UNIVERSITÉ LIBRE DE BRUXELLES

Faculté des Sciences

Département d'Informatique

MEMO-F-524

Master thesis

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

Arabella Brayer

Arabella.Brayer@ulb.ac.be

Promoteur : Pr. **Joël Goossens**

Expert externe : **Paul Rodriguez**

Mémoire présenté en vue de l'obtention du diplôme de Master en Sciences informatiques

Année académique 2018 – 2019

Les sources de ce document se trouvent sur ce dépôt

https://github.com/subsib/Scheduling

Table des matières

Abstract

Le problème de l'ordonnancement des tâches dans un système d'exploitation est central : il a un impact majeur dans la gestion des ressources. En effet, même si les applications sont sobres individuellement, dans le cas où l'ordonnanceur n'offre pas une utilisation optimisée des ressources, alors il en résultera un gaspillage tant d'énergie que d'infrastructure. Malgré l'existence d'ordonnanceurs multi-processeurs globaux optimaux décrits dans la littérature scientifique, l'industrie continue à ce jour à préférer des solutions mono-processeurs voire multi-processeurs partitionnées plus simples, éprouvées mais moins efficaces. Ainsi, les conditions, les contraintes, et les effets de la mise en œuvre pratique de tels ordonnanceurs restent mal connus.
Le présent document expose le choix d'un ordonnanceur : **UEDF**, son implémentation dans un RTOS : **HIPPEROS** ainsi qu'une évaluation de ses performances, en le comparant à un autre algorithme connu : Global-EDF, avant de proposer des améliorations.

Nous prouvons avec ce travail qu'**UEDF** est implémentable, présente des caractéristiques intéressantes mais nécessite de l'optimisation avant de pouvoir éventuellement montrer sa supériorité sur **Global-EDF**.

Remerciements

Je voudrais d'abord adresser un grand merci au directeur de ce mémoire, Joël Goossens, pour son expertise reconnue, ses très bons conseils, et sa supervision bienveillante.

Je tiens également à adresser de chaleureux remerciements à Paul Rodriguez pour sa grande patience, sa compréhension même lorsque les phrases sont incomplètes et mal formulées et ses grandes compétences dans le domaine de ce travail.

Je remercie **HIPPEROS** et son équipe, de m'avoir donné la chance d'apprendre de précieuses compétences en m'accueillant en stage dans leur entreprise.

Je remercie chaleureusement l'**ULB** et son équipe pédagogique, particulièrement Olivier Markovitch, de m'avoir aiguillée vers la voie du Master et d'avoir rendu cette reprise d'études possible.

Je ne peux remercier suffisamment mon épouse pour son soutien inconditionnel, tant logistique que psychologique, sans qui je n'aurais jamais pu faire ce travail. Je remercie également ma mère, de son aide précieuse et régulière.

Introduction

0.1 Introduction générale

Le monde du système embarqué est actuellement en plein foisonnement. Tous ces systèmes n'ont cependant pas les mêmes contraintes. Certains doivent respecter des échéances afin de garantir mathématiquement un fonctionnement conforme. Typiquement, on peut tolérer un retard d'affichage de la page de son navigateur, il n'est cependant pas concevable qu'un ABS de voiture soit en retard à cause d'une autre tâche qui prendrait du temps, mettant en péril la sécurité des passagers. Une proportion des systèmes embarqués est dite "en temps réel" et nécessite des ordonnanceurs adaptés afin de prouver que les échéances seront toujours respectées. Ce champ de recherche – temps réel – a été largement nourri durant les vingt dernières années par de nombreuses publications scientifiques.

Toutefois, il demeure un décalage important entre la connaissance scientifique et la mise en pratique de celle-ci. Par exemple, les systèmes embarqués disposent bien souvent 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 [39]. 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].

Si les systèmes embarqués n'ont pas toujours de grands besoins en efficacité, un système dont l'efficacité n'est pas maximale n'utilise pas ses ressources de façon optimale, ce qui conduit à du gaspillage. Or, la gestion énergétique est un poste coûteux qui se doit d'être le plus réduit possible. En l'occurrence, pour des instal-

lations comprenant beaucoup de systèmes embarqués, comme les voitures ou les avions, cette dépense peut être substantielle.

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, la consommation en énergie des solutions déployées n'est pas optimale. 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 à recharger.
4. Sur de grosses installations, comme des voitures ou un avion, cela peut représenter un coût important en terme d'occupation de l'espace, de câblage entre différents systèmes, consommation d'énergie, poids, achat de composants.

Toutes ces raisons poussent à s'intéresser à une implémentation réelle et réaliste d'ordonnanceurs décrits dans la littérature, mais pas ou peu implémentés dans la réalité.

L'objectif de ce travail est de réaliser l'implémentation d'un ordonnanceur global en ligne dans un **RTOS**, afin d'en extraire des observations de différents niveaux : sur l'ordonnanceur choisi lui-même en premier lieu, afin de vérifier qu'il est applicable en pratique. Puis,de façon plus générale, de proposer des changements dans le travail des théoriciens afin de faciliter le passage de la description à l'implémentation, afin d'en multiplier le nombre, et pourquoi pas, d'améliorer la connaissance globale de ce type d'ordonnanceurs à l'avenir.

Dans ce travail, on se propose d'analyser plusieurs éléments, outre d'exposer les choix d'implémentation et d'en expliquer les raisons :

- Comparer les performances attendues théoriquement et des résultats mesurés en pratique
- Comparer pour des mêmes ensembles de tâches les performances d'UEDF avec Global-EDF, tous les deux implémentés dans **HIPPEROS**
- Proposer des éléments afin de faciliter le passage de la littérature à la pratique, pour permettre ultérieurement à des chercheurs d'y travailler et rendre ces implémentations faisables, voire moins fastidieuses.

Chapitre 1

Vocabulaire général sur les systèmes temps réel

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

in : « *Computing machinery and intelligence* »

1.0.1 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.
- **Temps de réponse** : Temps maximum nécessaire à la réalisation de la tâche entre le temps de réalisation et Fin.
- **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
  Le WCET diffère donc du temps d'exécution réel de la tâche. En pratique, il peut être plus ou moins éloigné du temps moyen d'exécution.
- **Tâches périodiques** : Une tâche périodique est une tâche qui génère régulièrement des travaux. 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 travail.
  - T : est la période, c'est-à-dire le temps qui sépare deux générations de travaux par la tâche. Le premier travail est généré à l'instant O. Soit i, le nombre de fois que la tâche a été générée depuis le temps 0, alors pour $i \in \mathbb{N}$, $\forall i \in \{0, \infty\}$, $t = O + i \times T$ 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 travail 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.

- **Laxité** : La laxité d'une tâche est la durée entre sa réalisation et son échéance. Pour une tâche τ_i, soit D_i son échéance et C_i son temps d'exécution, la laxité est définie comme :

$$L_i = D_i - C_i$$

- **Tâche sporadique** : Une tâche sporadique est une tâche qui génère de nouveaux travaux, comme dans le cas de la tâche périodique. La différence entre ces deux types est que la tâche sporadique génère deux travaux 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 travaux. Les propriétés du travail ne sont connues que lorsqu'un travail est généré dans le système.

1.0.2 Migration

Une migration est le transfert d'une tâche ou d'un travail d'un processeur vers un autre. Habituellement, les modèles négligent les temps de surcoût liés aux migrations. Néanmoins, une migration entraîne un déplacement de contexte, ce qui occupe les ressources.

1.0.3 Système de tâches

Un système de tâches et un ensemble de tâches devant être exécutées conjointement.

1.0.4 Faisabilité

Un système de tâches est dit faisable si un ordonnancement existe.

1.0.5 Synchrone/asynchrone

Une tâche est définie par son offset, et lorsque toutes les tâches ont un *offset* égal à 0, elles sont dites « à départ simultané », et on parle de système synchrone. Lorsque les *offset* ne sont pas tous de 0, le système est dit à départs « différés », on parle également de système asynchrone.

1.0.6 Hyperpériode

L'hyperpériode est un intervalle de temps qui est défini de façon formelle comme suit :$P = ppcm\{T_i\} \in \{1, ...m\}\}$

Cette notion est souvent utilisée afin d'obtenir l'intervalle de temps minimal durant lequel tester l'ordonnançabilité d'un système par un ordonnanceur.

1.0.7 Périodes harmoniques

Nous définissons un ensemble comme étant composé de tâches de périodes harmoniques si pour deux tâches prises au hasard, la plus petite période des deux multipliée par un entier est égale à la plus grande.
$\forall i \in TaskSet, \forall j \in TaskSet, d_i(t) < d_j(t), \exists n \in \mathbb{N}\ t.q.\ d_i(t) \times n = d_j(t)$

1.0.8 Travail

Un travail est une instance d'une tâche. Il peut avoir des propriétés. Dans certains algorithmes, certaines propriétés sont plutôt liées à la tâche, et dans d'autres, ces mêmes propriétés sont plutôt dynamiques, et donc liées au travail.

- **Début, Fin** : Le moment où un travail commence à être exécuté, ou termine son exécution.

- **Échéance, temps limite** : Correspond au temps limite au delà duquel le travail doit avoir été exécuté. Il existe deux types d'échéances : une absolue, et une relative. L'échéance relative est une valeur fixe qui est une propriété de la tâche, elle dépend donc du temps de réalisation. L'échéance absolue quant à elle est calculée avant l'exécution, et correspond à un temps t absolu, elle est donc une propriété du travail.

1.0.9 RTOS

(**R**eal-**T**ime **O**perating **S**ystem) Système d'Exploitation Temps Réel.
Un système d'exploitation Temps Réel est 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 fiabilité 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'exploitation des ordinateurs de bureau puisque l'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. En effet, dans un système dit « critique » , une défaillance du système pourrait entraîner des dégâts très importants.
Dans ce cadre, l'on a besoin de garanties que le système soit fiable, et c'est là une priorité absolue. L'efficacité en elle-même n'est pas le besoin le plus fondamental de ce type de système.

1.0.10 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.

Dans la littérature scientifique, on parle de **hard real-time** dans le cas d'un ordonnanceur qui ne tolère aucun dépassement, et de **soft real-time** dans le cas contraire.

1.0.11 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 travaux : La priorité est fixée par l'ordonnanceur à l'arrivée du travail dans l'exécution, et elle ne peut pas changer jusqu'à la réalisation du travail. 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 travail avec un exemple connu, Least Laxity First.

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.

1.0.12 Optimalité

Un ordonnanceur est dit optimal si pour un système de tâches pour lequel un ordonnancement existe, l'ordonnanceur permet de l'ordonnancer. Autrement formulé, si un ordonnanceur optimal ne peut pas trouver d'ordonnancement pour un système, aucun autre ordonnanceur ne le peut.

1.0.13 Économe

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.

1.0.14 En ligne, hors ligne

Un ordonnanceur peut faire ses opérations d'ordonnancement de deux façons :

- Durant l'exécution, on parlera d'opérations en ligne
- Avant l'exécution, on parlera d'opérations hors ligne

En pratique, les ordonnanceurs hors ligne sont généralement plus simples et génèrent moins de surcoût 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 [1.0.2] sont possibles. Le problème est donc de réduire leur nombre pour ne pas faire de mouvements inutilement.

1.0.15 Clairvoyance

Un ordonnanceur est dit « clairvoyant » s'il peut prédire des éléments (le début d'un travail) des tâches du système qu'il ordonnance.

1.0.16 Préemption

Un système est dit préemptif s'il a la capacité de mettre l'exécution d'un travail en pause et d'exécuter un autre à la place.

1.0.17 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 $WCET = 30ms$ et $T = 50ms$, 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. En pratique, l'utilisation est calculée avec le WCET, qui est la borne supérieure du temps d'exécution réel, on peut donc penser que l'utilisation réelle sera inférieure à ce nombre. 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 [29] en donnent une définition formelle.

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 :

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

1.0.18 Laxité

La laxité d'un travail est définie par la différence entre le temps de réponse et l'échéance d'une tâche. Sa définition formelle varie en fonction de l'ordonnanceur dont il est question.

1.0.19 Instant critique

Un instant critique est le moment où le temps de réponse d'une tâche est le plus long.

Chapitre 2

État de l'art

Computer Science is embarrassed by the computer.

Alan Perlis

Epigrams on Programming

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 multi-processeurs Globaux

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

2.1 Ordonnanceurs mono-processeur

2.1.1 Ordonnanceurs à priorité fixe sur tâche

Rate Monotonic

L'ordonnanceur **R**ate **M**onotonic (RM) est décrit par Liu et Layland [29] en 1973. C'est donc un ordonnanceur classique, bien connu, ainsi que largement documenté [24]. 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 d'ordonnançabilité 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 $m \to \infty$, $\sum_{i=1}^{m} \frac{C_i}{T_i} \leq m(2^{\frac{1}{m}} - 1) \to \ln(2) \approx 0.69$. 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 [22], qui ont trouvé une condition nécessaire et suffisante. Ainsi, si $RT(t_i)$ est le temps de réponse (1.0.1) d'une tâche t_i, alors si un ensemble de tâches périodiques est trié

par ordre décroissant, l'équation récurrente suivante définit la borne supérieure du temps de réalisation, lorsqu'elle tend vers un point fixe :

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

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

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.

Deadline Monotonic

Deadline Monotonic (DM) est un ordonnanceur optimal pour la classe de systèmes à départ simultanés et à échéances contraintes. Il a été décrit par Leung et Whitehead [26]. 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 d'ordonnançabilité basé sur l'utilisation du système. Pour vérifier celle-ci, il faut avoir recours à l'équation décrite plus haut. On peut la résoudre itérativement à l'aide de ce système d'équations :

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

2.1.2 Ordonnanceurs à priorité fixe sur travail

2.1.3 Earliest Deadline First

Earliest **D**eadline **F**irst (EDF) est un ordonnanceur qui a été introduit en 1973, dans le même article que celui où est présenté Rate Monotonic par Lui et Layland [29]. C'est un ordonnanceur capable d'ordonnancer aussi bien les tâches à départs simultanés que celles à départs différés, et les tâches à échéances contraintes ainsi qu'arbitraires (pas de contrainte sur l'échéance par rapport à la période). Il fixe les priorités sur les travaux. 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, l'intervalle de temps qu'il est nécessaire de vérifier est $[0, L]$ 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 multi-processeur. 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 multi-processeurs :

- 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 ($\forall i \in \{1, n\}, U_i < 1$), soit m le nombre de processeurs, alors il existe un ordonnanceur de $m \leq n$ 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 *« Si un ensemble de m tâches est ordonnancé selon l'algorithme Rate-Monotonic, alors le facteur d'utilisation minimum réalisable est $m \times (2^{\frac{1}{m}} - 1)$ »*.

Ils ajoutent :
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 d'ordonnançabilité 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 d'ordonnançabilité 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 multi-processeur [17].

Cet algorithme a lui aussi été adapté à une utilisation multi-processeur en y joignant des heuristiques de bin-packing : Dans [40], 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 (*next fit decreasing*)

et proposent une analyse de la complexité. Ils renvoient eux-mêmes vers [30] 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 ordonnançabilité 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 [44].

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 [42]. Enfin, une autre limite est que ce processus ne permet pas à une tâche d'évoluer avec le temps. Ses propriétés doivent être fixes. 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 [30]. 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* [1.0.2]. Dans l'article [7], *Baruah* compare les deux techniques pour des systèmes de tâches *sporadiques*, et ses recherches montrent plusieurs résultats importants :

Lemme 1 *il y a des systèmes de tâches ordonnançables avec des algorithmes globaux à priorité fixe sur travail que des algorithmes partitionnés à priorité fixe sur travail ne peuvent pas ordonnancer.*

Lemme 2 *il y a des systèmes de tâches ordonnançables avec des algorithmes partitionnés à priorité fixe sur travail que des algorithmes globaux à priorité fixe sur travail ne peuvent pas ordonnancer*

Ceci est prouvé en détaillant des contre-exemples dans l'article et permet de conclure :

Théorème 2 *Les ordonnanceurs globaux et partitionnés à priorités fixes sur travail 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 travaux 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.Par ailleurs, il a été montré 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 ordonnançabilité 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.

2.4.2 Anomalies

Les anomalies sont décrites dans la littérature comme des changements qui intuitivement ne devraient pas affecter l'ordonnançabilité des tâches, 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 travail : 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. [30], où l'on obtient ces conditions d'ordonnançabilité :
$U(\tau) \leq \frac{(m+1)}{2}$ 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 [21], Hong et Leung montrent :« *Pour tout* $m > 1$*, il n'existe pas d'ordonnanceur en ligne optimal pour les ensembles de travaux avec au moins deux échéances distinctes* ». Un autre article de Fisher et al [19] élargit cela aux tâches sporadiques. Ces résultats, quoi que négatifs, ne peuvent être généralisés à toutes les classes de tâches.
C'est un résultat embarrassant, 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.

2.4.6 PFair

En 1996, Baruah et al. décrivent l'algorithme *PFair*. [6] L'article apporte plusieurs définitions importantes :
Soit x, une tâche. Posons que $x.w$ représente le « poids » , défini de cette façon :
$x.w = \frac{x.e}{x.p}$ où $x.e$ représente le nombre d'unités de temps nécessaires à la réalisation de la tâche x et $x.p$ la période. Ainsi, $0 < x.w < 1$.

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 travail,* t *un instant, et où* $S(x, t) = 1$ *si le travail* x *est exécuté à l'instant* t*,* 0 *sinon.*

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

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 travail en sous-travaux qui devra s'exécuter dans une fenêtre de temps, considérée comme sous-échéance. Dès lors, à chaque sous-échéance, chaque travail 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.

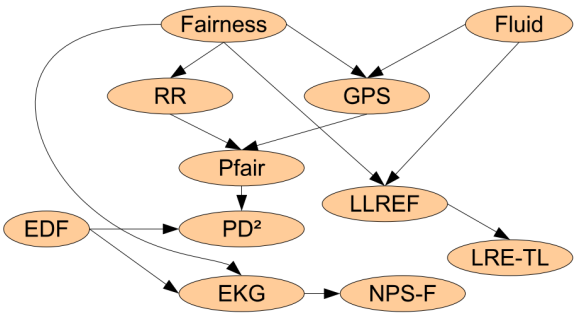
Toutefois, *PFair* est optimal pour les tâches synchrones à échéances implicites, selon la condition suivante :

Définition 3 *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 $U_{max} \leq 1$ [6].*

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* [45] , *ER* [1], et d'autres moins « classiques » comme *DP-Fair* [27], *LB-Pfair* (Loop-Back Proportionate Fair) [25]. 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

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



enjeu très important 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 [38]), 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 4 *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 [23] (Earliest Deadline Critical Laxity).

Des travaux récents continuent d'implémenter des versions d'*EDF* avec stratégie globale, par exemple *GEDF* [28] [35], donnant lieu à *PGEDF*. Cette branche continue donc d'être étudiée et présente des intérêts.

2.4.8 Global-EDF

Cet algorithme est une généralisation d'EDF [2.1.3] en version multicœur. Dans Global-EDF, on considère un ensemble de tâches ordonné par échéances, où la priorité p de la tâche τ_i est plus élevée si son échéance est plus proche. Pour $\tau_1 : d(t) = a$ et $\tau_2 : d(t) = b$, si $a < b$, alors $p(\tau_1) > p(\tau_2)$.

La décision d'ordonnancement est prise à l'instant t de cette façon :
Si un cœur est disponible, il effectue la tâche de priorité supérieure qui reste à effectuer. L'algorithme n'est pas optimal, et dans le cas de systèmes périodiques, il faudra s'assurer de la faisabilité du système sur un temps au moins aussi long que

l'hyper-période [1.0.6]

2.4.9 U-EDF

U-EDF est présenté en 2011 par Nelissen et al. [36]. 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. [37]. Le principal but de cet algorithme est de réduire le nombre de préemptions, ce qui est un grand inconvénient de l'approche globale. Soit m le nombre de processeurs, notons que dans le cas particulier où $m = 1$, alors U-EDF $\equiv$ EDF.

2.4.10 RUN

Reduction to UNiprocessor (**RUN**) est un algorithme présenté par Regnier et al. en 2013 [43]. 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* (Quasi Partitioned Scheduling) 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. [31] 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*. L'avantage de *QPS* est de pouvoir adapter sa stratégie en fonction de la charge. En effet, durant l'exécution, *QPS* peut s'adapter, passant d'une stratégie *globale* à *partitionnée*, ou le contraire. Pour ce faire, *QPS* va diviser les tâches en sous-systèmes, classés différemment selon leur besoin en processeur :

- *mineur*, si le sous-système a besoin d'un seul processeur
- *majeur*, si le sous-système a besoin de plusieurs processeurs

Par conséquent, si tous les sous-systèmes sont mineurs, *QPS* fonctionne comme *EDF* partitionné. Dans le cas contraire, l'exécution dépendra des besoins des sous-systèmes, et pourra être composé de plusieurs *serveurs QPS* en mode multi-processeur, ou en *EDF* sur simple processeur, cumulant deux modes. Les résultats attendus de *QPS* sont donc très prometteurs, car il permet de conjuguer les avantages des approches partitionnées ainsi que globales.

Chapitre 3

Éléments du contexte

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

in : « *Computing machinery and intelligence* »

3.1 Choix de l'ordonnanceur

3.1.1 Enjeux du choix de l'ordonnanceur

L'état de l'art nous a montré un vaste choix d'algorithmes qui présentent des intérêts différents, et pour un certain nombre, n'ayant pas encore été implémentés.

La première étape du travail consiste donc à départager ces algorithmes pour arrêter notre choix sur l'un d'entre eux. Ceci nécessite de clarifier nos attentes concernant le choix de l'ordonnanceur.

Pour rappel, les objectifs de ce travail sont de procéder à une implémentation dans un **RTOS** dans plusieurs buts. D'abord, en implémentant un algorithme qui n'a pas d'implémentation connue, nous pouvons vérifier que ce passage de la théorie à la pratique est possible (car cela n'est pas toujours évident), nous pouvons également mesurer ses performances. Plus largement, nous pouvons émettre des propositions à l'attention de chercheurs afin qu'ils puissent envisager de développer certains points lorsqu'ils proposent de nouveaux algorithmes.

Il est donc primordial de se tourner vers un algorithme qui pourra apporter sur le plan de l'expérience des analyses intéressantes, c'est à dire exploitables pour la communauté. Insistons peut-être sur le fait que des résultats ne sont pas nécessairement chiffrés mais peuvent être des constats qualitatifs quant à la mise en œuvre elle-même.

3.1.2 Pourquoi UEDF

Le choix le plus rationnel vis à vis des objectifs devrait se faire en fonction des promesses théoriques de performance et de stabilité. En effet, pour faire avancer la connaissance et permettre d'augmenter la confiance des utilisateurs potentiels, il est judicieux de choisir un algorithme dont on attend au moins ceci :

- Un algorithme global
- Qui minimise le nombre de migrations

- L'ordonnanceur est optimal pour la classe périodique
- Il n'a pas bénéficié d'une implémentation sur un **RTOS**
- Il promet des performances intéressantes

Le premier choix a été RUN [2.4.10], puis finalement, son descendant, QPS [2.4.10]. Ces choix étant guidés sur leur intérêt d'ordonnanceurs en tant que tels. Toutefois, l'aspect très théorique des papiers les présentant a rendu la première phase de travail difficile. En outre, nous n'avons pas trouvé d'implémentation ou de simulation malgré nos recherches. Finalement, nous n'avons pas réussi à entrer en contact avec les créateurs de ces algorithmes.

En effet, la possibilité de contacter les créateurs d'un algorithme peut singulièrement améliorer le travail. Cela étant, cette nécessité découle du caractère très théorique de la littérature disponible et du manque fréquent de prise en considération des contraintes pratiques d'implémentation. Pour illustrer cette remarque, un exemple assez commun est la non prise en considération du fait que le fonctionnement des ordinateurs est évidemment discret, là où les calculs présentés se basent sur un fonctionnement continu. Toute mise en pratique nécessite de faire des arrondis, ceux-ci doivent être cohérents.

Nous pouvons aussi évoquer les moments où l'algorithme doit faire des calculs. Dans un modèle, on peut arrêter l'exécution quand on le veut, avec des temps de surcoût négligeables. Dans la réalité, comme on le verra, cela n'est pas le cas. Les auteurs qui connaissent bien ce fonctionnement auront la possibilité de devancer certaines questions. Néanmoins, dans le cas d'un papier très théorique qui s'emploie à faire des démonstrations mathématiques, cela n'apparaîtra pas forcément.
En d'autres termes, l'implémentation d'un ordonnanceur peut s'avérer une tâche bien compliquée selon le degré de description pratique et d'anticipation des problèmes que les auteurs auront pris la peine d'aborder dans les papiers publiés. **UEDF** bénéficie quant à lui d'une littérature qui développe mieux l'aspect pratique.

Finalement, donc c'est un argument quant à la communication (clarté, possibilité de poser des questions) qui a fixé le choix. Par conséquent, c'est **UEDF** qui a été

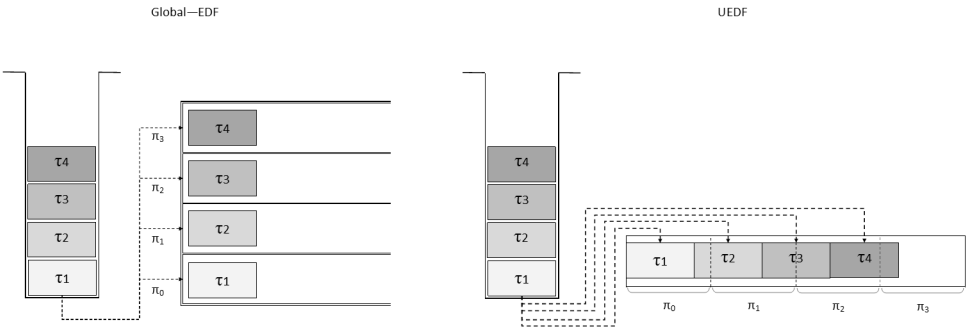
sélectionné, car non seulement il respectait les promesses énoncées auparavant, mais aussi, en plus d'être très bien documenté, nous pouvions poser nos questions directement à des personnes ayant participé à son élaboration, puisqu'il est le résultat de recherche d'équipes au sein de l'**ULB**. Néanmoins, ses paramètres sont moins idéaux, et nous avons revu nos attentes quant à l'efficacité à la baisse. Cela n'a en rien modifié l'objet scientifique, à savoir le regard critique et la proposition d'améliorations afin de stimuler et faciliter des implémentations futures, mais cela a sans doute diminué les performances finales obtenues, tel que nous le verrons dans la partie Résultats [6.1].

3.2 Présentation UEDF

Nous avons déjà présenté précédemment **UEDF**, de façon globale et succincte. Dans cette partie, nous allons détailler l'algorithme afin d'en avoir une meilleure compréhension. Cela permettra de mesurer la différence entre les attentes théoriques et les résultats obtenus.

Avant toute chose, **UEDF**, comme **Global-EDF**, sont des généralisations multi-cœur d'**EDF**. Pour **Global-EDF**, on parle d'une généralisation verticale, là où on parle d'une généralisation horizontale pour **UEDF**, et cette dénomination provient du fait que **Global-EDF** considère les processeurs « verticalement » là où **UEDF** les considère à l'horizontale, comme on peut le voir dans ce schéma.

FIGURE 3.1 – Généralisations d'EDF, verticalement, horizontalement



En d'autres termes, Global-EDF va décider d'attribuer une tâche par processeur libre à un instant t alors qu'UEDF va « remplir » l'utilisation de chaque processeur jusqu'à 100% avant de passer au suivant.

3.2.1 Définitions spécifiques

Nous posons quelques définitions qui sont nécessaires à la compréhension de l'algorithme. Par ailleurs, par souci de clarté, nous simplifions les calculs en supposant que les offsets [1.0.1] des tâches sont nuls dans la partie qui suit, cela ne change pas l'algorithme mais rend les explications moins complexes.

Tâche active

Ce qui définit une tâche « active » peut différer selon le contexte. Dans l'approche d'**UEDF**, une tâche est considérée comme active si un travail a été relâché et que ce travail n'a pas encore atteint son échéance. Un travail τ_i est actif s'il a été relâché sur ou avant t et que $d_i(t) \geq t$. Une tâche peut donc être considérée comme active même si l'exécution d'un travail en cours est terminée. Concrètement, si une tâche

est périodique à échéance implicite [1], elle est active en permanence dès le premier relâchement de travail. Le travail sera actif durant son relâchement jusqu'à ce que $t \geq d_i(t)$, où une nouvelle instance de la tâche τ_i sera relâchée.

Ensemble des tâches actives

L'ensemble des tâches actives est écrit $A(t)$ dans les formules qui suivent. Notons que dans un système périodique à échéances implicites [1], cet ensemble sera toujours composé des mêmes tâches.

Dans **UEDF**, on effectue le parcours d'une liste des tâches actuellement actives dans le système, cette liste étant ordonnée par échéances pour les besoins de l'algorithme. On notera $sorted\_ActiveList(t)$ la liste triée créée à partir de l'ensemble des tâches $A(t)$.

Pour décrire **UEDF**, nous allons le décomposer en étapes. À chaque fois qu'un nouveau travail est relâché, cela va provoquer le calcul d'$Allot_{ij} \forall i \in A(t), \forall j \in \{1...m\}$.

Allot

Les tâches actives [3.2.1] sont donc ordonnées dans la liste $sorted\_ActiveList(t)$. La priorité est définie selon l'échéance, la plus petite étant la plus prioritaire, comme dans EDF [2.1.3]. Dans cette phase, l'algorithme va « réserver » des tranches de temps d'exécution des tâches sur les processeurs, cela en calculant un tableau de valeur :
$allot_{ij}$ pour $i \in \{0, nb\_de\_taches - 1\}$, pour $j \in \{0, nb\_processeurs - 1\}$. La valeur $allot_{ij}$ représente le nombre d'unités d'exécution de la tâche i réservé sur le processeur j.
Si cette valeur est positive, alors le temps $allot_{ij}$ du travail i est réservé sur le processeur j. Si cette valeur est égale à 0, aucun temps n'est réservé à l'exécution de la tâche τ_i sur le processeur π_j. Cela permet la constitution d'une liste, **Eligible**.

Les calculs sont longuement décrits et expliqués dans la thèse de G. Nelissen [37], aussi nous invitons le lecteur curieux à se référer à ce document pour de plus amples détails. Mais en résumé, ce calcul se déroule de cette façon :

Nous posons :

- $delay = d_i(t) - t$
- $ret_i = wcet-$ le nombre d'unités de temps déjà exécutées
- L'opérateur $[x]_a^b \stackrel{\text{def}}{=} \max\{a, \min\{b, x\}\}$

Dans l'algorithme qui suit, le but est d'« allouer » le temps qui reste à exécuter pour chaque travail actif aux processeurs présents. Pour ce faire, on calcule la valeur maximale $allot_{max}$ que l'on peut réserver sur un processeur, en faisant appel à la somme de l'*utilisation* des travaux déjà traités (W). Cela permet de calculer le minimum entre ce maximum et le nombre d'unités d'exécutions qui restent effectivement à allouer pour le travail τ_i, et finalement, de l'allouer dans $allot_{ij}$.

Algorithm 1 Compute Allot

Require: $sorted\_ActiveList(t)$
  $W = 0$
  for all $j \leftarrow \{1...m\}$ **do**
    $RES_j \leftarrow 0$
    $ALLOT_j \leftarrow 0$
  end for
  for all $\tau_i \in sorted\_ActiveList(t)$ **do**
    $prev_i \leftarrow 0$
    for all $j \leftarrow \{1...m\}$ **do**
      $RES_j \leftarrow RES_j + \{[W]_{j-1}^{j} - (j-1)\} \times (d_i(t) - d_{i-1}(t))$
      $allot_{max} \leftarrow delay - ALLOT_j - RES_j - prev_i$
      $allot_{ij} \leftarrow min\{allot_{max}, ret_i - prev_i\}$
      $prev_i \leftarrow prev_i + allot_{ij}$
      $ALLOT_j \leftarrow ALLOT_j + allot_{ij}$
    end for
    $W \leftarrow W + U_i$
  end for

Dans l'exemple qui suit, notre système est composé de deux tâches :

1. $\tau_1 : \{o : 0; w : 30; d = p : 60; \}$
2. $\tau_2 : \{o : 0; w : 60; d = p : 80; \}$

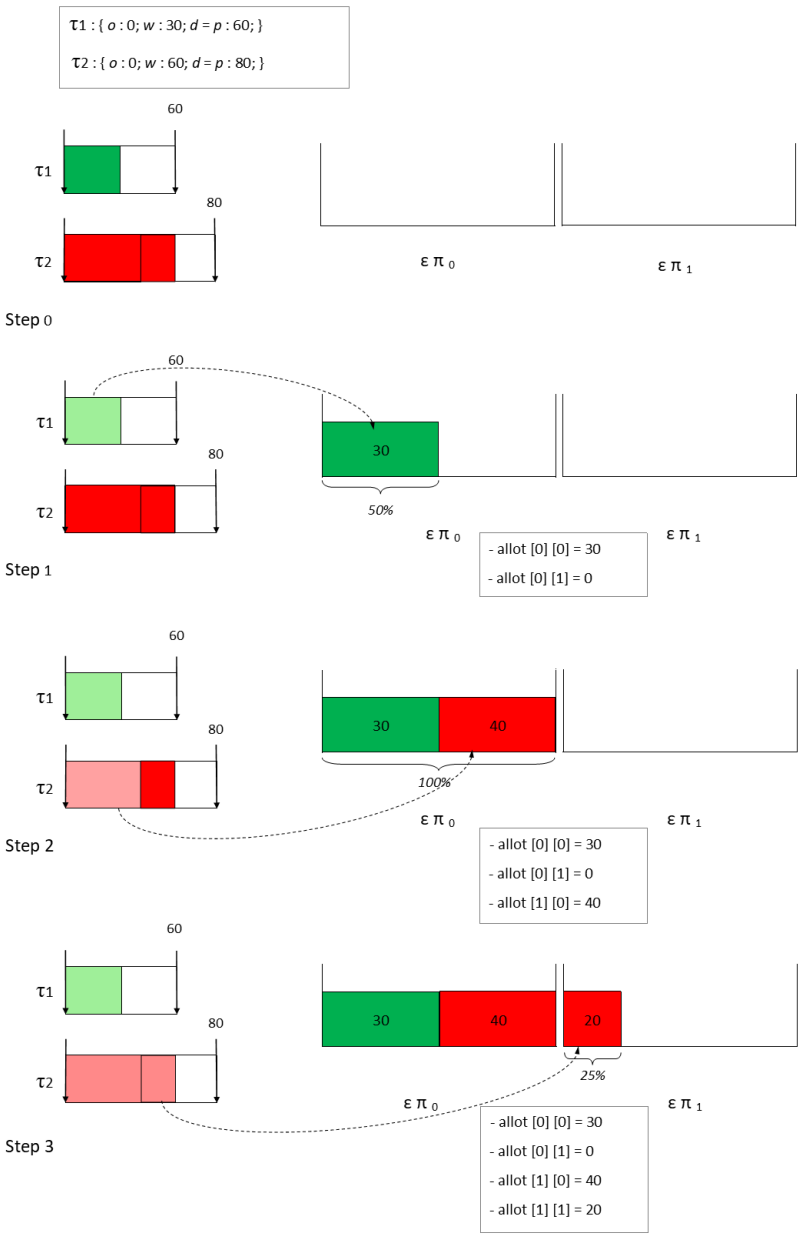


FIGURE 3.2 – Étapes de calcul UEDF

Eligible

À ce stade, aucune décision d'ordonnancement n'est encore prise, mais des ensembles de tâches (*Eligible*, noté ϵ dans la suite) sont créés sous forme de listes. Chaque liste *Eligible* définit les seules tâches que les processeurs associés peuvent exécuter (en cela, cette phase d'**UEDF** peut être considérée comme une phase de

partitionnement) durant l'exécution à venir.

Notons que les calculs fournis par l'algorithme permettent en théorie de « remplir » la charge d'un processeur à 100% d'utilisation avant de passer au suivant. Concrètement, cela signifie que si la première tâche a une utilisation de moins de 100%, elle sera suivie par au moins une partie d'une autre.

Les valeurs d'$Allot$ ne doivent pas être considérées comme le temps d'exécution qui sera réellement exécuté sur le processeur. Il faut garder à l'esprit qu'à chaque relâchement de travail, $Allot$ sera recalculé.

Prise de décision

La phase de décision de l'ordonnancement se déroule lors de la phase suivante. **UEDF** s'inspire pour finir de **EDF-Delay**. Cet algorithme est très simple :

- Une tâche $\tau_i \in \epsilon_j, j \in m$ est attribuée à un processeur π_j si aucun autre processeur d'index plus bas n'est en train de l'exécuter
- τ_i s'exécute sur le processeur π_j tant qu'$allot_{ij} > 0$

Déroulement

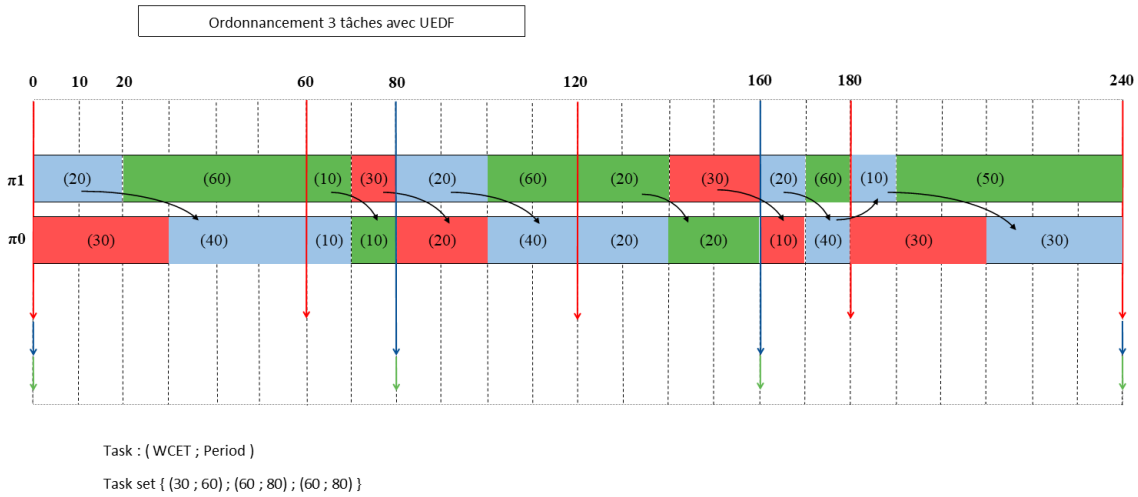
L'algorithme $ComputeAllot$ [3.2.1] est à effectuer à chaque relâchement de tâche. La valeur $Allot_{ij}$ doit être mise à jour à chaque prise de décision. Une décision est attendue pour chacun des événements suivants :

- Une instance de tâche (travail) a été relâchée
  $\sum_{i \in A(t)} \sum_1^m allot_{ij} = WCET$
- Le temps alloué sur un processeur a été exécuté, dans ce cas
  $\exists i \in A(t), \exists j \in m : allot_{ij} = 0$
- Un travail est terminé complètement :
  $\sum_{i \in A(t)} \sum_1^m allot_{ij} = 0$.

On peut voir en théorie l'ordonnancement de trois tâches, afin de mieux comprendre le déroulement d'une exécution. À titre de remarque, le travail d'illustration d'une exécution nous semble important pour procéder à une implémentation, sinon le

risque est d'omettre le développement d'outils indispensables au bon déroulement de l'exécution. L'algorithme n'est pas trivial à mettre en œuvre.

FIGURE 3.3 – Ordonnancement avec UEDF



Les nombres écrits sur les cases représentant le temps exécuté par un travail sur un processeur sont les valeurs $Allot$, on peut constater qu'elles sont différentes du temps qui sera en réalité exécuté.

Nous pouvons déjà formuler ici une nuance par rapport à la théorie et les attentes que l'on peut en avoir en pratique : nous savons dès le départ que l'algorithme **UEDF** est assez gourmand en calcul puisque chaque relâchement de tâche va provoquer l'exécution de l'algorithme $ComputeAllot$, or, le calcul de cette valeur implique un parcours de toutes les tâches ordonnées, pour chaque processeur. Outre qu'il est nécessaire de gérer une structure de données efficace, nous pouvons déjà considérer que le tri de la structure sera d'un coût non négligeable lors de l'exécution.

Par ailleurs, le calcul même de $ComputeAllot$ implique lui aussi une certaine complexité. Ainsi :

- soit m le nombre de processeurs
- soit n le nombre de tâches actives [3.2.1] du système
- $ComputeAllot \in O(m \times n)$

Et pour finir, régulièrement, la valeur $allot_{ij} \forall i \in TaskSet, \forall j \in m$ doit être mise à jour, dont la complexité est donc :

- soit m le nombre de processeurs
- soit n le nombre de tâches actives [3.2.1] du système
- $updateAllot \in O(m \times n)$

Nous pouvons attendre un surcoût important sur cette base. Une hypothèse étant que ce surcoût augmente avec le nombre de tâches, mais aussi avec le nombre de cœurs. Un même système pourrait avoir plus de surcoût avec plus de cœurs.

3.2.2 Comparaison avec Global-EDF

Nous avons choisi de comparer **UEDF** à **Global-EDF** [2.4.8]. La raison de ce choix est simple : cet algorithme est implémenté sur le RTOS HIPPEROS et il est global, comme l'indique son nom. Les autres ordonnanceurs disponibles sur HIPPEROS sont pour la plupart partitionnés.
Aussi les tests effectués sur **UEDF** seront également faits en utilisant **Global-EDF** pour élément de comparaison.

Ces deux algorithmes ont toutefois de grandes différences. **Global-EDF** ne permet pas — même théoriquement — d'atteindre l'optimalité. Ceci s'explique par le fait que **Global-EDF** peut être considéré comme « vertical » là où **UEDF** serait « horizontal ». Mais une grande différence entre les deux algorithmes est que **Global-EDF** n'est pas optimal, en théorie, contrairement à **UEDF**.

Fonctionnement de Global EDF

L'algorithme est relativement simple, et pour les besoins de la comparaison, nous résumons ici son fonctionnement.
À un moment t, l'ordonnanceur prend sa décision de cette façon : si un processeur

est libre, il se voit attribué le travail de priorité supérieure parmi tous les travaux actifs et pas encore attribués. Cela permet d'avoir un algorithme peu gourmand.

Algorithm 2 Global-EDF

Require: $JobList(t)$ ordonné par échéances
  for all $j \in m$ **do**
    $decision \leftarrow pop(JobList(t))$
  end for

L'algorithme permet de prendre une décision au moment t en ne considérant que les travaux à exécuter par ordre de priorité et les processeurs disponibles.
Si l'on reprend l'exemple illustré précédemment, la décision sera prise de cette façon :

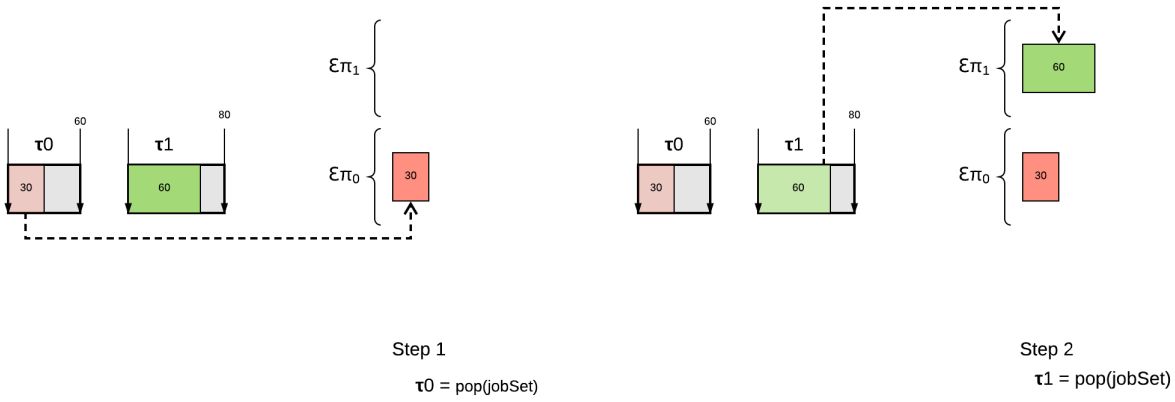


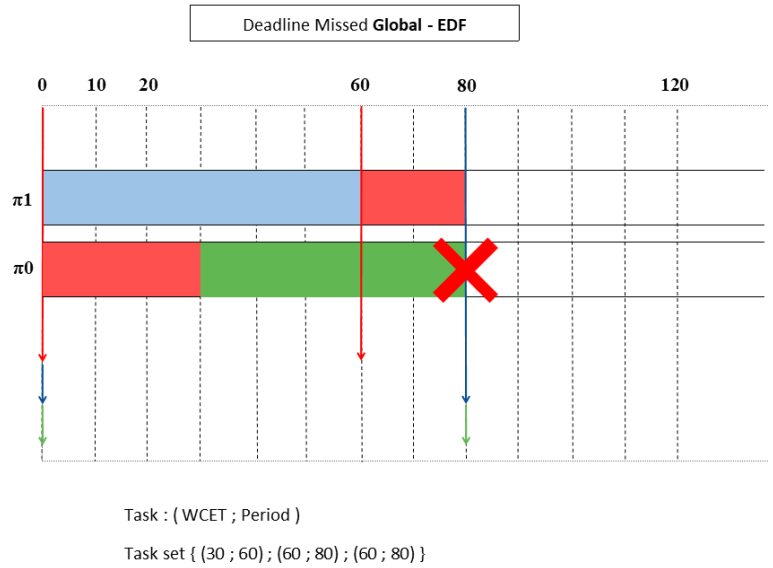
FIGURE 3.4 – Global EDF

On peut facilement voir que cela peut entraîner de « mauvais choix ». En changeant l'exemple précédent, et en prenant un exemple avec 3 travaux, on n'obtiendra pas du tout le même ordonnancement dans les deux algorithmes, et celui-ci mettra en échec **Global-EDF** quasiment immédiatement.

Prenons un ensemble de trois tâches comme suit :

1. $\tau_1 : \{o : 0; w : 40; d = p : 60; \}$
2. $\tau_2 : \{o : 0; w : 40; d = p : 60; \}$
3. $\tau_3 : \{o : 0; w : 40; d = p : 60; \}$

FIGURE 3.5 – Ordonnancement avec Global-EDF



Malgré ce point, **Global-EDF** est « efficace » en terme de calculs. Le point le plus complexe concerne la structure de données qui conserve les tâches courantes. Une bonne idée est d'utiliser un Heap de Heaps [Mok [32]] qui va permettre de fournir toujours en tête le travail de priorité supérieure. Pour s'assurer de la faisabilité de l'ensemble de tâches, il faudra tester l'ordonnancement suffisamment longtemps (hyper-période).

3.3 HIPPEROS

HIPPEROS (**HI**gh **P**erformance **P**arallel **E**mbedded **R**eal-time **O**perating **S**ystems) est un **RTOS** (Real-Time Operating System) 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 principalement deux installations en fonction des particularités.

HIPPEROS est un candidat idéal pour l'implémentation d'un ordonnanceur global. Il a cependant un fonctionnement propre qui pourra rendre l'implémentation

plus ou moins facile. Par exemple, HIPPEROS, à l'inverse de *Linux*, a un ordonnanceur asymétrique (un processeur est *Master*, et les autres sont *Slaves*). En d'autres termes, l'ordonnancement est calculé uniquement sur l'un des processeurs, qui sera désigné au moment de la compilation. Cela simplifie l'implémentation, puisque l'on ne doit pas s'occuper de savoir quel processeur est actuellement en train d'effectuer l'algorithme d'ordonnancement.
En résumé, une nouvelle implémentation sur un OS différent peut elle-aussi apporter à la connaissance générale des détails importants.

Notons qu'HIPPEROS est un RTOS privé, avec les conséquences logistiques que l'on peut imaginer : la documentation est privée également, et pour comprendre le kernel, la seule ressource est l'équipe de développeurs qui l'a créé.

3.4 Attentes

Des présentations des différents acteurs qui ont été faites dans les parties précédentes, à savoir :

1. UEDF
2. Global-EDF
3. HIPPEROS

Nous pouvons déjà formuler certaines hypothèses que nous aimerions vérifier dans ce travail.

- Vérifier que l'algorithme ne comporte rien qui empêche son implémentation dans HIPPEROS
- Mesurer les surcoûts liés à l'algorithme **UEDF**, car il n'est pas évident qu'il soit possible de l'utiliser en pratique.
- Vérifier également si l'on observe bien en pratique que **Global-EDF** ne peut pas ordonnancer certains systèmes là où **UEDF** le peut. En effet, il est possible que ce résultat soit modifié ou largement à nuancer en fonction de certains paramètres, comme le WCET, dont nous reparlerons.

- Comparer le nombre de migrations/préemptions obtenues pour un même système, ordonnancé par l'un ou l'autre des algorithmes.

Chapitre 4

Implémentation

Every good idea will be discovered twice. Once by a logician, and once by a computer scientist.

Philip Wadler

Dans ce chapitre, nous développons et expliquons nos choix d'implémentation. En effet, il nous semble utile d'exposer nos choix, afin de permettre à de nouveaux candidats à l'exercice de profiter de nos essais/erreurs. Également, cela permet de comprendre les résultats qui suivent, et de proposer des améliorations pour le futur.

Données temporelles

Dans **HIPPEROS**, il existe plusieurs ordonnanceurs déjà implémentés. Ils ont leur fonctionnement propre, et de façon générale, sont assez différents d'**UEDF** par rapport aux variables auxquelles ils accèdent. **UEDF** a besoin de l'*utilisation* d'une tâche dans certains de ses calculs. Or, l'*utilisation* est une fraction. Nous souhaitons éviter la représentation à virgule flottante pour gérer ces fractions, et par conséquent, cela nécessite un décalage des valeurs.

Se pose alors la question du décalage à appliquer afin d'occuper une place raisonnable en ayant le moins de pertes possibles. La réponse à apporter diverge en fonction de plusieurs facteurs : appelons le décalage $SHIFT$.

- Pour rappel, l'*utilisation* est définie de la sorte : $U(i) \stackrel{\text{def}}{=} \frac{C_i}{T_i}$. En pratique, c'est le ***WCET*** et la période de la tâche que l'on utilise pour calculer cette fraction : $\frac{WCET}{T_i}$. HIPPEROS stocke les valeurs temporelles $wcet$ et $period$ en entiers non signés sur 64 bits.
- Le décalage $SHIFT$ doit être choisi en fonction du rang des valeurs que l'on peut obtenir. Nous pouvons négliger les tâches de moins de 1% d'utilisation pour conserver un rang de 1 à 100%. Concrètement, nous pourrons avoir les valeurs de $\frac{SHIFT}{100}$ à $SHIFT$.
- En théorie, le nombre de processeurs est libre. En pratique, notre matériel ne dépassera pas les 4 cœurs, aussi aurons-nous une utilisation maximale de 400%. Quand bien même nous irions plus loin, nous ne dépasserions pas les 8 cœurs pour des raisons matérielles, donc 800% : $SHIFT \times 8$.
- En reprenant l'algorithme en section 3.2.1, l'*utilisation* sert à multiplier la différence entre deux échéances de travaux différents. Cette valeur est libre, et on peut l'imaginer grande dans certains cas. Admettons qu'une tâche ait

une échéance de $8.000.000\mu s$, et une autre du double, comme cela peut tout à fait se croiser dans un cas pratique, on aura une différence de $8.000.000$ à multiplier par cette proportion. La valeur limite d'un entier de 64 bits étant $2^{64} - 1$ ($18.446.744.073.709.551.615$), nous pouvons estimer que la marge est grande et nous laisse libre de choisir un décalage grand.

Pour ces raisons, dans notre implémentation, nous avons estimé que décaler les valeurs de 1000 était un bon compromis. Par exemple, pour une tâche qui serait définie comme suit : $\tau_1 = \{w : 1; d = p : 3\}$, l'*utilisation* sera $\frac{1000}{5000} = 200$. Une *utilisation* de 1% est stockée dans l'implémentation d'**UEDF** comme une utilisation de 10, tandis qu'une de 100% vaudra 1000. Ce décalage sert dans tout l'algorithme décrit en section [3.2.1]. Aussi, toutes les variables seront décalées d'autant, comme l'index du processeur, afin d'avoir un calcul cohérent. Toutefois, le nombre d'unités de temps $allot$ n'est lui-même pas décalé, afin de contenir le bon nombre d'unités de temps à exécuter pour la suite.

4.0.1 Structures de données

Dans ses nombreux calculs, **UEDF** fait appel à un certain nombre de données concernant les tâches ou les travaux. Mais l'ordonnanceur a également besoin d'accéder à des variables qui lui sont spécifiques à divers moments de l'exécution. En outre, il doit également conserver les travaux n'ayant pas atteint leur échéance, car, entre deux relâchements de tâche, un travail i peut se terminer ou être préempté, il continue cependant de « peser » dans l'ordonnancement au temps t tant que $t < d_i(t)$.

Allot Les valeurs du tableau multidimensionnel $Allot$ sont primordiales et doivent être conservées durant l'exécution. Elles permettent de décider si une tâche doit être exécutée ou non. Elles devront être actualisées à chaque événement. Ainsi, on doit conserver un tableau à deux dimensions $Allot$, qui contient des données temporelles (entiers non signés de 64 bits). Cette structure est de taille $m \times n$ en théorie. En pratique, dans **HIPPEROS**, m et n sont des valeurs prédéfinies dans chaque version. Le nombre de tâches peut être à ce jour 32, 128 ou 1024. Il ne nous a pas semblé utile de changer cela. Nos tests ont tous pour configuration un nombre de

tâche maximum de 32.

L'algorithme de calcul du tableau $Allot$ ($ComputeAllot$[3.2.1]) attribue les travaux aux processeurs par ordre ascendant d'index. Notre implémentation s'est adaptée à **HIPPEROS**, car c'est le cœur 0 le *Master*, qui exécute l'algorithme d'ordonnancement. Par conséquent, nous avons inversé l'ordre, pour que le cœur 0 soit le dernier « rempli » par l'algorithme – ce – dans le but d'éviter que les surcoûts rendent des ensembles non faisables. Ainsi, pour 4 cœurs, une exécution jusqu'à 300% d'*utilisation* laissera le cœur 0 uniquement occupé à gérer l'ordonnancement.

Liste triée de tâches Il est nécessaire de maintenir à jour un ensemble trié de tâches, pour le calcul d'$Allot$. Pour cette structure, notre choix a beaucoup évolué au cours du temps. La première idée était le besoin de parcourir cette liste dans l'ordre, plusieurs fois, d'y ajouter et d'en retirer des éléments. A priori, pour ces raisons, le *Heap* ne correspond pas bien à nos besoins, aussi avons-nous utilisé une liste liée en premier lieu. Néanmoins, les mauvaises performances de la liste liée nous ont amené à reconsidérer ce choix, d'autant plus que les parcours de listes étaient nombreux. À l'heure actuelle, notre implémentation construit un *Heap* avant le recalcul d'$Allot$. Ce *Heap* est copié chaque fois avant d'en faire le parcours, dans les diverses parties du code. En pratique, lorsque des travaux sont relâchés, nous parcourons à l'heure actuelle 3 fois ce *Heap*, chaque fois copié au préalable. Ce choix a fait diminuer drastiquement le surcoût par rapport au choix de la liste liée, néanmoins reste perfectible. Par exemple, il est tout à fait imaginable de ne pas reconstruire le *Heap* à chaque fois qu'il faut calculer $Allot$, il faudrait maintenir une structure persistante, et mieux gérer les activations ou suppressions. C'est une amélioration simple qui n'a pas encore été faite, et pourrait réduire légèrement les surcoûts.

Liste d'états Une première chose à laquelle il faut être attentif est de rendre possible les accès aux données, par exemple en conservant des tableaux de références, afin de minimiser les temps d'accès. Cela nécessite de faire un choix : on optimise le temps d'accès en conservant plus de données. L'ordonnancement en cours est ainsi doublement référencé :

- $Job\_id \leftarrow coreToSchedule_{core}$ permet d'obtenir le travail en cours d'exécution sur le processeur $core \in m$
- $core \leftarrow scheduledToCore_{job\_id}$ permet d'obtenir le processeur qui exécute le travail $job\_id \in n$.

Nous devons également conserver les divers états d'un travail. En effet, rappelons que dans **UEDF**, un travail est considéré comme *actif* tant qu'il n'a pas rencontré son échéance – et ce, même s'il est *terminé*. Nous conservons donc un tableau d'états permettant d'accéder facilement à cette information. De même, nous devons différencier l'état *bloqué* d'un travail *terminé*, ce qui n'est pas le cas dans les autres ordonnanceurs implémentés dans **HIPPEROS**. Par exemple, dans **GlobalEDF**, si un travail attend une ressource, la tâche correspondante au travail bloqué sera purement et simplement retiré de la liste de tâches actives, et lorsqu'elle sera réactivée, elle sera réintroduite dans cette liste. Ce comportement n'est pas possible dans **UEDF**, qui « réserve » du temps, et « prévoit » un ordonnancement. Si le travail sort, et est simplement réintroduit, cela provoquerait un nouveau calcul d'$Allot$, basé sur le $WCET$, et pas le temps restant d'exécution.

4.1 WCET, et Temps d'exécution

Les autres ordonnanceurs implémentés dans **HIPPEROS** n'utilisent pas les données concernant le temps d'exécution de la tâche dans leur prise de décision. Pour **UEDF**, nous avons besoin d'un accès efficace et correctement mis à jour à ces données. Typiquement, le **Temps d'exécution restant**(**RET**) est utilisé pour le calcul et la mise à jour d'**Allot**.

En pratique, **HIPPEROS** met à jour le temps exécuté d'un travail lors d'un événement qui le concerne. Ainsi, si un travail est effectué, on note le moment où le travail a été dispatché. Au temps *t*, si l'on veut savoir combien de temps a déjà été exécuté, s'il n'a pas croisé d'événement, il faudra calculer ce temps dans **UEDF** car **HIPPEROS** n'aura pas encore mis à jour cette donnée.
Si en terme de temps d'accès, cette requête n'est pas très compliquée, il n'en demeure pas moins que le résultat sera un peu approximatif, puisque l'ordonnanceur n'arrête pas l'exécution des tâches pendant son exécution. Cela ne change pas

fondamentalement le calcul car cela change très faiblement les valeurs, mais cela constitue une adaptation de la théorie en pratique.

4.2 Modèle VS réel

Dans un modèle, l'exécution s'arrête, l'algorithme d'ordonnancement est effectué, puis les travaux sont dispatchés et l'exécution reprend. Le surcoût peut être totalement ignoré.

L'ordonnanceur ne fait pas partie de l'ensemble des tâches du système. Cependant, il est exécuté sur un des processeurs, et occupe une charge non nulle de temps. Outre que cela fausse les calculs liés à la charge possible (concrètement, les 100% d'utilisation libres sur le cœur qui exécute les actions de l'ordonnanceur ne sont plus 100% mais sont amputés du surcoût), cela créé un décalage entre les valeurs calculées au moment t et le moment réel de l'exécution. Il faudra adapter les WCET des tâches en fonction de cela, ce qui implique une partie du travail qui consiste à évaluer le surcoûts, ainsi que de proposer une façon de déterminer les WCET de tâches.

4.3 Événements et calcul de l'ordonnancement

Dans HIPPEROS, voici une exécution typique :

- Des tâches sont activées
- Le *dispatcher* demande à l'ordonnanceur de calculer l'ordonnancement
- Le *dispatcher* prend les décisions de l'ordonnanceur et effectue les changements s'il y a lieu (dispatcher une tâche, en préempter une autre, etc.)
- Des événements viennent réveiller le *dispatcher*, comme la fin, la préemption, ou le relâchement d'un travail. Entre deux événements, l'exécution de chacun des travaux sur les processeurs a lieu sans interruption.

Concrètement, c'est une fonction *compute_schedule* qui est appelée, sans que l'on

sache forcément quel événement a provoqué son appel.

Or, dans **UEDF**, il est nécessaire de différencier les moments où un travail a été relâché d'un autre, où l'on ne devra pas recalculer $Allot$. Il suffit dès lors d'activer un booléen indiquant qu'un nouveau travail a été relâché. Si ce booléen est vrai, l'algorithme calcule totalement $Allot$, dans le cas contraire, il ne fait que mettre ses valeurs à jour, avant de prendre dans les deux cas une décision.

4.4 Domaine système

Dans le monde des systèmes embarqués, la norme **P**ortable **O**perating **S**ystem **I**nterface (POSIX) est une spécification des fonctions de base d'un système d'exploitation. Afin de permettre la portabilité des programmes, cette norme garantit des propriétés quant à l'implémentation de certaines fonctions.

L'ordonnanceur est chargé d'ordonnancer les tâches selon leurs priorités. La norme **POSIX** définit des niveaux pour les threads : *local*, et *System*. Mais en pratique, **Linux** n'utilise pas cela, tous les *Threads* sont de niveaux *System*, et il ne gère pas de *Process* comme ensemble de *Threads*.

Une variante importante ressort ici : **HIPPEROS** gère des *Process*, qui sont des ensembles de *Threads*. L'ordonnanceur gère des *Process*, qui ont un ou plusieurs *Threads*, et les priorités pour chacun d'eux sont locales. C'est dans le but de gérer des exceptions et interruptions au niveau utilisateur qu'**HIPPEROS** a implémenté des niveaux différents de domaines. S'il est du domaine *system*, un *Process* est de priorité supérieure, quoi que décide l'ordonnanceur. Cela s'explique de cette manière :

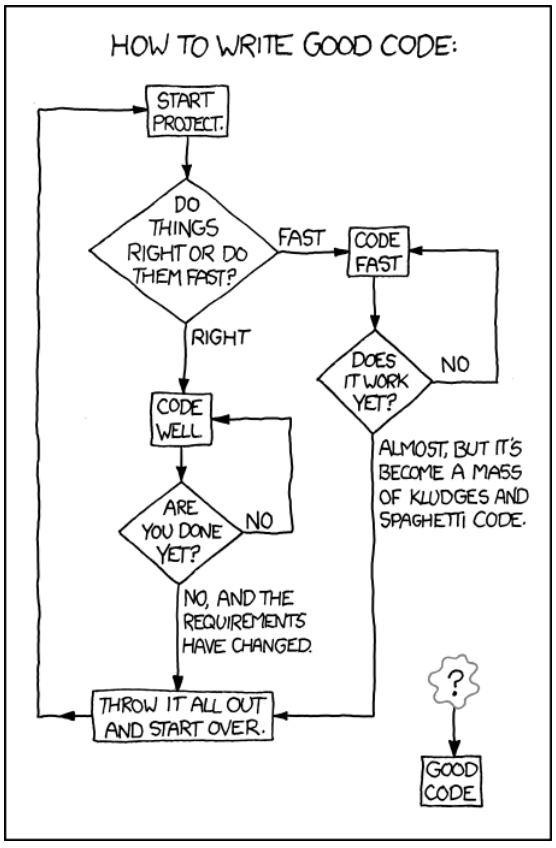
Concrètement, on imagine une sonde qui envoie très rarement des données. On ne réserve pas de temps d'exécution régulier pour effectuer cette écoute, on vérifie simplement régulièrement qu'elle a des données à traiter, et si c'est le cas, on les traite.

C'est un cas d'utilisation du domaine *System* : garantir que les données seront traitées vite, dès qu'elles arrivent, sans attribuer une tâche pour le faire régulièrement.

Notre solution pour gérer cette situation avec **UEDF** est simplement de faire passer en priorité la tâche du domaine *System* s'il y en a une, sans la comptabiliser dans **UEDF**. Cela pourrait invalider potentiellement un système, puisqu'on peut en théorie exécuter une proportion de temps qui n'a pas été comptabilisée dans l'ordonnanceur. Nous conseillons dans ce cas d'adapter d'autant le WCET des tâches.

Chapitre 5

Méthodologie



xkcd

5.1 Objectif des tests

Dans cette partie, nous allons exposer quels tests nous avons effectués ainsi que la méthodologie utilisée. Nous avons plusieurs attentes que vous souhaitons évaluer avec des tests.

Tout d'abord, dans le processus de réalisation de l'algorithme, nous devons nous assurer que notre implémentation est correcte. Nous souhaitons également vérifier par des tests sur du matériel dédié les hypothèses émises plus tôt dans ce travail [3.4], à savoir : vérifier que les surcoûts augmentent avec le nombre de cœurs, de tâches, de périodes différentes. Nous chercherons à confronter des résultats théoriques avec une réalisation. Nous pouvons également comparer la charge des processeurs avec **Global-EDF** pour des mêmes ensembles de tâches.

5.2 Matériel utilisé

HIPPEROS est peut être installé sur un nombre limité de machines. L'une d'elle est une **SABRE Lite** dont voici les spécifications :

SABRE Lite i.MX6

1. Architecture **ARMv7 Cortex-A9**
2. Processeur Freescale i.MX6 Quad 1GHz
3. 1 GB de ram
4. Mémoire sur carte micro-SD

C'est un kit de développement couramment utilisé. Les spécifications qui ont vraiment de l'importance ici sont le nombre de cœurs et la mémoire ram, le reste étant assez secondaire dans les tests que nous réalisons.

5.3 Création de tâche

Description de tâche dans HIPPEROS

Pour exécuter des tâches avec **HIPPEROS**, il faut définir des fichiers de configuration sous un format *XML*, dans ce fichier doivent apparaître certaines propriétés des tâches. Dans notre cas :

- le nom
- l'adresse de l'exécutable
- Si elles sont en temps réel
- Si elles ont une ou plusieurs occurrences {*OneShot ; Periodic ; Sporadic*}
- leur WCET
- leur période
- leur échéance

et ceci autant de fois que de tâches que l'on aura à décrire.

Les tests que nous faisons passer doivent nous permettre de contrôler au mieux la durée réelle d'exécution de la tâche, afin de maîtriser l'écart entre cette valeur et le WCET attribué dans les fichiers de configuration.

Pour ce faire, nous créons un unique programme qui sera lancé autant de fois que de tâches déclarées dans le fichier de configuration. Nous définissons un fichier de *header* contenant un tableau de valeurs. Ainsi, lors de chaque exécution, le programme va lire une valeur dans un tableau, qui représente la limite de temps qu'il devra atteindre avant de se terminer proprement.

Concrètement, un programme **Measure** possède un *Process Id*, et exécute cette boucle :

Algorithm 3 Measure_main

$boundary \leftarrow BoundaryList[Process\_id] + now$
repeat
  $t = now$
until $t < boundary$

Cette tâche périodique est relancée autant de fois que nécessaire ; c'est à dire qu'à la fin d'une boucle, elle est suspendue, puis réactivée à chaque période.

Déroulement d'un test

Nous créons un fichier contenant les informations utiles sur le test :

- WCET
- Offset
- Période
- Échéance

Nous exécutons ensuite les programmes compilés sur la *SABRE Lite*, en nous assurant qu'il n'y a pas de dépassement de WCET ou de période. Lorsque l'ensemble a été validé en test, il sera exécuté plus longuement (1 à 2h) afin d'être analysé ultérieurement.

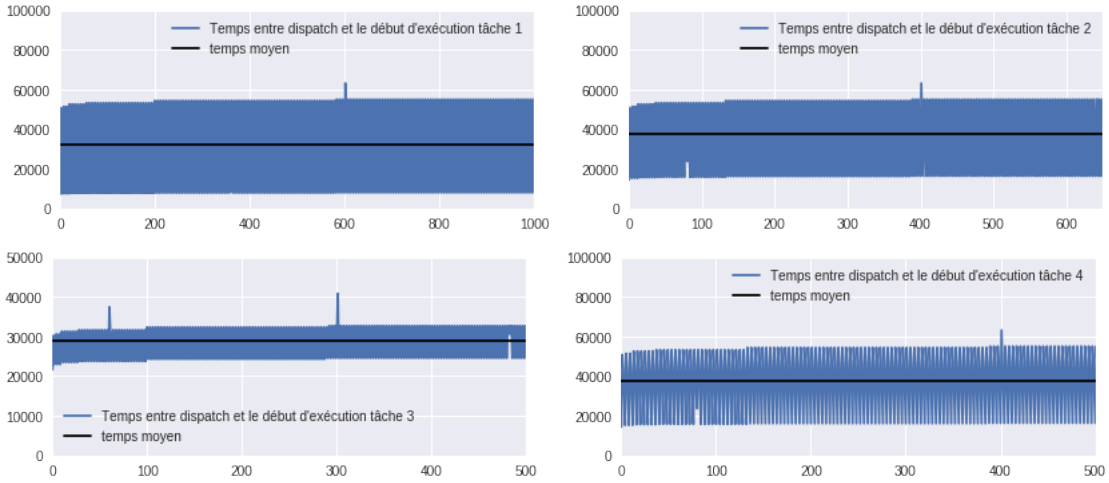
Il est important de signaler ici qu'**HIPPEROS** permet de générer des *logs*. En branchant sur port USB une sonde, il est possible de récupérer un certain nombre d'informations sur port série, comme ces événements :

- la tâche est (ré)activée
- le travail démarre son exécution
- le travail est préempté
- le travail est arrêté

Il y a un décalage entre le moment des décision et le moment d'exécution de cette décision. Par exemple, il peut s'écouler un temps relativement long entre la décision de dispatcher un travail sur un cœur et le début de l'exécution. Les logs rendent un peu compte de cet écart quoi qu'il faut avoir conscience qu'eux-mêmes faussent les données. En effet, la production de logs prend un certain temps en provoquant des écritures sur le port série. Par conséquent, les résultats présentés ici ne peuvent être considérés autrement que comme des approximations.

À titre d'illustration, voici l'exemple sur un de nos tests du décalage entre le moment où une certaine tâche est dispatchée et le début de son exécution.

FIGURE 5.1 – Temps écoulé entre une décision et le début d'une exécution



5.4 Génération des ensembles

Nous ne disposons pas d'un générateur aléatoire de tâches, et ce pour les raisons que nous allons exposer maintenant. L'ordonnanceur **UEDF** est fort sensible aux valeurs des périodes et WCET, puisque c'est avec l'*utilisation* qu'il réserve du temps d'exécution sur chaque cœur. Concrètement, on imagine bien qu'en générant de façon aléatoire des WCET, ces proportions sur les périodes ne donneront pas forcément une valeur entière.

Imaginons un cas de figure où la somme des utilisations des n tâches de priorités supérieures n'atteint la somme de 101%, dans ce cas, une tâche va réserver 1% de son WCET sur un cœur avant de devoir être préemptée, puis migrée sur le cœur précédent. Ce comportement serait trop contraignant et nous risquons de ne pas avoir de résultats intéressant à proposer. De plus, l'industrie procède généralement à une génération de tâches de périodes harmoniques, si bien que ce choix se défend également de ce point de vue. Néanmoins, cela pourrait faire partie d'un travail ultérieur.

Pour créer un ensemble de tâches, nous générons un nombre aléatoire de périodes différentes, ainsi qu'un multiple de la première période. Nous ne sommes donc pas

avec une classe de tâches de périodes harmoniques uniquement, mais proches de ce cas de figure.

Nous précisons également que les tests que nous avons faits passer comportent un nombre peu élevé de tâches, allant de 6 à 12, la majorité étant des ensembles de 10 tâches. C'est un ordre de grandeur qui s'approche d'un cas d'utilisation industriel, qui nous permettait d'obtenir des résultats avec des propriétés différentes, et conserve l'utilité des résultats obtenus.

5.5 Détermination des WCET des tâches

UEDF est optimal, en théorie. Revenons un instant sur ce point. Cela signifie que si aucun ordonnancement pour un système donné existe, aucun autre ordonnanceur ne pourra l'ordonnancer.

En pratique, le WCET d'une tâche est une valeur difficile à donner, et qui va dépendre de beaucoup de facteurs.

Elle dépend :

1. du nombre d'opérations à faire dans le code (approximation du nombre d'instructions) et de leur type (IO, calcul...)
2. de la plateforme d'exécution de la tâche (les caractéristiques de la machine)
3. de l'algorithme d'ordonnancement, des surcoûts.

5.5.1 Nombre et type d'opération

Le pire temps d'exécution peut varier en fonction du nombre d'instructions et surtout de leur type. En effet, on peut penser qu'une tâche qui doit uniquement faire du calcul prend un temps déterministe, en pratique, il y aura toujours quelques modifications.
Par ailleurs, certaines instructions prennent un temps plus ou moins variable. Par exemple, les entrées-sorties ont un temps qui peut fort varier. En outre, selon la disponibilité ou des ressources, des temps peuvent s'ajouter.

5.5.2 La plateforme d'exécution

Pour des raisons évidentes, une tâche ne mettra pas un temps équivalent sur une plateforme ou sur une autre, et le temps n'est donc pas une propriété théorique de la tâche elle-même mais dépend bien également de la plateforme. On peut bien avoir une idée en comptant le nombre d'instructions et en faisant des calculs par rapport à la fréquence du processeur, mais cela reste approximatif. Cela peut néanmoins suffire dans le cas de systèmes non-critiques.

Dans un cas idéal où le WCET ne dépendrait que de la machine et du nombre d'instructions, on aurait pu imaginer faire un certain nombre d'exécutions de la même tâche et prendre une valeur statistiquement en dehors des valeurs possibles avec certitude de $n\%$.

5.5.3 L'ordonnanceur

En fin de compte, l'ordonnanceur va ici avoir un impact sur le WCET. Concernant **Global-EDF**, le WCET ne fait pas partie des paramètres utilisés pour calculer l'ordonnancement. La variable déterminante est l'échéance absolue ($d_i(t)$). Le WCET en revanche est déterminant dans le cas d'**UEDF**. Voyons l'impact de cette variable et les risques liés à l'évaluation de cette valeur.

En bref rappel de la partie Contexte [3.2], Le WCET va permettre de calculer l'*utilisation* de la tâche. Cette proportion est utilisée pour déterminer le nombre d'unités de temps d'exécution du travail i attribuées au processeur π_j.
Nous savons qu'il y a un temps plus ou moins long entre la décision d'exécuter et le moment où le travail débute son exécution. Ceci explique pourquoi au moment de commencer les tests, il est très difficile d'évaluer le rapport entre les temps d'exécutions moyens des tâches et les WCET que nous devons fixer.

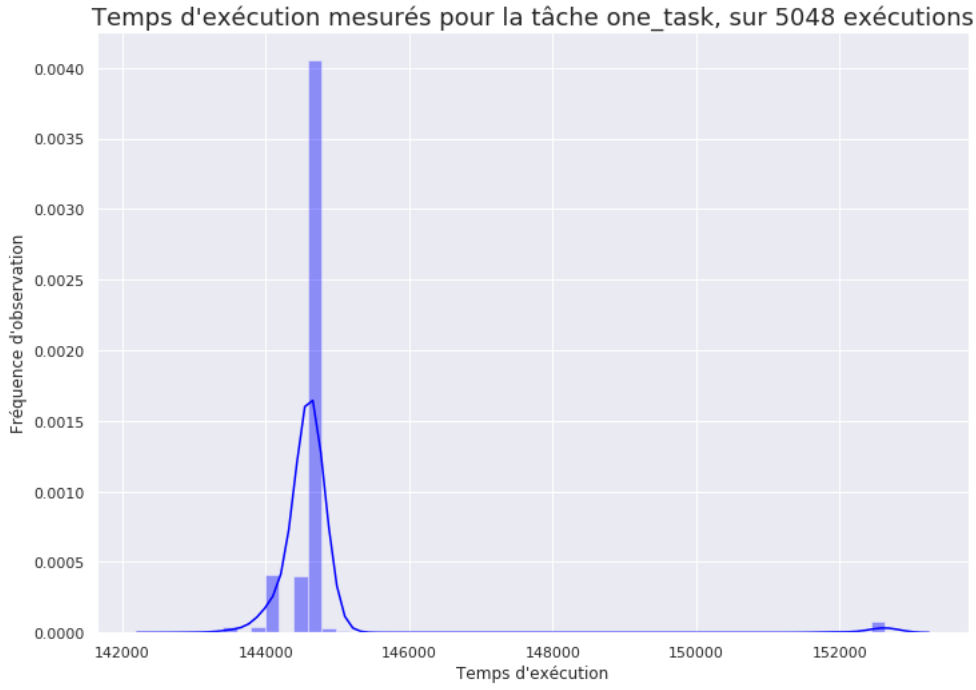
Cette procédure d'évaluation a consisté au départ à paramétrer très manuellement les WCET et durées d'exécutions demandées, pour proposer finalement une façon plus systématique de fixer ces valeurs.

5.6 Rapport WCET vs temps d'exécution moyen

La première difficulté rencontrée lors de la mise en place de tests a été de déterminer la façon de configurer les ensembles afin que ceux-ci soit ordonnançables sans provoquer de dépassement de WCET ou d'échéances tout en conservant une charge de travail la plus élevée possible, et ce, dans le but de mesurer l'évolution des surcoûts.

D'après nos tests, la façon de déterminer les WCET par rapport à la durée moyenne d'exécution de la tâche dépend du type d'ensemble, du nombre de tâches ainsi que du nombre de cœurs utilisés (donc de la somme des *utilisations* de l'ensemble). Nous pouvons en revanche considérer qu'en fixant comme WCET environ le double de la durée moyenne d'exécution, nous laissons une marge suffisante. En effet, nous avons commencé par mesurer l'évolution d'une de nos tâches sur un cœur afin de voir si sa durée d'exécution était stable ou non.

FIGURE 5.2 – Évolution du temps d'exécution moyen d'une tâche



Ce graphique nous montre qu'une variabilité existe bien, mais les extremums restent très proche de la valeur moyenne.

Nous avons finalement opté pour donner des valeurs de WCET d'environ deux fois plus grandes que les durées d'exécution moyennes, ceci étant basé sur nos nom-

breux essais.

5.7 Récolte des résultats

Afin de pouvoir analyser les performances de l'implémentation d'**UEDF**, il nous faut être capable de reconstituer l'historique d'une exécution.

Cela est possible en produisant des sorties (*logs*) que nous parsons à l'aide d'outils développés en *Python*.

5.8 Comparaison avec G-EDF

Les expériences réalisées avec **UEDF** ont toutes été également réalisées avec **Global EDF**. Les mêmes ensembles de tâches ont été rejoués en changeant juste l'ordonnanceur.

5.9 Premiers tests de calibration, somme d'utilisation de 100 %

Pour une somme des utilisations de 100%, **UEDF** va se comporter comme un ordonnanceur monocore, même s'il dispose de plusieurs cœurs. Cela ne présente pas beaucoup d'intérêt d'étudier la répartition de la charge, puisque ce résultat est totalement attendu.

En revanche, la configuration – autrement dit : le calibrage – diffère, puisque **Global-EDF** n'utilise pas du tout le WCET pour prendre ses décisions, contrairement à **UEDF**. Cette donnée doit donc être fixée avec une grande attention.

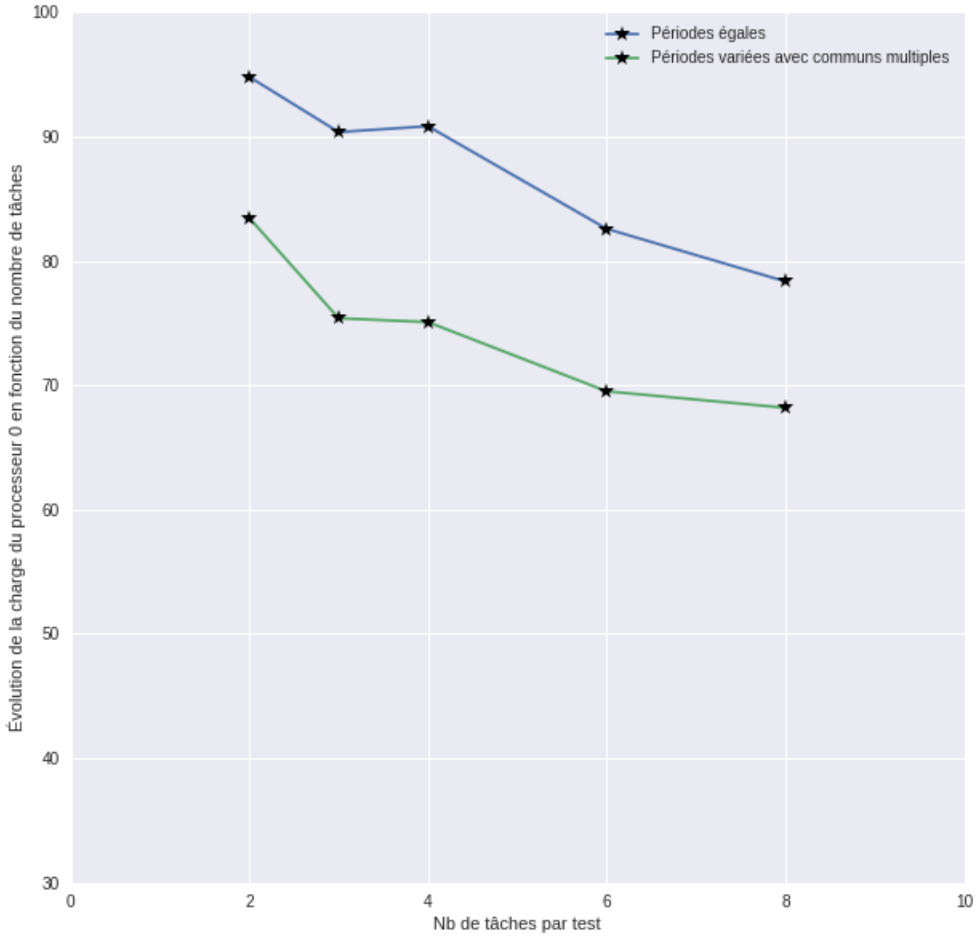
Ce premier graphique montre l'évolution de la charge du cœur 0 dans une série d'ensembles dont la somme des utilisations ($\sum_0^{n-1} U(i)$) est de 100%. Cela nous permet de tester plus facilement l'évolution de la charge possible d'un processeur en fonction du nombre de tâches.

Un ensemble est choisi s'il ne provoque pas de dépassement de WCET ou d'échéance durant une exécution mais qu'en variant de quelques dizaines de millisecondes la durée moyenne d'exécution de la tâche, une de ces erreurs arrive. Cela signifie que l'on s'approche par essai/erreur de la limite au delà de laquelle nous trouvons un dépassement.

Les premières courbes montrent l'évolution du rapport entre les temps d'exécution moyens et les WCET renseignés dans la configuration de la tâche. Dans ce premier test, nous avons exécuté et analysé l'exécution de 5 ensembles de tâches composés respectivement de 2, 3, 4, 6 et 8 tâches.

Par construction, la première courbe indique l'évolution pour des tâches de périodes toutes égales comme référence, et la seconde indique l'évolution sur base de périodes de soit harmoniques, soit proches de l'harmonie.

FIGURE 5.3 – Évolution de la charge d'utilisation en fonction du nombre de tâches, somme d'utilisation de 100%



Cette illustration est ainsi développée dans ce travail afin de montrer l'effort de calibration, ainsi que de démontrer une différence avec d'autres ordonnanceurs pour lesquels le calcul du WCET des tâches est sans doute moins crucial. À l'issue de cette analyse, ainsi que d'autres essais non détaillés dans ce document, nous constatons que le cœur 0, qui se trouve être le cœur *Master* dans le système d'exploitation **HIPPEROS**, sera fort sollicité à exécuter les décisions, migrations, etc. Une façon simple d'améliorer la répartition de la charge sera exposée dans le chapitre Perspectives [7]. Sans cette amélioration dans le code, nous forcerons les ensembles ultérieurs à avoir une charge bien moins élevée sur le cœur 0 que sur les autres, ce qui rend les systèmes plus facilement ordonnançables. En ayant conscience de la façon de procéder de cet ordonnanceur, concrètement, nous seront plus pessimiste sur le temps moyen d'exécution des tâches dont les périodes sont les plus petites dans nos tests.

Chapitre 6

Résultats

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times ; premature optimization is the root of all evil (or at least most of it) in programming.

D. Knuth

in : « Computer Programming as an Art »(1974)

6.1 Résultats sur l'implémentation

6.1.1 Implémentation possible

Une des questions auxquelles nous devions répondre était la possibilité d'implémentation d'**UEDF**, qui n'avait bénéficié jusque là d'aucune implémentation sur un RTOS. Le travail effectué a d'ores et déjà montré qu'il était possible de l'implémenter, ce qui est un premier résultat attendu. Il est néanmoins important de préciser que cette implémentation n'a pas atteint sa phase d'optimisation et que la volonté dans ce travail a été avant toute chose d'avoir une implémentation fonctionnelle qui mettrait déjà en avant des caractéristiques intéressantes de cet ordonnanceur. Le lecteur est d'autant plus invité à prendre les résultats non pas pour une affirmation générale mais bien comme issue d'une première implémentation « exploratoire » qu'elle est observée en vis–à–vis d'une implémentation de **Global-EDF** déjà optimisée. Nous donnerons des pistes d'amélioration dans le chapitre suivant.

6.1.2 À la lisière entre théorie et pratique

Nous avons montré au chapitre Contexte [3.2] que certains ensembles en théorie n'étaient pas ordonnançables en considérant les WCET comme temps d'exécution, mais bien par **UEDF** . Dans la pratique — comme on en a déjà parlé dans le chapitre Méthodologie [5.5] — les WCET ne doivent jamais être des temps d'exécution, et cette limite peut être plus ou moins éloignée de la durée moyenne d'exécution d'une tâche. Nous avons également parlé du fait que l'ordonnanceur doit lui aussi être pris en considération pour paramétrer cet écart entre moyenne des temps d'exécution et WCET.

Nous avons souhaité tester deux ensembles afin d'illustrer notre propos ici. Ces ensembles sont constitués respectivement de 6 et 7 tâches de périodes non harmoniques, dont les hyper-périodes n'étaient pas excessivement grandes par rapport à la plus petite des périodes.

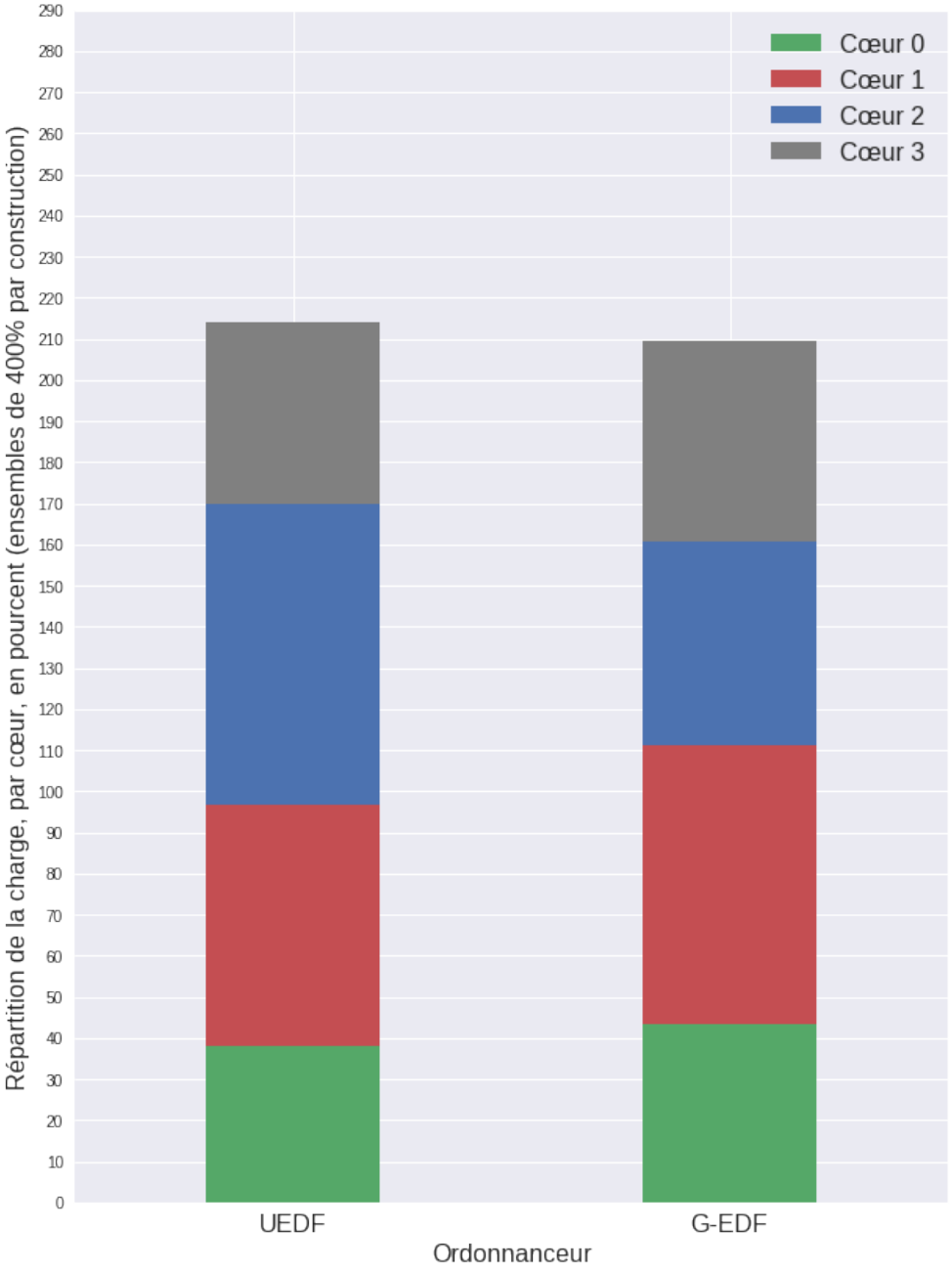
- Hyper-période = 6 fois la plus petite période pour le premier ensemble
- Hyper-période = 8 fois la plus petite période pour le second

Ces ensembles, en théorie, ne devraient pas être ordonnancés par **Global-EDF**, si les WCET étaient peu éloignés du temps moyen d'exécution. [41]

Répartition de charge de travail

Une première analyse de la répartition de charge de travail sur les 4 cœurs, en fonction de l'ordonnanceur, ne montre pas de grande différence, comme on peut en juger sur la figure 6.2.

FIGURE 6.1 – Comparaison charges de travail **UEDF**, **Global-EDF**



La charge totale du système est proche de 200%. La différence qui s'observe entre les deux totaux s'explique par les légères variations d'une exécution à une autre, or notre échantillon est petit pour cette expérience puisque nous n'avons que deux ensembles par ordonnanceur, le but n'étant pas de généraliser mais bien de montrer un type de comportement qui a de l'influence.

Nombre de migrations

En l'absence d'optimisations qui empêchent ce comportement, par défaut,**UEDF** va provoquer un nombre important de migrations, tandis que dans le même temps, **Global-EDF** n'en produit aucune. Sur les deux ensembles, nous avons souhaité rendre compte du nombre de migrations en calculant ce nombre d'événements sur l'hyper-période, que nous nommons « densité de migration », qui est la somme des migrations durant l'hyper-période divisée par le nombre de tâches. Nous trouvons, pour **UEDF** :

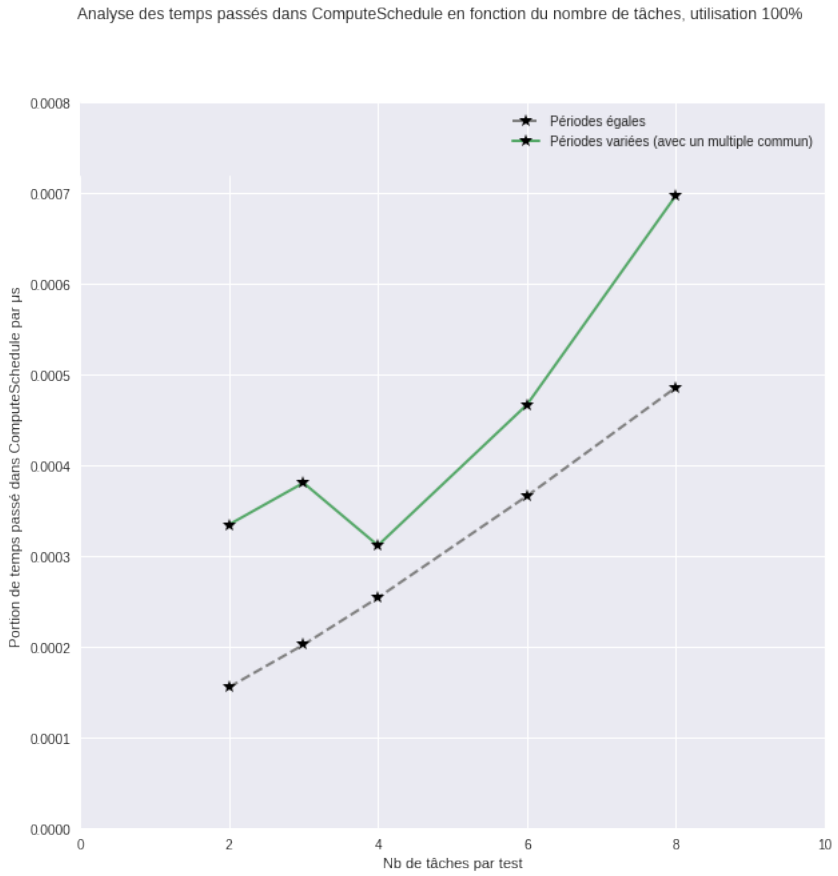
1. 10 migrations par hyper-période pour le premier ensemble de 6 tâches, soit une densité de $1,66$
2. 6 pour le second, de 7 tâches soit une densité de $0,85$
3. aucune pour **Global-EDF** dans les deux systèmes

Les migrations vont provoquer du surcoût. Ce surcoût doit être intégré dans le paramétrage du WCET. Ceci explique pourquoi en l'état actuel de l'implémentation, et avec ce type d'ensembles de tâches de périodes non harmoniques, **UEDF** ne montre pas sa supériorité par rapport à **Global-EDF**, on peut même dire l'inverse, car la répartition de charge est légèrement meilleure pour **Global-EDF**, qui déploie beaucoup moins d'efforts et de calculs qu'**UEDF** pendant le même temps.

6.2 Temps passé dans l'algorithme

Nous avons voulu vérifier l'évolution du temps passé dans la fonction $ComputeSchedule$, qui nous semblait critique au début de ce travail. Nous présentons ici un schéma qui montre l'évolution du temps passé pour 1 unité d'exécution dans cette fonction. C'est donc la proportion de temps passée dans cette fonction par rapport au nombre de tâches.
Cette expérience n'a été réalisée qu'avec des ensembles de 100% d'utilisation théorique.

FIGURE 6.2 – Évolution du temps passé dans ComputeSchedule, **UEDF**



On voit bien que ce temps évolue, néanmoins, il reste à des valeurs qui nous semblent totalement acceptables. La courbe des ensembles non harmoniques n'est pas linéaire. Ceci s'explique par le fait que les ensembles n'ont pas un nombre de périodes non harmoniques linéaire non plus, et que par conséquent, par « hasard », il y a plus de variété dans les ensembles de 3 tâches que de 4. On peut néanmoins s'attendre à ce que cette courbe suive cette tendance en augmentant le nombre de tâches. Nous pensons qu'il est plus important de viser de diminuer les préemptions et autres migrations que d'optimiser — du moins en premier lieu — cette partie du code.

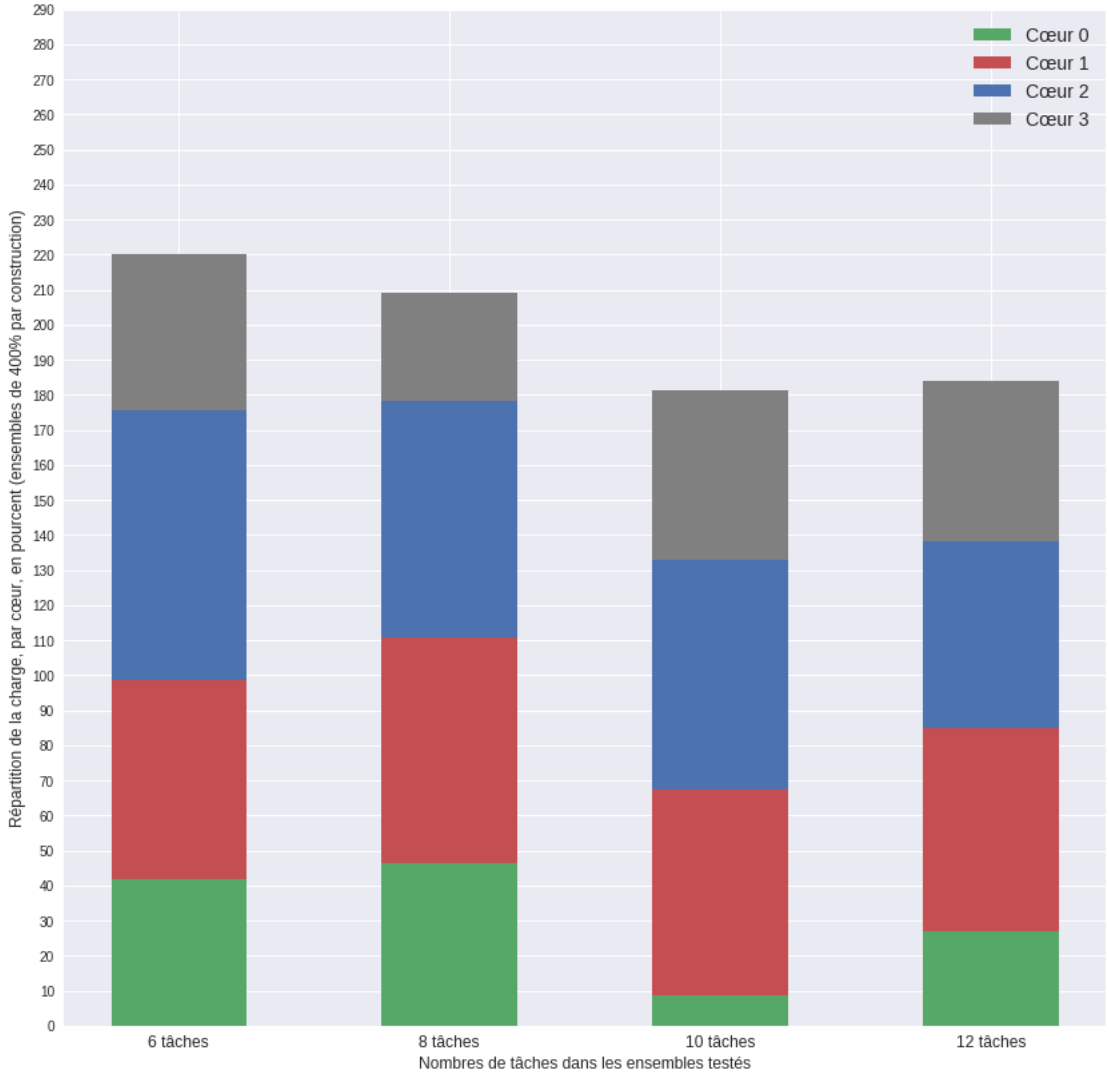
6.3 Ensembles variés dont la somme d'utilisation est de 400 %

6.3.1 UEDF

Dans les expériences suivantes, nous avons constitué un nombre variable d'ensembles de 6, 8, 10 et 12 tâches, et avons procédé à l'analyse de la répartition de charge des processeurs.

Les expériences précédentes nous ont mené à produire des ensembles constitués de périodes harmoniques, ou « presque ». Ainsi, pour éviter d'avoir des périodes extrêmement longues, certaines périodes ont des multiples communs avec les autres périodes mais ne sont pas directement multiples. Leur nombre est limité, aussi, nous pouvons caractériser ces ensembles de « simples ».

FIGURE 6.3 – Comparaison charges de travail **UEDF**



La charge moyenne, dans ce cas, dépasse de peu les 200%, ce qui semble plutôt convenable. Nous rappelons ici la différence entre une charge théorique et observée : la charge théorique étant celle basée sur les WCET, éloignés d'aussi peu que possible des temps moyens d'exécution tout en garantissant de ne pas arriver à cette frontière. La charge de travail observée dans l'expérience ci-dessus concerne une exécution réelle, avec des temps d'exécution forcément plus faibles que les WCET. La charge totale ne peut jamais atteindre $100\% \times m$ car les temps de surcoût et de « précaution » par rapport aux WCET s'ajoutent à la somme.

Un phénomène apparaît, cependant, qui est une charge moins importante pour le premier cœur que pour les autres, et également moins importante pour le dernier

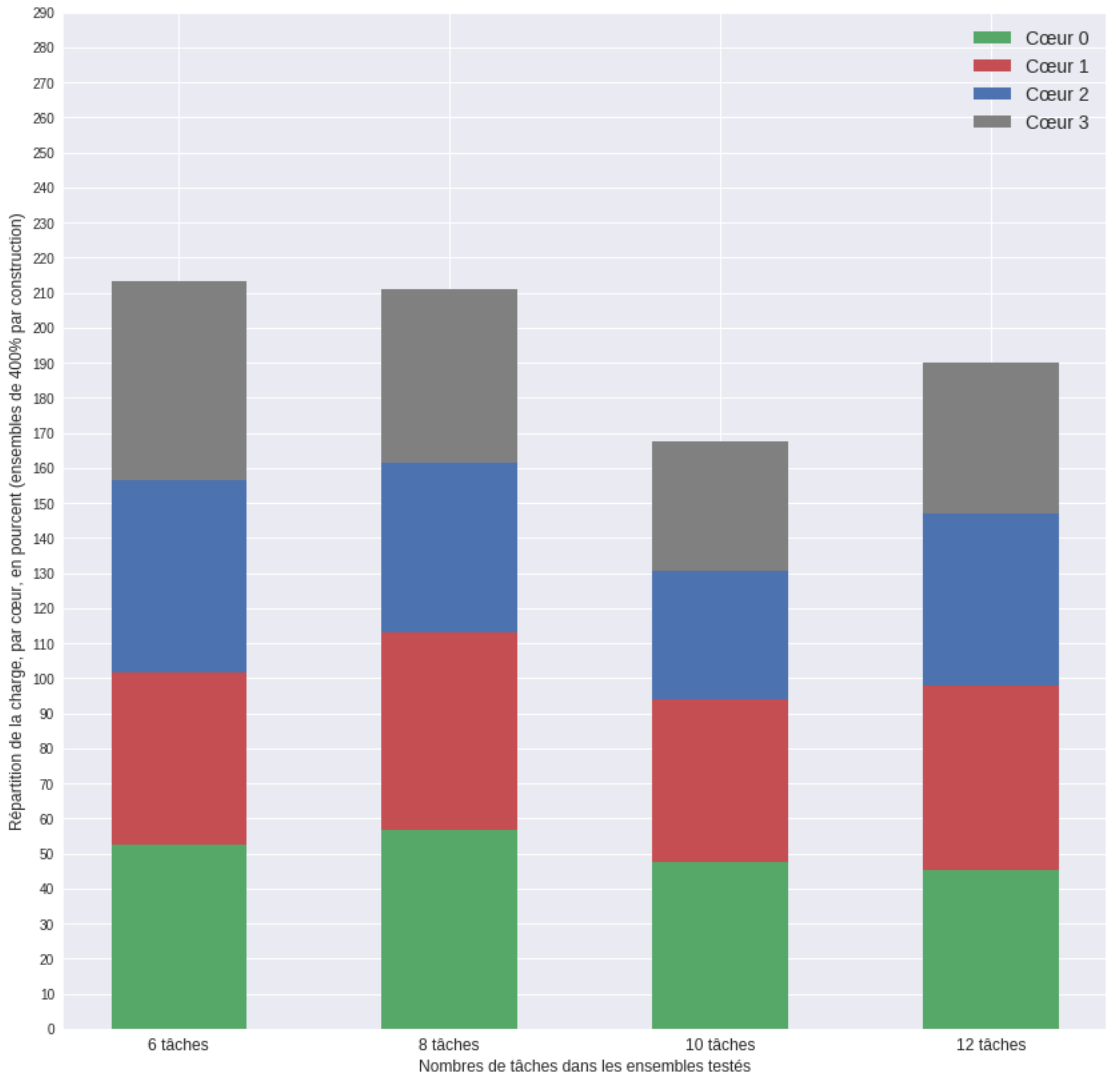
cœur. Cela s'explique facilement :

- Nous avons volontairement été pessimiste avec les tâches de périodes les plus petites qui s'exécutent sur le premier cœur car c'est le *Master*, aussi une grande partie des surcoûts pèsent sur sa charge totale.
- D'après le fonctionnement d'**UEDF**, le dernier cœur est chargé d'exécuter les travaux d'échéances les plus éloignées, aussi, comme les écarts entre les temps moyens et les WCET sont de près du simple au double, le travail sera souvent terminé voire très avancé dans son exécution lorsqu'il sera déployé sur le dernier cœur.

6.3.2 Global-EDF

Nous avons rejoué les mêmes ensembles en utilisant cette fois **Global-EDF**. Nous pouvons constater que la charge totale est répartie différemment, puisqu'au contraire d'**UEDF**, nous ne voyons pas spécialement d'écart entre les cœurs : la répartition est équitable entre tous les cœurs.

FIGURE 6.4 – Comparaison charges de travail **Global-EDF**



Pour cette expérience encore, la supériorité d'**UEDF** n'a pas été démontrée. En revanche, comme nous en parlerons dans la partie Perspectives [7], en procédant à quelques changements dans l'implémentation, nous pensons que ces résultats peuvent grandement s'améliorer, et en fin de compte, ces premiers indices peuvent être considérés comme prometteurs pour la suite.

Chapitre 7

Perspectives et propositions

It's important to remember that when you start from scratch there is absolutely no reason to believe that you are going to do a better job than you did the first time. First of all, you probably don't even have the same programming team that worked on version one, so you don't actually have "more experience". You're just going to make most of the old mistakes again, and introduce some new problems that weren't in the original version.

Joel Spolsky

Dans cette partie, nous revenons sur le travail effectué et proposons des voies d'amélioration. Ces propositions portent sur plusieurs domaines :

- Des améliorations concernant l'implémentation
- De nouveaux tests pour compléter nos premières observations
- plus largement sur la littérature scientifique, lorsqu'une équipe souhaite proposer un algorithme nouveau

7.1 Poursuivre l'implémentation

L'état actuel de notre implémentation n'est pas totalement satisfaisante sur bien des points et gagnerait fort à être continuée.

7.1.1 Inverser les cœurs

Les résultats nous ont montré que le cœur *Master* avait une charge de travail plus basse que les autres cœurs. Cette observation découle directement d'un artefact : nous avons volontairement augmenté les écarts entre le WCET et le temps moyen d'exécution de certaines tâches afin de rendre **UEDF** capable d'ordonnancer. En l'absence de cette manipulation, nous avons constaté de nombreux dépassements et n'arrivions pas à trouver d'ensembles satisfaisants à étudier.

Il est tout à fait possible d'éviter cette manipulation, et la modification n'est pas des plus compliquées : en effet, en inversant l'ordre des processeurs dans l'algorithme, il est possible de changer cette tendance. Dans ce cas, on peut tout à fait imaginer que le dernier cœur serait naturellement le moins chargé, et que par conséquent, aucune manipulation particulière ne soit nécessaire afin de rendre l'algorithme bien plus performant au niveau de la répartition des charges de travail.

Un premier essai a été fait, et a montré que le changement méritait de l'attention pour éviter d'incorporer des erreurs, néanmoins reste de complexité tout à fait abordable.

7.1.2 Structures de données

L'implémentation et ses structures ont été exposées plus haut dans document. Il serait sans aucun doute efficace d'optimiser les diverses structures utilisées.

Une amélioration viendrait d'un moyen efficace de conserver le *Heap* et de le maintenir à jour durant l'exécution. En effet, actuellement, ce *Heap* est reconstruit à chaque relâchement de tâche. Il gagnerait à n'ajouter que les nouvelles tâches et à retirer celles qui sont terminées. Cette amélioration a été testée brièvement, et s'est révélée prometteuse.

7.1.3 Ordonnanceur virtuel

Dans la thèse de G. Nelissen [35], une amélioration est proposée et serait très intéressante à ajouter à notre implémentation. En effet, afin de limiter le nombre de migrations, l'auteur propose de conserver une copie des décisions « virtuelles » et de vérifier au moment d'une migration si celle-ci peut être évitée, en échangeant la liste *Eligible* d'un cœur donné avec celle d'un autre cœur. Dans ce cas, le nombre de migrations devrait réduire drastiquement, ce qui pourrait aider à diminuer de beaucoup les surcoûts. Le coût d'une telle opération en complexité est relativement élevé, toutefois, nous avons vu que le temps de calcul est moins coûteux que les migrations, aussi ce serait une dépense justifiée et très efficace.

7.2 Tests supplémentaires

7.2.1 Compléter les résultats actuels

En premier lieu, beaucoup de résultats mériteraient d'être plus détaillés. Il y aurait encore d'autres résultats à produire, comme l'analyse du temps d'exécution de $ComputeSchedule$, mais sur plus d'échantillons, comme un détail plus fin de l'écart type (nous savons que le temps passé dans cette fonction dépend beaucoup du fait qu'il y a eu un relâchement de tâche auparavant ou pas, mais n'avons pas en l'occasion de présenter ces résultats). En bref, il resterait d'autres résultats à montrer et détailler depuis les expériences que nous avons menées.

7.2.2 De nouveaux résultats dans de nouvelles recherches

Nous avons limité nos recherches et expériences à des ensembles de tâches de périodes harmoniques [1.0.7], ce qui est justifié par plusieurs arguments, développés dans la partie Méthodologie [5.5]. Néanmoins, en apportant les quelques modifications exposées ici, nous pensons qu'il serait envisageable et intéressant de produire des résultats pour des ensembles de tâches de périodes non harmoniques, avec des rapports plus complexes entre elles. Nous attendrions de constater une meilleure répartition de la charge de travail, et beaucoup moins de migrations, et ce résultat doit encore être démontré par des expérimentations.

7.3 La littérature scientifique

Dans le cas de **UEDF**, les documents disponibles montrent un effort de l'équipe afin de rendre compréhensible l'algorithme et permettre sa mise en œuvre. Comme cela a été dit précédemment, cet effort n'est pas lisible dans tous les documents que nous avons eu l'occasion de parcourir, ce qui rend l'implémentation compliquée. Le niveau de complexité d'un algorithme en lui-même n'est pas forcément équivalent à la difficulté d'implémentation dans un environnement déjà écrit, partagé avec d'autres ordonnanceurs. Aussi, si cette complexité est déjà très grande, nous pensons que les chercheurs devraient tout mettre en œuvre afin de rendre le passage du papier à l'implémentation plus facile, en illustrant avec des exemples les calculs, en donnant des conditions, des invariants, et pourquoi pas proposer des cas particuliers auxquels il faut particulièrement être attentifs.

Chapitre 8

Conclusion

L'objectif initial de ce travail était avant tout de choisir un ordonnanceur issu de la littérature scientifique n'ayant bénéficié d'aucune implémentation dans un véritable RTOS. Au terme d'une revue de l'état de l'art sur les ordonnanceurs actuels, nous avons choisi de développer une implémentation d'**UEDF**, un algorithme qui a été élaboré par une équipe de chercheurs de l'**ULB**, pour ses propriétés mais également car la communication autour de lui nous semblait accessible.

Au terme d'une première implémentation dans **HIPPEROS**, nous pouvons déjà affirmer qu'aucun élément ne rend cet algorithme non implémentable dans cet RTOS.

Par ailleurs, au terme des tests que nous avons fait afin de mesurer ses comportements, nous pouvons dire :

- sa répartition de charge dans le cas d'ensembles de $m \times 100$ % n'est pas mauvaise, mais le dernier cœur n'a pas autant de charge que les autres. Ceci devrait être amélioré en apportant quelques modifications, et nous pensons que rien ne s'oppose à ce que ce résultat s'améliore grandement.
- Des optimisations telles que de maintenir un ordonnancement virtuel afin d'éviter de nombreuses migrations sont indispensables, sinon l'algorithme perd de son intérêt, et ce, même face à un algorithme décrit comme n'étant pas optimal dans la littérature. En d'autres termes, cela fait perdre l'optimalité d'**UEDF** de procéder à son implémentation « naïve ».
- L'algorithme détermine ses décisions en se basant sur le WCET de la tâche, or, cette valeur dépend elle-même de l'algorithme d'ordonnancement. Cela comporte des avantages, néanmoins rend le calibrage des ensembles délicat.

Au terme de ce travail, nous pensons plus largement à la littérature scientifique décrivant d'autres algorithmes. Le travail d'implémentation s'est avéré complexe pour bien des raisons, et si une équipe souhaite que son algorithme bénéficie d'une implémentation dans un véritable RTOS, il nous semble absolument indispensable que les papiers dépassent le stade de la preuve mathématique. Le monde de l'industrie comme des sciences bénéficierait de multiples travaux de ce genre afin d'augmenter la confiance en des solutions théoriques non éprouvées.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Aloysius Mok. Task management techniques for enforcing ed scheduling on a periodic task set. 01 1988.

[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, April 2014.

[35] Geoffrey Nelissen. *Efficient optimal multiprocessor scheduling algorithms for real-time systems*. PhD thesis, Université libre de Bruxelles, Ecole polytechnique de Bruxelles, January 2013.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. Global EDF scheduling of directed acyclic graphs on multiprocessor systems. In *Proceedings of the 21st International conference on Real-Time Networks and Systems*, pages 287–297, Sophia Antipolis, France, October 2013.

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

[43] 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.

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

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