuristiques liées au Bin-Packing et laisserons de côté les algorithmes de recherche.

2.3.1 Ordonnanceurs à priorité fixe sur tâche

La stratégie de partitionnement consiste à diviser l'ensemble de tâches en sous-ensembles qui seront attribués à un processeur particulier. Cela permet de conserver les mêmes

algorithmes ainsi que tests d'ordonnançabilité que ceux décrits précédemment puisqu'en divisant le système en sous-systèmes attribués à un processeur chacun, cela revient à appliquer "localement" une stratégie mono-processeur, contre une stratégie globale multi-processeur [34].
Concrètement, cela revient à diviser un ensemble Γ de tâches en n sous-ensembles $\gamma \in \Gamma$ et d'attribuer à chacun des m processeurs un de ces sous-ensembles γ.

Le problème du partitionnement consiste donc en premier lieu à résoudre une division du système en sous-systèmes, ce qui est connu dans la littérature scientifique comme le problème du bin-packing [14]. Ce problème est NP difficile, si bien qu'en pratique, des heuristiques sont appliquées, comme par exemple :

— First-fit
— Best-fit
— Next-fit

Plusieurs algorithmes de ce type ont été présentés dans la littérature scientifique dans les années 80 comme par exemple [18] dans ce document de Dhall et Liu en 1978. Dans leur article, ils donnent notamment des fourchettes d'utilisations permettant d'ordonnancer pour le cas multi-processeur utilisant *RM*, considérant les échéances implicites :

Théorème 1 *"if a set of m tasks is scheduled according to the rate-monotonic scheduling algorithm, then the minimum achievable utilization factor is* $m \times (2^{\frac{1}{m}} - 1)$ *".*

(Si un ensemble de m tâches est planifié selon l'algorithme *Rate-Monotonic*, alors le facteur d'utilisation minimum réalisable est $m \times (2^{\frac{1}{m}} - 1)$).
Ils ajoutent :
Note that according to theorem 1, as m approches infinity, the minimum achievable utilization approaches $ln(2)$ (En déduction, si m tend vers l'infini, alors l'utilisation minimale possible approche $ln(2)$.)

Dans cet article, ils décrivent également :

RMNFS : *Rate Monotonic Next-Fit Scheduler*
RMFFS : *Rate Monotonic First-Fit Scheduler*

qui finalement, sont des algorithmes basés sur *RM* dont le système de tâche est réparti selon les algorithmes heuristiques de *bin-packing Next-Fit* et *First-Fit*. Afin de trier les tâches, on considère leur utilisation.
Par la suite, d'autres algorithmes basés sur *RM* et *DM* avec heuristique de bin-packing sont proposés et leurs performances analysées. Un historique complet est proposé dans le travail de thèse de Ndoye [34]. Les recherches tendent à trouver des conditions de faisabilité utiles.

Ce pan des ordonnanceurs est donc très bien connu à ce jour, très bien documenté. Dans ces conditions, il n'est pas étonnant de voir que ces solutions sont encore largement répandues dans l'industrie à ce jour, les algorithmes *RM* et *DM* étant assez simples à implémenter, des heuristiques ainsi que leur efficacité ayant été analysées, en théorie ainsi qu'en pratique.

Un défaut majeur de ces algorithmes est que bien souvent, l'utilisation des processeurs sera sous-performante, ce qui est une raison très souvent invoquée dans la littérature pour se tourner vers d'autres solutions. Andersson, Baruah et Jonsson proposent RM-US$\frac{m}{3m-2}$ avec un test de faisabilité plus permissif [2]. Ils prouvent qu'un système est réalisable sous deux conditions :

— $\forall \tau_i \in \tau, U_i \leq \frac{m}{3m-2}$
— $U(\tau) \leq \frac{m^2}{3m-2}$

Pour arriver à ces conditions, l'algorithme d'attribution des priorités est légèrement modifié :

— si $U_i > \frac{m}{3m-2}$, τ_i a la plus grande priorité
— sinon, τ_i se voit assigner une priorité sous les mêmes conditions que *RM*.

2.3.2 EDF partitionné

Il a été rappelé plus tôt dans ce document qu'*EDF* mono-processeur est optimal, c'est à dire qu'il peut ordonnancer tout type de système qui est ordonnançable. Malheureusement, ce résultat n'est pas valable dans le cas multiprocesseur [17].

Cet algorithme a lui aussi été adapté à une utilisation multi-processeur en y joignant des heuristiques de bin-packing :
Dans [39], Zapata et Alvarez décrivent les algorithmes EDF avec partitionnement préalable, par exemple :

— EDF-FF (first-fit)
— EDF-NF (next-fit)
— EDF-WF (worst-fit)
— EDF-NFD (Earliest Deadline First Next Fit Decreasing)

et proposent une analyse de la complexité. Ils renvoient eux-mêmes vers [31] qui est cité de nombreuses fois et semble être la référence à propos de ces algorithmes. Dans cet article, Lopez, Diaz et Garcia proposent une limite de l'utilisation à la faisabilité en fonction de l'algorithme d'attribution des tâches. Ces conditions sont suffisantes, mais pas nécessaires.

2.3.3 Avantages et inconvénients des ordonnanceurs partitionnés

Avantages

Nous avons vu dans les sections précédentes que les algorithmes d'ordonnanceurs partitionnés sont bien documentés, et présentent un certain nombre d'avantages parmi lesquels le fait d'être bien connus aussi bien en pratique qu'en théorie. Certains algorithmes donnent même de bons résultats, et l'on pourrait se satisfaire de cette famille d'algorithmes. De nouvelles études et améliorations sont encore à ce jour proposées dans des recherches récentes [42].

Inconvénients

Toutefois, ces algorithmes présentent quelques inconvénients. L'un d'eux est que leur résolution ne garantit pas de trouver la solution optimale, puisque ce sont des heuristiques. Il n'est pas envisageable de penser que ces algorithmes puissent donc être optimaux pour une famille de tâches.
Un autre problème important est que dans le processus partitionné, une tâche est assignée à un seul processeur. Ainsi, si des solutions existent qui consistent à migrer une tâche sur un autre processeur, elles ne pourront pas être trouvées par un ordonnanceur partitionné [40]. Enfin, une autre limite est que ce processus ne permet pas à une tâche d'évoluer avec le temps. Le mode doit être décidé avant l'exécution, et devra être fixé une fois pour toutes. Cela peut convenir à un certain nombre de systèmes, mais ne peut pas être une solution générale. De façon générale, on retrouve dans la littérature l'idée que les ordonnanceurs globaux permettent généralement d'ordonnancer plus de systèmes que les ordonnanceurs partitionnés, mais la conclusion ne peut être considérée comme définitive : cela dépend des situations [31]. Il y a aussi un problème lié à la communication entre les tâches. Dans nos recherches, la plupart des articles posent que les tâches sont indépendantes entre elles. Dans la réalité, les tâches sont souvent dépendantes, elles doivent communiquer, ce qui pose des problèmes de précédence. L'approche partitionnée gère plusieurs ordonnanceurs indépendants entre eux, si bien que chaque système pourra gérer ces communications localement, mais pas les systèmes entre eux.

Il est connu que les algorithmes d'ordonnanceurs globaux et partitionnés ne sont pas comparables, pour la principale raison que les ordonnanceurs partitionnés ne permettent pas de faire des *migrations*. Dans l'article [7], *Baruah* compare les deux techniques pour des systèmes de tâches *sporadiques*, et ses recherches montrent plusieurs résultats importants :

Lemme 1 *There are task systems that are schedulable using global FJP algorithms that partitioned FJP algorithms cannot schedule.*

(il y a des systèmes de tâches ordonnançables avec des algorithmes globaux à priorité fixe sur job que des algorithmes partitionnés à priorité fixe sur job ne peuvent pas ordonnancer)

Lemme 2 *There are task systems that are schedulable using partitioned FJP algorithms that global FJP algorithms cannot schedule*

(il y a des systèmes de tâches ordonnançables avec des algorithmes partitionnés à priorité fixe sur job que des algorithmes globaux à priorité fixe sur job ne peuvent pas ordonnancer)
Ceci est prouvé en détaillant des contre-exemples dans l'article et permet de conclure :

Théorème 2 *Global and partitioned FJP scheduling are incomparable.*

Les ordonnanceurs globaux et partitionnés à priorités fixes sur job sont incomparables. Cela ne dit rien des autres classes, et ne peut pas être généralisé. C'est un résultat cependant important, qui montre que le fait de permettre les migrations change considérablement le comportement des ordonnanceurs.

2.4 Ordonnanceurs multi-processeurs globaux

Un ordonnanceur global est un ordonnanceur qui gère un ensemble de processeurs et un système de tâches. Il ne procède pas à un tri préalable comme dans le cas partitionné. Il gère une *queue* de n tâches prêtes à être exécutées qui peuvent être assignées à m processeurs à chaque fois que l'ordonnanceur en a l'occasion.

Les algorithmes mono-processeurs vus précédemment peuvent donc être adaptés à ce type de stratégie facilement : en attribuant les priorités à toutes les tâches, et en exécutant plusieurs jobs sur plusieurs processeurs. Plusieurs résultats qui viennent assombrir les perspectives d'une adaptation aussi simple.

2.4.1 L'Effet Dhall

Dans leur article de 1978 [18], Dhall et Liu montrent que certains ordonnanceurs initialement mono-processeurs, (*RM* ou *EDF*) peuvent donner lieu à une utilisation faible des processeurs, ce qui signifie qu'ils ne seraient pas utilisés de façon optimale. Ils montrent par ailleurs que pour certains algorithmes, il existe des systèmes qui présentent des *pathologies*. Cela signifie que certains systèmes ne sont pas ordonnançables malgré une utilisation totale de $1 + \epsilon$ et ce quel que soit le nombre de processeurs m. La déduction suivante peut en être tirée : la limite de l'utilisation totale pour RM ou EDF est de $1+\epsilon$, avec ϵ arbitrairement petit, ce quel que soit m.
Cela est connu dans la littérature comme l'"effet Dhall" (Dhall's effect) et remet en question l'intérêt de baser les tests de faisabilité sur l'utilisation totale, et montre que d'autres facteurs devraient être pris en considération. Dans un premier temps, cependant, cela a participé à montrer la supériorité des approches partitionnées, ce pourquoi elles ont largement été plus étudiées dans les années 80-90 [16].

De plus, dans le cas global, les instants critiques ne sont plus aussi faciles à prédire que dans le cas mono-processeur : pour rappel, un instant critique se produit lorsqu'une tâche libère un job à un moment où tous jobs de priorités supérieures doivent être exécutés. Mais cela n'est plus forcément vrai dans le cas multi-processeur.

2.4.2 Anomalies

Les anomalies sont décrites dans la littérature comme des changements qui intuitivement devraient améliorer l'utilisation des processeurs, et qui rendent cependant le système non-ordonnançable. Cela peut se produire dans diverses circonstances, comme par exemple :

- le temps d'exécution décroît
- La période d'une tâche augmente

et le système qui était auparavant ordonnançable ne l'est plus.

2.4.3 Priorité fixe sur tâche : RM/DM global

Si l'assignation des priorités *RM* (ou *DM*) en version multi-processeur global fonctionne de la même façon que dans sa version mono-processeur - c'est à dire, la priorité la plus élevée est accordée à la tâche qui a la période la plus faible - *RM* est donc concerné par

l'effet Dhall.

2.4.4 Priorité fixe sur job : EDF

Comme pour *RM* et *DM*, *EDF* peut simplement être adapté à une exécution multi-processeur, considérant un ensemble n de tâches sur m processeurs. Mais le gros avantage d'*EDF* dans sa version mono-processeur est son optimalité, qui n'est malheureusement pas conservée ici. Plusieurs versions globales reposant sur *EDF* sont décrites dans la littérature. Il y a *EDF* appliquant *First Fit Decreasing Utilization*, décrit par Lopez et al. [31], où l'on obtient ces conditions de faisabilité :
$U(\tau) \leq \frac{(m+1)}{2}$ et $U_{max} \leq 1$.
Un autre exemple est EDF-US. Dans ce cas, la priorité des tâches est assignée légèrement différemment :

— si $U_i > \frac{m}{2m-1}$, la tâche τ_i se voit assigner la plus haute priorité
— sinon on applique les mêmes règles que pour *EDF* mono-processeur

Une condition suffisante est alors :
$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \frac{m^2}{3m-2}$ alors le système de tâches est ordonnançable sur m processeurs [2].
D'autres documents encore décrivent des conditions d'ordonnançabilité et sont régulièrement cités comme références comme de nombreux articles de Baker [4] [5] ou Baruah [9] [8]. Dans cet article de Davis et Burns, les auteurs rappellent les limites supérieures d'utilisation pour ces techniques [16].

2.4.5 Il n'existe pas de stratégie en ligne optimale sans clairvoyance

Quelques résultats déjà connus dans la littérature montrent que le sujet est difficile. En effet, dans leur article de 1988 [22], Hong et Leung montrent :
"for every $m > 1$, no optimal online scheduler can exist for task systems with two or more distinct deadlines." (Pour tout $m > 1$, il n'existe pas d'ordonnanceur en ligne optimal pour les ensembles de tâches avec au moins deux échéances distinctes). Ce résultat, quoi que négatif, ne peut être généralisé à toutes les classes de tâches, il s'applique aux tâches sporadiques.
C'est un résultat embarrassant, car beaucoup de systèmes embarqués gèrent des tâches sporadiques : en effet, bon nombre d'appareils sont des détecteurs, qui attendent un signal qui par définition ne peut pas arriver à un moment déterministe. Toutefois, de nombreuses publications ultérieures ont montré des algorithmes optimaux pour d'autres classes de tâches. Par ailleurs, ce résultat est juste dans le cas d'un ordonnanceur non-clairvoyant, mais ne peut pas être généralisé dans le cas où l'algorithme a accès à des données sur les tâches au préalable [19].

2.4.6 PFair

En 1996, Baruah et al. décrivent l'algorithme *PFair*. L'article apporte plusieurs définitions importantes :

Définition 1 *$Lag(S, x, t) = x.w \times t - \sum_{i\in[0,t)}^{max} S(x, i)$, où S est un ordonnanceur, x, un job, t un instant, et où $S(x, t) = 1$ si le job x est exécuté à l'instant t, 0 sinon.*

Définition 2 *Un ordonnanceur est P-fair si et seulement si :*
$\forall x, t : x \in \Gamma, t \in, \mathbb{N} : -1 < lag(S, x, t) < 1$

Définition 3 *Un ordonnanceur est P-fair au temps t si et seulement si un ordonnanceur S' existe tel que :*
$\forall x : x \in \Gamma lag(S, x, t) = lag(S', x, t)$

Théorème 3 *La P-équité (PFairness) implique que toutes les échéances soient satisfaites.*

PFair est associé dans la littérature à l'idée de *P-Fairness*, qui est une notion idéale permettant de prouver des propriétés importantes pour le domaine. L'une d'elle est que chaque ordonnanceur P-Fair est périodique.

Il s'applique aux tâches périodiques synchrones à échéances implicites, et l'on connaît le temps de réalisation des tâches, ce qui prévient des résultats négatifs précédents contre l'existence d'un ordonnanceur optimal.

L'idée principale de cet algorithme est basée sur le taux d'utilisation de chaque tâche. L'algorithme divise chaque job en sous-jobs qui devra s'exécuter dans une fenêtre de temps, considérée comme sous-échéance. Dès lors, à chaque sous-échéance, chaque job aura reçu la proportion de temps nécessaire. Cet algorithme présente des intérêts mais aussi un inconvénient : dans certains cas, l'ordonnancement provoque de nombreuses préemptions, ce qui est coûteux en terme de ressources.

Toutefois, *PFair* est optimal pour les tâches synchrones à échéances implicites, selon la condition suivante :

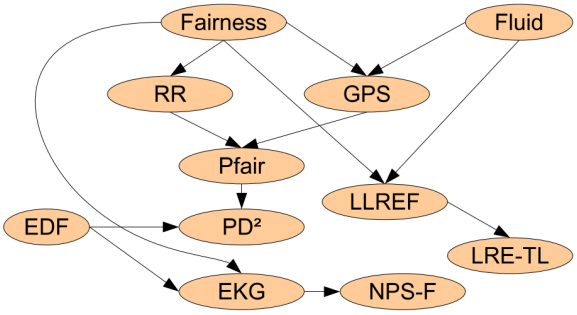
Définition 4 *Let τ be a periodic synchronous implicit-deadline system. A PFAIR schedule exists for τ on m processors if and only if :*
$U(\tau) \leq m$, *and* $U_{max} \leq 1$ *[6].*

(Prenons un système synchrone périodique à échéances implicites. Un ordonnanceur PFair existe pour τ sur m processeurs si et seulement si :
$U(\tau) \leq m$, et $U_{max} \leq 1$)

La notion de *P-Fairness* est très souvent utilisée dans la littérature et d'elle dérivent de nombreuses propositions d'algorithmes, comme les classiques *PF* [6], *PD* [10], *PD²* [43] , *ER* [1], et d'autres moins "classiques" comme *DP-Fair* [28], *LB-Pfair* (Loop-Back Proportionate Fair) [26]. Müller et Werner proposent ainsi ce schéma dans leur article de 2011 :
Ils précisent qu'*EDF* n'est pas concerné par cette classification, mais qu'ayant inspiré certaines approches, il méritait de se trouver sur cette image. Nous invitons le lecteur intéressé par les influences des algorithmes entre eux à prendre connaissance de cet article extrêmement documenté.

De la lecture des articles, il ressort qu'une partie non négligeable d'entre eux est de parution récente, ce qui montre l'intérêt scientifique actuel pour ce type d'algorithmes. Bien évidemment, les propositions diffèrent par certaines propriétés. Par exemple, *LB-Pfair* est orienté vers les systèmes critiques tolérants aux erreurs. Un enjeu très important

FIGURE 2.1 – "Genealogy of fully dynamic scheduling algorithms with full migration.", issu de l'article de Müller et Werner[33]



de ce grand nombre d'algorithmes est de trouver une méthode qui garantisse l'optimalité pour la classe sporadique, tout en conservant de bonnes performances.
La préoccupation principale actuelle est de trouver un moyen d'améliorer l'utilisation des processeurs (limitée à 50% pour les algorithmes partitionnés [37]), sans détériorer les performances par une explosion du nombre de migrations, comme c'est généralement le cas avec des algorithmes globaux. Dans l'article d'Andersson de 2006 [3], la limite d'utilisation n'est pas aussi importante qu'avec une approche globale, mais le nombre de préemptions est limité par une constante k. En 2006 toujours, Cho et al. [12] proposent *LLREF* (largest local remaining execution time), dont l'optimalité est prouvée pour les systèmes périodiques. Ses performances ne sont à ce jour pas bien connues en pratique. La liste proposée ici n'est pas exhaustive, un très grand nombre de propositions existe actuellement.

2.4.7 EDF-k

En 2003, Goossens et al. proposent l'algorithme *EDF-k* [20], dont l'idée principale est d'être basé sur la priorité. La définition de *Priority-driven Algorithm* (axé sur les priorités) est donnée par Ha et Liu en 1994 :

Définition 5 *A Scheduling algorithm is said to be a priority driven scheduling algorithm if and only if it satisfies the condition that for every pair of jobs J_i and J_j, if J_i has a higher priority than J_j at some instant in time, then J_i always has higher priority than J_j.*

(Un algorithme d'ordonnancement est considéré comme axé sur les priorités si et seulement si il satisfait la condition suivante que pour chaque paire J_i et J_j, si J_i a une plus grande priorité que J_j à un instant, J_i a alors toujours une plus grande priorité que J_j.)
Suivant cette idée, *EDF* est axé sur les priorités, là où *PF* ne l'est pas.

Reprenant l'idée précédente de *EDF-US* dont certaines tâches étaient de priorité supérieures, et d'autres de priorité simplement assignées par *EDF* "classique", le nombre k représente le nombre stable de tâches (-1) concernées par ces priorités supérieures. En d'autres termes, *EDF-k* donne la priorité la plus élevée à $k-1$ tâches, tandis que les autres sont ordonnancées suivant *EDF*.

L'article propose une équation dont on dérive la valeur optimale de k, et où l'on peut atteindre m le plus petit, cette valeur étant améliorée par rapport à *EDF-US*.

Une autre approche consiste à considérer la laxité pour modifier *EDF*. Ainsi, l'idée d'*EDZL* (Earliest Deadline until Zero Laxity) [13] ou encore d'EDCL [24] (Earliest Deadline Critical Laxity).

Des travaux récents continuent d'implémenter des versions d'*EDF* avec stratégie globale, par exemple *GEDF* [29], donnant lieu à *PGEDF*. Cette branche continue donc d'être étudiée et présente des intérêts.

2.4.8 U-EDF

U-EDF est présenté en 2011 par Nelissen et al. [35]. Il n'est pas "*P-Fair*", et se démarque donc d'une bonne partie des algorithmes par le fait qu'il ne cherche pas à vérifier de condition de "P-équité". Il prend en charge les systèmes périodiques à échéances implicites, et est optimal pour cette classe. Tout d'abord, la preuve de son optimalité se limite aux tâches périodiques dont on a dit plus haut qu'elles ne représentaient pas la majorité des classes e tâches des systèmes embarqués. En 2012, la preuve de son optimalité est élargie aux systèmes sporadiques par Nelissen et al. [36]. Le principal but de cet algorithme est de réduire le nombre de préemptions, ce qui est un grand inconvénient de l'approche globale.
L'idée principale d'*U-EDF* est

2.4.9 RUN

Reduction to UNiprocessor (*RUN*) est un algorithme présenté par Regnier et al. en 2013 [41]. Cet algorithme est appliqué aux systèmes périodiques préemptifs à tâches indépendantes à échéances implicites. *RUN* – contrairement aux exemples vus précédemment – n'applique pas la *P-Fairness*, et parvient à réduire significativement le nombre de préemptions. *RUN* réduit un ensemble de tâches en plus petits ensembles plus facilement ordonnançables en suivant deux opérations :

— Une opération "*Dual*"
— Une opération *Pack*

Un *Dual* se construit par complémentarité avec un *Primal*, dont les règles de constructions sont données dans l'article. Un serveur est chargé d'ordonnancer chaque sous-système. Celui-ci fonctionne à l'aide d'*EDF*, dont les avantages nombreux ont déjà été évoqués plus tôt dans ce document. Il n'est pas évident à la lecture de l'article de comprendre en quoi diffère l'approche de *RUN* par rapport à un ordonnanceur partitionné qui donnerait lieu à des systèmes ordonnancés à l'aide d'*EDF*. En réalité, *RUN* n'est pas un algorithme global, mais plutôt semi-partitionné, et la différence tient particulièrement du fait que les systèmes ne sont pas gérés par des processeurs distincts mais par des serveurs. Les auteurs insistent sur les avantages théoriques de RUN, qui devraient motiver une implémentation pratique afin de tester ses avantages. Cette implémentation a été faite [15] sur $LITMUS^{RT}$, et dont les résultats demandent à être confirmés mais montrent les bonnes performances sur ce système.

RUN – malgré son apparition récente – a déjà des successeurs. *QPS* est un algorithme

également semi-partitionné, mais à la différence de RUN, il peut ordonnancer les systèmes de tâches indépendantes sporadiques à échéances implicites. Il est décrit dans un article de Massa et al. [32] en 2014. Comme pour *RUN*, *QPS* génère des sous-ensembles de tâches qui seront ordonnancés selon *EDF* par des serveurs. Les auteurs présentent l'avantage d'une approche semi-partitionnée, qui fait un compromis entre les avantages de l'approche partitionnée et ceux de l'approche globale mais *RUN* comporte quant à lui l'inconvénient de ne pas pouvoir s'appliquer aux tâches sporadiques. C'est ce qui explique la nécessité de l'existence de *QPS*.

2.4.10 Tableau comparatif des algorithmes globaux et semi-globaux

Afin de faire le choix de l'algorithme à implémenter, une comparaison de ces algorithmes devrait être faite afin d'orienter le choix vers un ordonnanceur plutôt qu'un autre.

Nom	**Classe**	**Optimal**	**Migrations**
EDF-k	Sporadique, échéance implicite	Optimal (P-Fair)	Nombreuses
GEDF	Sporadique, échéance implicite	Non optimal	Nombreuses
U-EDF	Sporadique, échéance implicite	Optimal	Réduites
DP-WRAP	Sporadique, échéances arbitraires	Optimal (P-Fair)	Réduites
LLREF	Périodique, échéances arbitraires	Optimal	Réduites
BF	Périodiques, échéances implicites	Optimal	Réduites
RUN	Périodiques, échéances implicites	Optimal	Un peu réduites
QPS	Sporadiques, échéances arbitraires	Optimal	Réduites

2.5 Conclusion

Nous avons exploré une petite partie de la littérature scientifique au sujet des ordonnanceurs globaux ou semi-partitionnés afin d'en sélectionner un pour implémentation et tests. Il ressort de cette étude que plusieurs candidats présentent des intérêts, sont mieux connus dans la littérature que dans la pratique, et gagneraient à être mieux analysés. Notre choix sera influencé par ces paramètres :

— L'ordonnanceur est-il optimal
— Quels systèmes de tâche peut-il ordonnancer ?
— Des efforts ont-ils été fournis afin d'éviter les migrations ?
— A-t-il déjà bénéficié d'implémentations ?

De ces paramètres, il ressort que l'algorithme *QPS* est optimal, semi-partitionné, peut ordonnancer les systèmes sporadiques à échéances arbitraires et les articles à son sujet parlent de bonnes performances concernant les migrations.
C'est donc vers cet algorithme que notre choix se porte.

Bibliographie

[1] J.H. Anderson and A. Srinivasan. Early-release fair scheduling. pages 35–43. IEEE Comput. Soc, 2000.

[2] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. pages 193–202. IEEE Comput. Soc, 2001.

[3] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. pages 322–334. IEEE, 2006.

[4] T.P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. pages 120–129. IEEE Comput. Soc, 2003.

[5] T.P. Baker. An analysis of EDF schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8) :760–768, August 2005.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, June 1996.

[7] Sanjoy Baruah. Techniques for Multiprocessor Global Schedulability Analysis. pages 119–128. IEEE, December 2007.

[8] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3) :223–235, April 2008.

[9] S.K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6) :781–784, June 2004.

[10] S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. pages 280–288. IEEE Comput. Soc. Press, 1995.

[11] Giorgio C. Buttazzo. Rate Monotonic vs. EDF : Judgment Day. *Real-Time Systems*, 29(1) :5–26, January 2005.

[12] Hyeonjoong Cho, Binoy Ravindran, and E. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. pages 101–110. IEEE, 2006.

[13] Michele Cirinei and Theodore P. Baker. EDZL Scheduling Analysis. pages 9–18. IEEE, July 2007.

[14] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin-Packing — An Updated Survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, volume 284, pages 49–106. Springer Vienna, Vienna, 1984. DOI : 10.1007/978-3-7091-4338-4_3.

[15] Davide Compagnin, Enrico Mezzetti, and Tullio Vardanega. Putting RUN into Practice : Implementation and Evaluation. pages 75–84. IEEE, July 2014.

[16] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4) :1–44, October 2011.

[17] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, December 1989.

[18] Sudarshan K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1) :127–140, February 1978.

[19] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2) :26–71, June 2010.

[20] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, 25(2) :187–205, September 2003.

[21] Sangchul Han and Minkyu Park. Predictability of Least Laxity First Scheduling Algorithm on Multiprocessor Real-Time Systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Xiaobo Zhou, Oleg Sokolsky, Lu Yan, Eun-Sun Jung, Zili Shao, Yi Mu, Dong Chun Lee, Dae Young Kim, Young-Sik Jeong, and Cheng-Zhong Xu, editors, *Emerging Directions in Embedded and Ubiquitous Computing*, volume 4097, pages 755–764. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. DOI : 10.1007/11807964_76.

[22] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. pages 244–250. IEEE Comput. Soc. Press, 1988.

[23] M. Joseph. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5) :390–395, May 1986.

[24] Shinpei Kato and Nobuyuki Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. pages 441–450. IEEE, August 2007.

[25] Omar Kermia. *Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence, de périodicité stricte et de latence*. Thesis, Université Paris XI, UFR scientifique d'Orsay, 2009.

[26] Stefan Kramer, Jurgen Mottok, and Stanislav Racek. Proportionate fair based multicore scheduling for fault tolerant multicore real-time systems. pages 088–093. IEEE, March 2015.

[27] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4) :237–250, December 1982.

[28] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. DP-FAIR : A Simple Model for Understanding Optimal Multiprocessor Scheduling. pages 3–13. IEEE, July 2010.

[29] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global EDF scheduling for parallel real-time tasks. *Real-Time Systems*, 51(4) :395–439, July 2015.

[30] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1) :46–61, January 1973.

[31] J. M. López, J. L. Díaz, and D. F. García. Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28(1) :39–68, October 2004.

[32] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. OUTSTANDING PAPER : Optimal and Adaptive Multiprocessor Real-Time Scheduling : The Quasi-Partitioning Approach. pages 291–300. IEEE, July 2014.

[33] Dirk Müller and Matthias Werner. Genealogy of hard real-time preemptive scheduling algorithms for identical multiprocessors. *Open Computer Science*, 1(3), January 2011.

[34] Falou Ndoye. *Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation.* Thesis, UNIVERSITÉ PARIS-SUD, Sciences et Technologie de l'Information, des Télécommunications et des Systèmes INRIA, March 2014.

[35] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Reducing Preemptions and Migrations in Real-Time Multiprocessor Scheduling Algorithms by Releasing the Fairness. pages 15–24. IEEE, August 2011.

[36] Geoffrey Nelissen, Vandy Berten, Vincent Nelis, Joel Goossens, and Dragomir Milojevic. U-EDF : An Unfair But Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks. pages 13–23. IEEE, July 2012.

[37] Dong-Ik Oh and T.P. Bakker. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, 15 :183–192, 1998.

[38] Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, and Ben Rodriguez. A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms.

[39] Omar Pereira Zapata and Pedro Mejia Alvarez. EDF and RM Multiprocessor Scheduling Algorithms : Survey and Performance Evaluation. 2005.

[40] S. Ramamurthy and M. Moir. Static-priority periodic scheduling on multiprocessors. pages 69–78. IEEE, 2000.

[41] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott Brandt. Multiprocessor scheduling by reduction to uniprocessor : an original optimal approach. *Real-Time Systems*, 49(4) :436–474, July 2013.

[42] Paul Rodriguez, Laurent George, Yasmina Abdeddaïm, and Joël Goossens. Multi-Criteria Evaluation of Partitioned EDF-VD for Mixed-Criticality Systems Upon Identical Processors, 2013.

[43] Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6) :1094–1117, September 2006.