



Algorithmes pour l'ordonnancement temps réel multiprocesseur

Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche

► **To cite this version:**

Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche. Algorithmes pour l'ordonnancement temps réel multiprocesseur. Journal Européen des Systèmes Automatisés (JESA), Lavoisier, 2015, 48 (7-8), pp.613-639. <<http://jesa.revuesonline.com>>. <10.3166/JESA.48.613-63>. <hal-01221711>

HAL Id: hal-01221711

<https://hal.archives-ouvertes.fr/hal-01221711>

Submitted on 30 Oct 2015

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

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

1 Algorithmes pour l'ordonnancement 2 temps réel multiprocesseur

3 **Maxime Chéramy**¹, **Pierre-Emmanuel Hladik**¹,
4 **Anne-Marie Déplanche**²

5 1. CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
6 Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France
7 *maxime.cheramy@laas.fr, pehladik@laas.fr*

8 2. Université de Nantes, IRCCyN UMR CNRS 6597, ECN,
9 1 rue de la Noë, BP92101 F-44321 Nantes cedex 3, France
10 *anne-marie.deplanche@irccyn.ec-nantes.fr*

11 *RÉSUMÉ. Cet article expose les principales politiques d'ordonnancement temps réel existantes*
12 *pour des architectures multiprocesseurs et homogènes. Cette présentation a pour but de gui-*
13 *der le lecteur à travers la très grande profusion d'articles dans ce domaine et lui fournir les*
14 *principales clefs de compréhension des algorithmes. L'étude se focalise sur les approches dites*
15 *globales et hybrides et en montre leur très grande variété.*
16

17 *ABSTRACT. This article presents the main real-time scheduling policies available to homogeneous*
18 *multiprocessor architectures. The objective of this study is to guide the reader through the va-*
19 *riety of scientific publications in this domain and to give him the main keys for understanding*
20 *these algorithms. The article focuses on global and hybrid approaches and shows their consi-*
21 *derable diversity.*

22 *MOTS-CLÉS : ordonnancement, multiprocesseur, temps réel, algorithmes.*

23 *KEYWORDS: scheduling, multiprocessor, real-time, algorithms.*

24
25
26

27 **1. Introduction**

28 L'ordonnancement temps réel pour les architectures multiprocesseurs a connu un
29 regain d'intérêt ces dix dernières années qui s'est traduit par un très grand nombre
30 de publications scientifiques. Sur vingt ans, plus de cinquante politiques d'ordonnan-
31 cement ont été proposées et ont été accompagnées de nombreuses études sur leurs
32 propriétés. Depuis la synthèse de Davis et Burns (2011), réalisée en 2009, une dizaine
33 de nouveaux algorithmes ont été développés. La grande profusion de publications rend
34 difficile une vue synthétique dans ce domaine.

35 Nous proposons dans cet article de brosser un panorama des techniques d'ordon-
36 nancement en-ligne pour les systèmes temps réel sur des architectures multiproces-
37 seurs. Nous nous focalisons sur les principales politiques d'ordonnancement du point
38 de vue de leur comportement fonctionnel et évoquons leurs propriétés sans pour au-
39 tant les approfondir. Notre objectif est d'initier le lecteur à ce domaine très actif de
40 la recherche, de le guider dans la littérature existante et lui offrir, à travers les nom-
41 breuses références qui jalonnent cette présentation, les points d'entrée pour une étude
42 approfondie d'une politique en particulier.

43 Pour des raisons de place, les politiques exposées se limitent au cas d'architec-
44 tures à processeurs identiques, un lecteur intéressé par la prise en compte d'autres
45 types d'architecture peut consulter les publications de Funk (2004) et plus récemment
46 de Yang et Anderson (2014). De même, les politiques d'ordonnancement dites parti-
47 tionnées sont rapidement abordées. Ce sujet ayant peu évolué depuis quelques années,
48 un lecteur souhaitant approfondir ce point peut consulter l'article de Davis et Burns
49 (2011).

50 Cet article est ainsi organisé : la section 2 introduit rapidement les notations et
51 les définitions incontournables du domaine. La section 3 présente les grandes familles
52 d'algorithmes pour l'ordonnancement multiprocesseur afin d'avoir un aperçu synthé-
53 tique des stratégies mises en œuvre dans ce domaine. Elle se termine par un tableau
54 présentant, de manière classée et avec références, un grand panorama de ces poli-
55 tiques d'ordonnancement. Les trois sections suivantes exposent les principales po-
56 litiques d'ordonnancement globales et hybrides, afin de fournir au lecteur les clefs
57 nécessaires à leur compréhension. Enfin, une ouverture vers les difficultés liées à leur
58 évaluation est faite lors de la conclusion.

59 **2. Modélisation et vocabulaire**

60 Cette première partie introduit les principales notations et le vocabulaire employés
61 pour modéliser un système temps réel. Quelques définitions relatives à l'ordonnan-
62 cement sont aussi présentées afin de décrire les propriétés des politiques d'ordonnan-
63 cement.

64 **2.1. Modèle**

65 En matière d'ordonnancement temps réel multiprocesseur, un système est consti-
 66 tué d'un ensemble de tâches à exécuter sur un ensemble de processeurs. Une **tâche**
 67 contrôle le flot d'exécution d'un programme. Les instructions exécutées forment ce
 68 que l'on appelle un **travail**. Ainsi, si elle est récurrente, une tâche donne lieu à une
 69 succession de travaux, appelés aussi **instances** de la tâche. Les travaux d'une applica-
 70 tion temps réel doivent respecter un certain nombre de contraintes temporelles.

71 Le modèle de tâches décrit ici a été initialement formulé par Liu et Layland (1973).
 72 Ce modèle permet de traiter le cas de tâches périodiques et a été généralisé aux tâches
 73 sporadiques (Mok, 1983 ; Leung, Whitehead, 1982). Un système temps réel τ est
 74 constitué d'une ensemble fini de n tâches : $\tau = \{\tau_1, \dots, \tau_n\}$. Chaque tâche τ_i est
 75 constituée d'une suite infinie de travaux notée $\{\tau_{i,1}, \tau_{i,2}, \dots\}$ où $\tau_{i,j}$ est le j ème travail
 76 de la tâche τ_i . Une tâche τ_i périodique ou sporadique est caractérisée par le triplet
 77 (C_i, T_i, D_i) avec C_i la durée d'exécution dans le pire cas (*Worst-Case Execution*
 78 *Time*, WCET) de chaque travail de la tâche τ_i ; sa période d'activation T_i ou la durée
 79 minimale entre deux activations successives dans le cas de tâches sporadiques ; son
 80 échéance relative ou délai critique D_i , c'est-à-dire la durée entre le réveil d'un tra-
 81 vail et son échéance (un travail $\tau_{i,j}$ qui s'active à l'instant $a_{i,j}$ doit se terminer avant
 82 l'instant $d_{i,j} = a_{i,j} + D_i$). Une tâche est dite à **échéance implicite** si $D_i = T_i$ et un
 83 système **synchrone** si toutes les tâches ont leur première activation à la même date.

84 La **laxité** d'un travail $\tau_{i,j}$ est la marge temporelle qu'il lui resterait par rapport
 85 à son échéance absolue, si à l'instant t il était exécuté pendant sa durée d'exécution
 86 restante, soit $L_{i,j}(t) = d_{i,j} - t - c_{i,j}(t)$ avec $c_{i,j}(t)$ le temps d'exécution restant de
 87 $\tau_{i,j}$ à l'instant t . Lorsqu'un travail devient à laxité nulle, cela signifie que s'il n'est pas
 88 immédiatement exécuté, il ne pourra pas respecter son échéance.

89 Le **taux d'utilisation** d'une tâche correspond à la fraction de temps que la tâche
 90 consomme sur un processeur pour s'exécuter, $u_i = C_i/T_i$. Cette grandeur est très utili-
 91 sée dans les tests d'ordonnançabilité, ainsi que la **somme totale des taux d'uti-**
 92 **lisation**, $u_{sum} = \sum_{i=1}^n u_i$ ou encore le **plus grand taux d'utilisation** du système,
 93 $u_{max} = \max\{u_1, \dots, u_n\}$.

94 Dans la suite de cet article, nous nous limitons à des systèmes pour lesquels l'archi-
 95 tecture matérielle est constituée d'un ensemble $P = \{P_1, \dots, P_m\}$ de m **processeurs**
 96 **identiques** et pour lesquels les tâches sont **indépendantes**, c'est-à-dire qu'il n'y a
 97 pas de synchronisation entre elles, et **sans suspension**, c'est-à-dire que pendant son
 98 exécution, la tâche n'invoque aucune opération pouvant la conduire à se suspendre et
 99 à libérer ainsi le processeur. Nous supposons aussi que le coût temporel d'une pré-
 100 emption est nul. Les échéances des tâches sont par défaut implicites dans ce qui suit.
 101 De nombreux travaux ont été étendus au cas des échéances contraintes ou arbitraires,
 102 mais pour des questions de place nous n'abordons pas ces distinctions.

103 **2.2. Ordonnancement**

104 L'ordonnancement d'un système consiste à définir une allocation spatiale et tem-
 105 porelle des travaux sur les processeurs de sorte à ce que les contraintes temporelles
 106 soient satisfaites. Un système temps réel est qualifié de **dur** si les conséquences du
 107 non-respect d'échéances sont catastrophiques pour l'application, ou de **souple** si ce
 108 non-respect est acceptable dans certaines limites.

109 Historiquement, deux grandes approches existent pour réaliser un ordonnance-
 110 ment : l'approche **hors-ligne** qui consiste à construire une allocation des travaux avant
 111 le démarrage du système ; l'approche **en-ligne** qui décide de l'ordonnancement pen-
 112 dant l'exécution du système¹. Ces algorithmes reposent en général sur la notion de
 113 priorité. Ces priorités sont soit **fixes pour les tâches**, c'est-à-dire que tous les tra-
 114 vaux d'une tâche ont la même priorité ; soit **fixes pour les travaux**, c'est-à-dire que la
 115 priorité d'un travail ne change pas, mais que deux travaux d'une même tâche peuvent
 116 avoir des priorités différentes ; soit **dynamiques**, c'est-à-dire que la priorité peut évo-
 117 luer pendant la vie d'un travail.

118 Une grande partie de la littérature en ordonnancement temps réel s'attache à prou-
 119 ver le bon comportement d'un algorithme vis-à-vis d'un système. Un système S est
 120 dit **fiablement ordonné** par un algorithme d'ordonnancement A si et seulement
 121 si la séquence produite par A respecte les contraintes temporelles du système. Nous
 122 dirons alors qu'un système S est **ordonnançable** s'il existe un algorithme qui l'ordon-
 123 nance fiablement. Des conditions suffisantes et nécessaires permettent de déterminer,
 124 dans certains cas, si un système est ordonnançable (condition suffisante satisfaite),
 125 ou s'il ne l'est pas (condition nécessaire non satisfaite) pour un algorithme donné.
 126 Ainsi, du fait qu'à tout instant un travail ne peut s'exécuter que sur un seul processeur,
 127 on dispose de la condition nécessaire d'ordonnançabilité d'un système comportant m
 128 processeurs : $u_{sum} \leq m$ et $u_{max} \leq 1$.

129 Un algorithme d'ordonnancement A est dit **optimal** pour une classe de systèmes
 130 S et parmi une classe de politiques d'ordonnancement C ² si et seulement s'il peut
 131 ordonner fiablement tout système de S qui est ordonnançable par une politique
 132 appartenant à C . De nombreux travaux sur l'ordonnancement visent à fournir des
 133 algorithmes optimaux, mais cette optimalité n'est atteinte que dans un cadre précis
 134 (restriction sur le type de tâches et sur l'architecture matérielle) et sous certaines hy-
 135 pothèses (comme coût des préemptions nul ou échéances implicites).

136 Outre la notion d'optimalité, un algorithme A **domine** un algorithme B si tout
 137 système ordonnançable par B l'est par A et s'il existe au moins un système ordonnan-

1. Les travaux présentés ici concernent uniquement des algorithmes d'ordonnancement en-ligne.

2. Nous appelons classe de systèmes, respectivement de politiques d'ordonnancement, un ensemble de systèmes, resp. de politiques d'ordonnancement, affichant les mêmes caractéristiques. Par exemple, la classe des systèmes monoprocesseurs composés de tâches indépendantes, périodiques et à échéances implicites, ou la classe des politiques d'ordonnancement préemptives à priorités fixes sur les tâches. Plusieurs taxonomies pour ces classes existent, l'article de Carpenter *et al.* en est un bon exemple (Carpenter *et al.*, 2004).

138 cable par A et qui ne l'est pas par B . Par contre, s'il existe un système ordonnançable
 139 par A et non ordonnançable par B et un autre système ordonnançable par B et non
 140 ordonnançable par A , on dit que les deux algorithmes A et B sont **non comparables**.

141 L'ordonnancement multiprocesseur peut aussi faire apparaître des **anomalies d'or-**
 142 **donnancement**, c'est-à-dire un changement dans les paramètres du système qui en-
 143 gendre des effets contre-intuitifs (Davis, Burns, 2011). Un exemple est l'augmentation
 144 de la période d'une tâche pour une politique donnée qui devrait intuitivement améliorer
 145 l'ordonnançabilité (puisque diminuant le taux d'utilisation), mais qui en pratique
 146 peut rendre un système non ordonnançable. Andersson étudie en détail cette problé-
 147 matique dans le chapitre 5 de sa thèse (Andersson, 2003).

148 Un algorithme d'ordonnancement est dit **prédictible** si les temps de réponse³ des
 149 travaux ne peuvent pas être augmentés par une réduction de leur durée d'exécution,
 150 avec tous les autres paramètres constants (Ha, Liu, 1994). Cette propriété est impor-
 151 tante car la durée d'exécution des travaux est seulement modélisée par une durée
 152 maximale (WCET). Ainsi l'ordonnançabilité d'un système prédictible est garantie,
 153 si elle est vérifiée sur le modèle avec WCET. Un algorithme d'ordonnancement est dit
 154 **viable** (*sustainable*) pour une classe de systèmes, si et seulement si l'ordonnançabilité
 155 d'un ensemble de tâches conforme à ce modèle reste inchangée après tout change-
 156 ment positif correspondant à une réduction des durées d'exécution, une augmentation
 157 des périodes ou des dates d'inter-arrivée, ou une augmentation des échéances (Burns,
 158 Baruah, 2008).

159 3. Classification des politiques d'ordonnancement multiprocesseur

160 L'éclairage que nous souhaitons donner à cette présentation portant sur les algo-
 161 rithmes d'ordonnancement plutôt que sur leurs propriétés, nous adoptons la classi-
 162 fication traditionnelle des politiques d'ordonnancement selon le critère de la migra-
 163 tion, à savoir le degré selon lequel les travaux d'une même tâche sont susceptibles
 164 de s'exécuter sur des processeurs différents. Deux classes majeures sont alors identi-
 165 fiées : l'**ordonnancement partitionné** pour lequel aucune migration n'est possible et,
 166 à l'opposé, l'**ordonnancement global** pour lequel aucune restriction ne porte sur les
 167 migrations. Entre ces deux extrêmes, des politiques intermédiaires existent que l'on
 168 qualifie d'**hybrides**.

169 Cette classification n'est pas la seule et d'autres critères peuvent être mis en avant
 170 comme : l'évolution des priorités des travaux au cours du temps ; la possibilité ou non
 171 pour les travaux de migrer pendant leur exécution ; la capacité de préempter un travail ;
 172 la manière dont l'ordonnanceur est conduit par le temps ou par les événements ; si les
 173 algorithmes sont conservatifs (*work-conserving* en anglais), c'est-à-dire si un travail
 174 prêt est immédiatement exécuté quand au moins un processeur est libre ; etc.

3. Le temps de réponse d'un travail est la durée séparant le réveil du travail de sa terminaison.

175 **3.1. Ordonnancement par partitionnement**

176 L'ordonnancement par partitionnement consiste à partitionner (au sens mathéma-
 177 tique) l'ensemble des tâches en m parties et à associer chaque ensemble à un proces-
 178 seur. Les tâches associées à un même processeur sont alors ordonnancées par un or-
 179 donnancement monoprocesseur, comme RM (*Rate Monotonic*) ou EDF (*Earliest Dead-*
 180 *line First*) (Liu, Layland, 1973). Ce problème d'affectation des tâches aux processeurs
 181 peut se ramener à un problème de bin packing qui consiste à ranger un ensemble d'élé-
 182 ments caractérisés par leur taille dans des boîtes de capacité limitée. Dans le cas de
 183 l'ordonnancement partitionné, les éléments à ranger sont les tâches et les boîtes sont
 184 les processeurs dont la capacité est déterminée par des conditions suffisantes d'or-
 185 donnabilité monoprocesseur. Ces conditions sont utilisées pour assurer que l'algo-
 186 rithme choisi ordonnance correctement les tâches sur chaque processeur. L'affectation
 187 des tâches se limite alors à trouver une solution telle que le nombre de boîtes soit
 188 inférieur ou égal au nombre de processeurs.

189 De nombreuses méthodes existent pour résoudre le problème de bin packing. Coff-
 190 man et Csirik en font une bonne présentation dans (Coffman, Csirik, 2007). Des algo-
 191 rithmes optimaux sont efficaces pour un nombre d'éléments limités (inférieur à 100),
 192 mais au delà il convient d'utiliser des heuristiques. Les heuristiques les plus usuelles
 193 consistent à trier les tâches suivant un critère (par exemple, par ordre décroissant selon
 194 leur taux d'utilisation), puis pour chaque tâche, à chercher parmi les processeurs or-
 195 donnés le premier qui verra ses conditions d'ordonnabilité préservées (en intégrant
 196 cette tâche aux tâches déjà affectées). L'ordre d'examen des processeurs caractérise
 197 les variantes des heuristiques. Nous décrivons dans les grandes lignes les principales
 198 heuristiques utilisées :

- 199 – First-Fit : l'ordre est arbitraire et l'examen des processeurs commence par le
 200 premier pour chaque tâche ;
- 201 – Next-Fit : l'ordre est arbitraire et l'examen des processeurs commence par le
 202 processeur sur lequel la précédente tâche a été placée ;
- 203 – Best-Fit : les processeurs sont ordonnés selon leur capacité restante croissante
 204 et l'examen des processeurs commence par le premier pour chaque tâche ;
- 205 – Worst-Fit : à l'inverse de Best-Fit, les processeurs sont ordonnés selon leur ca-
 206 pacité restante décroissante. Cet algorithme peut aussi être vu comme un algorithme
 207 d'équilibrage de charge.

208 Ces heuristiques permettent généralement d'obtenir des résultats proches de la
 209 solution optimale (C. Lee, Lee, 1985 ; D.-I. Oh, Bakker, 1998). La combinaison de
 210 ces heuristiques aux politiques d'ordonnancement monoprocesseur, elles-mêmes as-
 211 sociées aux diverses conditions d'ordonnabilité a donné lieu à de très nombreux
 212 algorithmes (Davis, Burns, 2011). Nous ne reviendrons pas dans la suite de l'article
 213 sur les méthodes relevant strictement des approches partitionnées.

214 **3.2. Ordonnancement global**

215 La stratégie d'ordonnancement global consiste à n'utiliser qu'un seul ordonnan-
 216 ceur pour l'ensemble du système ainsi qu'une unique liste de tâches prêtes et à affecter
 217 les processeurs disponibles aux tâches prêtes les plus prioritaires. Les travaux sont
 218 ainsi autorisés à migrer d'un processeur à un autre au cours de leurs exécutions.

219 Les premiers algorithmes proposés dans ce contexte ont été naturellement des ex-
 220 tensions des algorithmes monoprocesseurs comme RM ou EDF. Pour cela, les m (au
 221 plus) travaux ayant la plus grande priorité sont simplement exécutés sur les m pro-
 222 cesseurs disponibles. Cependant, bien que la contrainte sur les migrations soit levée,
 223 il a été montré qu'appliquer RM ou EDF de manière globale, ne permet pas d'obte-
 224 nir des bornes sur les taux d'utilisation limites des systèmes meilleures qu'avec un
 225 ordonnancement par partitionnement (Dhall, Liu, 1978). Par la suite, les politiques
 226 qui sont généralisées au cas multiprocesseur par une approche globale seront notées
 227 « G- » suivi du nom de la politique.

228 Dans leur paragraphe *Why Greedy Schedulers Fail*, Levin *et al.* (2010) expliquent
 229 pourquoi les approches généralisant simplement les ordonnanceurs monoprocesseurs
 230 ne permettent pas d'atteindre l'optimalité dans le cas multiprocesseur. Pour cela, ils
 231 montrent qu'il est nécessaire que des décisions d'ordonnancement soient prises en de-
 232 hors des événements « habituels » donnant lieu à un réordonnancement dans le cas
 233 monoprocesseur, c'est-à-dire, autres qu'une fin de travail, un réveil de travail ou une
 234 laxité nulle. Nelissen *et al.* ont montré à travers l'algorithme U-EDF une autre façon
 235 de généraliser ces algorithmes afin de permettre une utilisation optimale des proces-
 236 seurs, mais ceci passe par la création de budgets virtuels, sources de nouveaux évène-
 237 ments (Nelissen *et al.*, 2012).

238 L'origine de ces avancées en matière d'ordonnancement global sont les méthodes
 239 dites équitables qui ont été présentées au début des années 1990. À la différence des
 240 algorithmes classiques d'ordonnancement, ces algorithmes imposent explicitement
 241 l'exécution des tâches à un rythme régulier (Baruah *et al.*, 1996). L'objectif d'un tel
 242 algorithme est de se rapprocher d'un ordonnancement fluide, ou encore idéal ou équi-
 243 table. Un ordonnancement est dit **équitable** lorsque chaque tâche τ_i reçoit exactement
 244 $u_i \cdot t$ unités de temps processeur dans l'intervalle $[0, t[$. Parmi les approches équitables,
 245 nous retrouvons notamment les familles d'algorithmes PFair et DP-Fair. Dans le cas
 246 d'une approche PFair, le temps est divisé en intervalles de temps uniformes appelés
 247 *slots*, faisant chacun l'objet d'un ordonnancement des travaux. Zhu *et al.* ont montré
 248 ensuite qu'il n'est pas nécessaire de se rapprocher autant de l'exécution fluide des
 249 travaux (Zhu *et al.*, 2003) et qu'il est suffisant qu'à chaque échéance, aucune tâche
 250 ne soit en retard par rapport à son exécution fluide. Ces méthodes ont permis d'at-
 251 teindre l'optimalité vis-à-vis de la limite d'utilisation ($u_{sum} \leq m$ et $u_{max} \leq 1$), mais
 252 introduisent de très nombreuses préemptions et migrations les rendant inefficaces en
 253 pratique.

254 3.3. Approches hybrides

255 L'impossibilité d'effectuer une migration d'un processeur à un autre dans le cas
 256 de l'ordonnancement par partitionnement limite fortement l'ordonnançabilité des sys-
 257 tèmes. À l'opposé, certains ordonnanceurs globaux offrent une totale liberté dans la
 258 migration des tâches ce qui permet d'atteindre l'optimalité. Cependant, cette flexibi-
 259 lité a un coût temporel lors de l'exécution et il est souhaitable de limiter le nombre
 260 de migrations de la même manière que le nombre de préemptions (Devi, Anderson,
 261 2005). Pour pallier de tels défauts, d'autres approches intermédiaires ont été propo-
 262 sées donnant lieu à de nombreuses politiques que l'on peut classer en deux grandes
 263 catégories : l'ordonnancement **semi-partitionné** et le **clustering**. Toutefois, des ap-
 264 proches récentes telles que RUN (Regnier *et al.*, 2011) ou QPS (Massa *et al.*, 2014)
 265 ne peuvent pas être aussi simplement catégorisées.

266 3.3.1. Ordonnancement semi-partitionné

267 L'ordonnancement semi-partitionné se base pour l'essentiel sur une approche par-
 268 tionnée mais, lorsque le système n'est pas complètement partitionnable, certaines
 269 tâches sont autorisées à migrer. Il s'agit donc là d'introduire un peu de flexibilité dans
 270 les algorithmes partitionnés. On distingue deux sous-familles parmi ces algorithmes,
 271 celles où la migration n'est possible qu'entre les travaux (on parle alors de *restric-*
 272 *ted migration*) et celle où la migration est possible pendant l'exécution d'un travail.
 273 Dans ce dernier cas, on parle de *portioned scheduling*, c'est-à-dire qu'une tâche est
 274 découpée en portion fixes, une portion est attribuée à un processeur et les portions
 275 sont exécutées à des dates garantissant l'absence de parallélisme en introduisant des
 276 échéances virtuelles fonction de l'échéance de la tâche. En section 6.1, nous présen-
 277 terons quelques politiques semi-partitionnées et verrons que la frontière avec l'ordon-
 278 nancement global est parfois mince.

279 Les principaux algorithmes semi-partitionnés sont implémentés dans LITMUS^{RT}
 280 (J. Calandrino *et al.*, 2006) et offrent un bon compromis en pratique entre le nombre
 281 de préemptions-migrations et l'ordonnançabilité (Bastoni *et al.*, 2011), comparé aux
 282 autres politiques.

283 3.3.2. Clustering

284 L'ordonnancement par *clustering* se situe aussi entre l'ordonnancement partitionné
 285 et l'ordonnancement global. Au lieu de limiter les tâches pouvant migrer, ce sont les
 286 processeurs sur lesquels les tâches peuvent migrer qui vont être restreints. Cet en-
 287 semble de processeurs est alors appelé *cluster*. L'approche par clustering peut être
 288 divisée en deux catégories : le *clustering* physique et le *clustering* virtuel.

289 Dans le cas d'un *clustering* physique, les processeurs sont associés aux *clusters*
 290 de façon définitive (J. M. Calandrino *et al.*, 2007) et ce conformément à l'architecture
 291 matérielle, c'est-à-dire que les *clusters* seront constitués des processeurs partageant les
 292 mêmes niveaux de mémoire. Seul un ordonnancement des tâches au sein de chaque
 293 *cluster* est nécessaire. L'intérêt de cette approche est multiple : cela permet de limiter

294 la migration de certaines tâches sur certains groupes de processeurs (par exemple sur
 295 des cœurs partageant des caches), mais aussi d'augmenter en pratique le nombre de
 296 systèmes ordonnançables par rapport à une approche partitionnée tout en limitant le
 297 nombre de migrations.

298 Le *clustering* virtuel offre une approche plus générale en permettant de changer dy-
 299 namiquement l'association des processeurs aux *clusters* (Shin *et al.*, 2008). En contre-
 300 partie, cela nécessite un algorithme d'affectation des clusters sur les processeurs.

301 Le *clustering* ne sera pas étudié dans cet article, en revanche, certaines politiques
 302 utilisent des notions proches. Par exemple EKG (cf. section 6.1) effectue des regrou-
 303 pements de processeurs tels qu'aucune tâche ne peut migrer entre deux groupes diffé-
 304 rents de processeurs. Les politiques RUN (section 6.2) ou encore NPS-F empruntent
 305 elles aussi des notions similaires.

306 Dans le tableau 1, nous présentons par catégorie et accompagné de ses références
 307 un récapitulatif des politiques d'ordonnancement, hors approches partitionnées et par
 308 *clustering*, qui ont été proposées en matière d'ordonnancement temps réel multipro-
 309 cesseur. Notre intention ici, sans prétendre à l'exhaustivité de ce large éventail (près
 310 d'une cinquantaine de politiques recensées), est d'offrir un point d'entrée bibliogra-
 311 phique au lecteur désireux d'approfondir le sujet.

312 4. Ordonnancement global par généralisation des algorithmes monoprocesseurs

313 Puisqu'une simple généralisation des algorithmes monoprocesseurs ne permet pas
 314 d'utiliser de manière optimale les processeurs, de nouvelles politiques ont été propo-
 315 sées pour en améliorer l'ordonnançabilité. Cette partie propose un survol des princi-
 316 paux algorithmes existants.

317 4.1. Algorithmes *RM-US*[ξ], *EDF-US*[ξ], *EDF*^(k) et *fpEDF*

318 Les travaux liés à l'étude de l'ordonnançabilité des systèmes ordonnancés par RM
 319 et EDF ont montré que la valeur u_{max} avait une forte importance sur les critères
 320 d'ordonnançabilité (Andersson *et al.*, 2001 ; Baruah, Goossens, 2003 ; Bertogna *et al.*,
 321 2005 ; Srinivasan, Baruah, 2002 ; Goossens *et al.*, 2003). Ainsi, les algorithmes RM-
 322 US[ξ], EDF-US[ξ], EDF^(k) et fpEDF ont été élaborés dans l'objectif d'améliorer les
 323 conditions suffisantes d'ordonnançabilité en dépassant la limite liée à u_{max} .

324 Les politiques RM-US[ξ] (Andersson *et al.*, 2001) et EDF-US[ξ] (Srinivasan, Ba-
 325 ruah, 2002) dérivent respectivement des algorithmes G-RM et G-EDF. Ces nouvelles
 326 politiques introduisent un seuil sur le taux d'utilisation au delà duquel les tâches
 327 sont considérées comme étant toujours plus prioritaires (avec résolution arbitraire des
 328 conflits entre elles) que les autres tâches, l'affectation des priorités restant classique
 329 pour les autres tâches. Des variantes de ces algoihmes existent, par exemple, DM-
 330 DS[λ_{th}] (Bertogna *et al.*, 2005) qui fixe le seuil à partir de la densité d'une tâche,
 331 c'est-à-dire le rapport entre sa durée d'exécution et son échéance, ou encore SM-

Tableau 1. Récapitulatif d'ordonnanceurs multiprocesseurs

Nom	Références	Page ^a
Ordonnanceurs globaux généralisant les politiques monoprocesseurs		
RM-US (Rate Monotonic with Utilization Separation)	(Andersson <i>et al.</i> , 2001)	p. 9
EDF-US (Earliest Deadline First with Utilization Separation)	(Srinivasan, Baruah, 2002)	p. 9
SM-US (Slack Monotonic with Utilization Separation)	(Andersson, 2008)	p. 11
DM-DS (Deadline Monotonic with Density Separation)	(Bertogna <i>et al.</i> , 2005)	p. 9
PriD/EDF ^(k) (Priority-Driven / Earliest Deadline First ^(k))	(Goossens <i>et al.</i> , 2003)	p. 11
fpEDF (fixed-priority Earliest Deadline First)	(Baruah, 2004)	p. 11
Tp-TkC (Fixed Priority with adaptiveTkC)	(Andersson, Jonsson, 2000)	p. 11
GFL (Global-Fair Lateness)	(Erickson, Anderson, 2012)	p. 11
EDZL (Earliest Deadline Zero Laxity)	(S. Lee, 1994)	p. 11
EDCL (Earliest Deadline Critical Laxity)	(Kato, Yamasaki, 2008a)	p. 12
FPZL, FPCL, FPSL (Fixed Priority Zero Laxity)	(Davis, Kato, 2012)	p. 11
GLLF (Global Least Laxity First)	(Mok, 1983)	p. 12
GMLLF (Global Modified Least Laxity First)	(S.-H. Oh, Yang, 1998)	p. 12
U-EDF (Unfair-EDF)	(Nelissen <i>et al.</i> , 2012)	p. 13
Ordonnanceurs globaux de type PFair et ERFair		
EPDF (Earliest Pseudo-Deadline First)	(Anderson, Srinivasan, 2000b)	p. 15
PF (Proportionate Fair)	(Baruah <i>et al.</i> , 1996)	p. 15
PD (Pseudo-Deadline)	(Baruah <i>et al.</i> , 1995)	p. 15
PD ² (Pseudo-Deadline ²)	(Anderson, Srinivasan, 1999)	p. 15
ER-PD ² (Early-Release Fair Pseudo Deadline ²)	(Anderson, Srinivasan, 2000a)	p. 16
PL (Pseudo-Laxity)	(Kim, Cho, 2011)	p. 16
Ordonnanceurs globaux de type DPFair et BFair		
LLREF (Largest Local Remaining execution time First)	(Cho <i>et al.</i> , 2006)	p. 16
LRE-TL (Local Remaining Execution-Time and Local plane)	(Funk, Nanadur, 2009)	p. 17
DP-WRAP (Deadline Partitioning-Wrap)	(Levin <i>et al.</i> , 2010)	p. 17
BF (Boundary Fair)	(Zhu <i>et al.</i> , 2003)	p. 17
BF ² (Boundary Fair ²)	(Nelissen <i>et al.</i> , 2014)	p. 17
NVNLF (No Virtual Nodal Laxity First)	(Funaoka <i>et al.</i> , 2008)	p. 17
SA (Scheduling Algorithm)	(Khemka, Shyamasundar, 1997)	
Ordonnanceurs semi-partitionnés		
EDF-fm (Earliest Deadline First-fixed or migrating)	(Anderson <i>et al.</i> , 2005)	p. 18
EKG (EDF with task splitting and k processors in a Group)	(Andersson, Tovar, 2006)	p. 18
EDHS (Earliest Deadline and Highest-priority Split)	(Kato, Yamasaki, 2008d)	p. 19
EDF-WM (EDF Window constrained Migration)	(Kato <i>et al.</i> , 2009)	p. 19
EDF-C=D (Earliest Deadline First with C=D)	(Burns <i>et al.</i> , 2012)	p. 20
EDF-RRJM (EDF-Round Robin Job Migration)	(George <i>et al.</i> , 2011)	p. 20
Ehd2-SIP ou EDDHP	(Kato, Yamasaki, 2007)	
EDDP (Earliest Deadline Deferrable Portion)	(Kato, Yamasaki, 2008b)	
RMDP (Rate Monotonic Deferrable Portion)	(Kato, Yamasaki, 2008c)	
DMPM (Deadline Monotonic with Priority Migration)	(Kato, Yamasaki, 2009)	
PDMS_HPTS_DS	(Lakshmanan <i>et al.</i> , 2009)	
SPA2 (Semi-Partitioned scheduling Algorithm 2)	(Guan <i>et al.</i> , 2010)	
HSP (Harmonic Semi-Partitioned)	(Fan, Quan, 2012)	
EDF-BR (EDF with Bandwidth Reservation)	(Massa, Lima, 2010)	
EDF-os (EDF-based optimal semi-partitioned scheduling)	(Anderson <i>et al.</i> , 2014)	
NPS-F (Notional Processor Scheduling-First-Fit bin-packing)	(Bletsas, Andersson, 2009)	
Autres		
RUN (Reduction to UNiprocessor)	(Regnier <i>et al.</i> , 2011)	p. 20
SPRINT (SPoradic Run for INdependent Tasks)	(Baldovin <i>et al.</i> , 2014)	p. 20
Carousel-EDF	(Sousa <i>et al.</i> , 2013)	
QPS (Quasi-Partitioning Scheduler)	(Massa <i>et al.</i> , 2014)	

a. Le numéro correspond à la page où sont présentées les politiques d'ordonnancement citées dans le présent article.

332 US (Andersson *et al.*, 2008) qui attribue les priorités statiques des tâches suivant un
333 critère basé sur la différence entre la période et la durée d'exécution d'une tâche.

334 L'algorithme EDF^(k) (Goossens *et al.*, 2003) a une approche similaire et propose
335 de fixer une priorité maximale aux $k - 1$ tâches ayant les plus grands taux d'utilisation
336 du système. Il est possible de choisir la valeur k telle que EDF^(k) se comporte comme
337 EDF-US[ξ]. L'algorithme PriD (Goossens *et al.*, 2003) complète EDF^(k) en détermi-
338 nant automatiquement la valeur du paramètre k . Pour finir sur cet aspect, l'algorithme
339 fpEDF (Baruah, 2004) donne une priorité maximale aux tâches ayant un taux d'utili-
340 sation supérieur à $1/2$ en se limitant aux $m - 1$ premières tâches par ordre décroissant
341 de taux d'utilisation.

342 Enfin, il est à noter l'existence de FP-TkC (Andersson, Jonsson, 2000) qui se base
343 sur un algorithme à priorités fixes, mais qui affecte les priorités aux tâches selon les
344 valeurs $T_i - kC_i$, avec k un facteur fixé pour le système.

345 4.2. Algorithme Global-Fair Lateness

346 L'algorithme G-FL (Erickson, Anderson, 2012) développé pour des systèmes temps
347 réel souples, propose une modification mineure de G-EDF afin de réduire le retard
348 maximum des travaux. Le retard d'un travail est défini par la différence entre la date
349 de fin du travail et son échéance. La priorité d'un travail selon la politique EDF cor-
350 respond à son échéance absolue ($d_{i,j}$) alors que G-FL utilise une priorité égale à
351 $d_{i,j} - \frac{m-1}{m}C_i$.

352 Les auteurs affirment que pour des systèmes temps réel souples, l'algorithme G-
353 FL devrait remplacer G-EDF. En effet, G-FL peut facilement être substitué à G-EDF
354 tout en réduisant le dépassement maximum d'échéance. De plus, les résultats expé-
355 rimentaux que nous avons menés montrent également un nombre plus important de
356 systèmes exécutés sans dépassement d'échéance (Chéramy, 2014).

357 4.3. Algorithmes d'ordonnancement à laxité nulle (Zero Laxity)

358 Les politiques RMZL, FPZL (Davis, Kato, 2012) et EDZL (S. Lee, 1994) sont
359 des algorithmes à priorités dynamiques, basés sur les algorithmes respectifs G-RM,
360 G-FP (*Global-Fixed task Priority*) et G-EDF, qui intègrent la notion de laxité. Ces
361 algorithmes *Zero Laxity* diffèrent des algorithmes initiaux en attribuant une priorité
362 maximale aux travaux de laxité nulle afin d'éviter un non-respect d'échéance. Dans
363 le cas contraire, la priorité est celle définie par l'algorithme initial. En conséquence,
364 lorsqu'un système est ordonnançable sans prise en compte de la laxité, le comporte-
365 ment fonctionnel de ces variantes est identique. La figure 1 montre l'existence d'un
366 système ordonnançable par EDZL qui ne l'est pas par G-EDF.

367 Park *et al.* (Park *et al.*, 2005) montrent que EDZL domine strictement G-EDF et par
368 simulation que EDZL permet d'ordonnancer un nombre plus important de systèmes

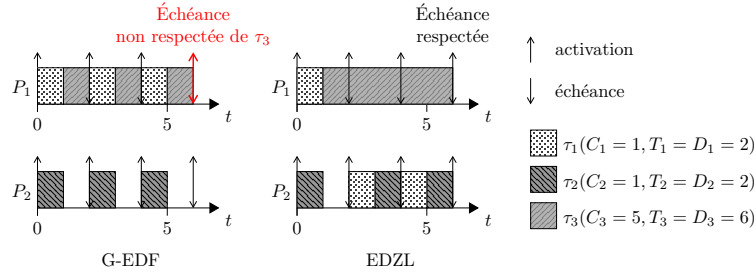


Figure 1. Exemple où EDZL permet un ordonnancement contrairement à G-EDF

369 que les variantes EDF-US et fpEDF. De plus, le nombre de préemptions reste similaire
 370 à G-EDF.

371 L'algorithme EDCL (Kato, Yamasaki, 2008a) (*Earliest Deadline Critical Laxity*)
 372 est une variante de EDZL qui ne change la priorité des travaux ayant atteint une cer-
 373 taine laxité qu'au moment de l'activation et la terminaison de travaux. Ceci permet de
 374 réduire le nombre de préemptions en l'absence de nouveaux événements. Cependant,
 375 le nombre de systèmes ordonnancables est plus faible qu'avec EDZL.

376 4.4. Algorithme Global-Least Laxity First

377 L'algorithme LLF est une politique d'ordonnancement monoprocesseur qui fixe la
 378 priorité des travaux en fonction de leur laxité. Les travaux disposant de la laxité la
 379 plus faible sont rendus prioritaires. L'inconvénient principal de cette approche est la
 380 nécessité de réévaluer à de nombreuses reprises les priorités des travaux. Aussi l'al-
 381 gorithme *Modified-Least-Laxity-First* (MLLF) tente-t-il de résoudre ce problème en
 382 exécutant consécutivement les travaux dont la laxité est identique, permettant ce qu'on
 383 appelle une inversion de laxité (S.-H. Oh, Yang, 1998). Ceci évite ainsi le phénomène
 384 de *task thrashing* où la priorité des tâches s'inverse très rapidement, provoquant de
 385 nombreuses préemptions.

386 L'algorithme G-LLF (Mok, 1983) adapte LLF à une architecture multiprocesseur
 387 en exécutant les m travaux les plus prioritaires et G-MLLF (S.-H. Oh, Yang, 1998)
 388 généralise MLLF avec des modifications mineures (que nous ne pouvons présenter ici
 389 faute de place).

390 Tout comme EDZL, G-LLF et sa variante G-MLLF rendent prioritaires les tâches
 391 dont la laxité devient nulle ce qui leur permet d'éviter un certain nombre d'échecs à
 392 l'ordonnancement. Cette stratégie semble donner de bons résultats et des travaux ont
 393 montré que les tests d'ordonnancabilité pour G-LLF donnent un plus grand ensemble
 394 de systèmes ordonnancables que ceux de EDZL (J. Lee *et al.*, 2010).

395 **4.5. Algorithme U-EDF**

396 L'algorithme U-EDF est une politique multiprocesseur optimale pour des tâches
 397 sporadiques à échéances implicites (Nelissen *et al.*, 2012). Les auteurs indiquent que
 398 U-EDF n'est pas basé sur la notion d'équité (*fairness*) (voir section 5), cependant du
 399 temps est réservé sur les processeurs en fonction des taux d'utilisation des travaux.
 400 Des évènements qui correspondent à des fins de budgets virtuels sont introduits pour
 401 permettre d'atteindre l'optimalité. Notons que l'apparition de cet algorithme s'est faite
 402 après les travaux sur les algorithmes équitables et les approches hybrides dans la pers-
 403 pective de réduire le nombre de préemptions et migrations par rapport à ces derniers.

404 Les auteurs présentent U-EDF comme étant une généralisation de EDF « hori-
 405 zontale » contrairement à G-EDF qu'ils qualifient de « verticale ». Pour illustrer cette
 406 notion, les auteurs proposent l'exemple d'un système composé de deux processeurs et
 407 trois travaux $J_1 = (2, 6)$, $J_2 = (3, 6)$ et $J_3 = (9, 10)$ avec (c, d) qui représente une
 408 exécution de c unités de temps et une échéance à l'instant d . Le chronogramme de
 409 gauche de la figure 2 illustre la séquence obtenue pour G-EDF où l'on peut constater
 410 que l'échéance du travail J_3 n'est pas respectée. Cet échec est causé par le fait que G-
 411 EDF favorise l'exécution au plus tôt des travaux sans anticiper l'impact sur les futures
 412 exécutions.

413 Le second chronogramme de la figure 2 montre une possible généralisation « hori-
 414 zontale » de EDF. Elle consiste à ordonnancer les travaux avec les plus grandes
 415 priorités (c'est-à-dire avec l'échéance la plus proche) de manière séquentielle sur un
 416 premier processeur et à ne commencer l'allocation sur un autre processeur que si le
 417 premier processeur ne peut pas assurer l'ordonnancement de tous les travaux. Dans
 418 l'exemple présenté, le travail J_3 sera exécuté sur le premier et le deuxième proces-
 419 seur car le premier processeur ne permet pas de traiter l'ensemble des travaux. Les
 420 auteurs qualifient cette approche de « horizontale » car l'algorithme va d'abord tenter
 421 de « remplir » un processeur au maximum avant d'allouer un travail à un nouveau pro-
 422 cesseur. La construction de l'ordonnancement telle qu'elle vient d'être présentée est
 423 faite hors-ligne, alors que l'algorithme U-EDF construit une telle séquence grâce à un
 424 algorithme en-ligne fonctionnant en deux étapes :

425 1. Lorsqu'un nouveau travail est réveillé, le budget des travaux prêts est calculé
 426 en « remplissant » les processeurs en fonction de leur taux d'utilisation, tout en provi-
 427 sionnant du temps sur chaque processeur pour les tâches inactives.

428 2. La seconde phase consiste à ordonnancer les travaux selon l'algorithme EDF-D
 429 en fonction des budgets calculés dans la première phase.

430 L'algorithme EDF-D est une variante de EDF qui ordonnance les tâches éligibles
 431 sur les processeurs. Une tâche prête est dite *éligible* sur un processeur si elle a du
 432 temps réservé sur ce processeur et si elle n'a pas été choisie pour s'exécuter sur un
 433 autre processeur d'indice inférieur (les processeurs étant homogènes, les processeurs
 434 sont simplement ordonnés par un indice). L'algorithme EDF-D sélectionne les tâches
 435 à exécuter pour chaque processeur en choisissant la tâche éligible avec la plus petite
 436 échéance.

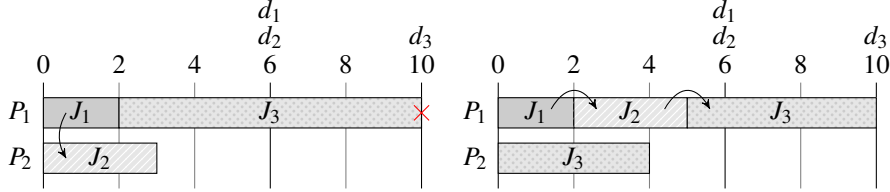


Figure 2. Généralisation « verticale » et « horizontale »

437 La difficulté de cette approche réside dans la détermination des budgets virtuels à
 438 réserver sur les processeurs lors du réveil d'un nouveau travail. Hélas, l'algorithme ne
 439 pouvant être expliqué simplement et de manière concise, un lecteur intéressé par les
 440 détails est invité à étudier la thèse de G. Nelissen (Nelissen, 2012).

441 5. Ordonnancement global dit équitable

442 5.1. Contrainte d'équité

443 L'expression de la contrainte d'équité conduit à discrétiser le temps en intervalles
 444 uniformes appelés des *slots*. Le slot $t \in \mathbb{N}$ correspond à l'intervalle de temps $[t, t +$
 445 $1[$. La séquence d'ordonnancement est modélisée par une fonction binaire $S : \tau \times$
 446 $\mathbb{N} \rightarrow \{0, 1\}$ qui renvoie 1 lorsque la tâche τ_i est exécutée sur le slot t et 0 sinon. La
 447 fonction d'erreur d'exécution d'une tâche τ_i à l'instant t , définie par $lag(\tau_i, t) = u_i t -$
 448 $\sum_{l=0}^{t-1} S(\tau_i, l)$, mesure l'écart entre l'exécution idéale et l'exécution réelle résultant de
 449 l'ordonnancement.

450 Un algorithme est dit *PFair* lorsque, à chaque instant t , l'erreur d'exécution est
 451 strictement inférieure à un quantum de temps, c'est-à-dire quand $\forall \tau_i \in \tau, |lag(\tau_i, t)| <$
 452 1 , ce qui implique que $\sum_{l=0}^{t-1} S(\tau_i, l) = \lfloor u_i t \rfloor$ ou $\lceil u_i t \rceil$. La figure 3 montre l'exécution
 453 d'une tâche dans le respect de cette contrainte.

454 5.2. Algorithmes d'ordonnancement *PFair*

455 La construction d'un ordonnancement *PFair* conduit à diviser chaque tâche en
 456 sous-tâches de durée d'exécution d'un quantum. Une tâche τ_i est ainsi divisée en
 457 une séquence infinie de sous-tâches τ_i^j avec τ_i^1 la première sous-tâche. Afin que la
 458 contrainte *PFair* soit respectée, il faut que chaque sous-tâche τ_i^j s'exécute dans une
 459 fenêtre temporelle $w(\tau_i^j) = [r(\tau_i^j), d(\tau_i^j)]$ avec la date de pseudo-réveil $r(\tau_i^j) =$
 460 $\lfloor (j-1)/u_i \rfloor$ et la date de pseudo-échéance $d(\tau_i^j) = \lceil j/u_i \rceil$, tout en respectant les précédé-
 461 dences entre elles.

462 La figure 4 montre, pour l'exemple précédent, les fenêtres d'exécution des diffé-
 463 rentes sous-tâches. On constate que l'ordonnancement de la tâche respecte bien l'exé-
 464 cution successive de chaque sous-tâche au sein de sa fenêtre.

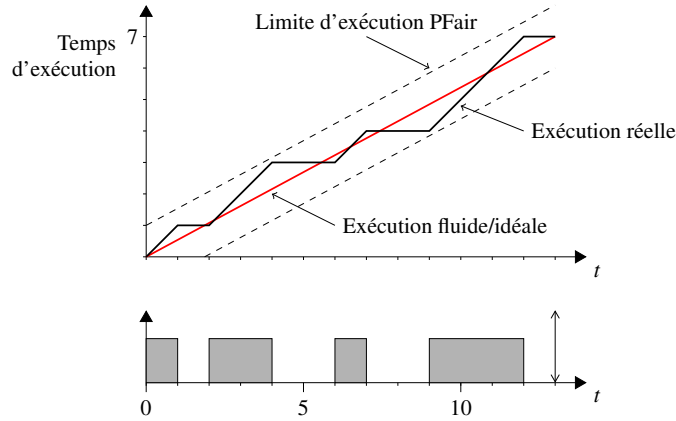


Figure 3. Exécution fluide d'une tâche τ_i ($C_i = 7, T_i = 13$)

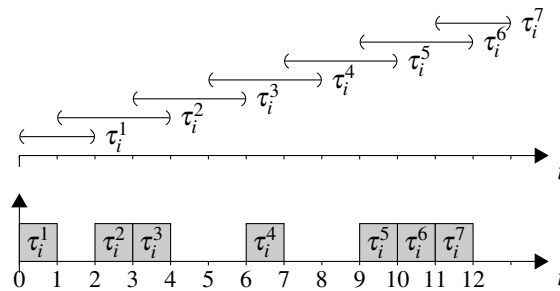


Figure 4. Fenêtres d'exécution des sous-tâches de τ_i ($C_i = 7, T_i = 13$)

465 Bien qu'en théorie un tel ordonnancement soit optimal, il convient de nuancer ce
 466 résultat en pratique. En effet, les surcoûts d'exécution tels que les prises de décision
 467 à chaque quantum, ou encore les nombreuses préemptions et possibles migrations, ne
 468 sont absolument pas prises en compte (Holman, Anderson, 2005a ; Chéramy, 2014).
 469 De plus, les processeurs doivent être parfaitement synchronisés entre eux pour respec-
 470 ter les précédences entre les sous-tâches d'une même tâche. Les travaux de Holman
 471 et Anderson (2005b) proposent une adaptation de PFair pour tenir compte de ce pro-
 472 blème de synchronisation.

473 Plusieurs algorithmes de type PFair existent : EPDF (Anderson, Srinivasan, 2000b)
 474 n'est optimal que pour deux processeurs alors que les algorithmes PF (Baruah *et al.*,
 475 1996), PD (Baruah *et al.*, 1996) et PD² (Anderson, Srinivasan, 1999) qui améliorent
 476 EPDF par l'évaluation et la comparaison de critères supplémentaires sont optimaux.

477 5.3. Variantes conservatives ER-Fair

478 Un ordonnanceur PFair n'est pas conservatif dans la mesure où il peut être amené
479 à ne pas exécuter des tâches pour éviter qu'elles ne prennent trop d'avance. Les po-
480 litiques dites ER-Fair (*Early-Release Fair*) proposent d'alléger la contrainte PFair en
481 ramenant la condition à $lag(\tau_i, t) < 1$. Cette approche permet d'obtenir des temps de
482 réponse moyens des travaux plus courts et une mise en œuvre simplifiée.

483 La politique ER-PD² (Anderson, Srinivasan, 2000a) est une adaptation ER-Fair
484 de l'algorithme PD². Plus récemment, Kim et Cho ont proposé la politique PL (Kim,
485 Cho, 2011) de type ER-Fair qui intègre la laxité comme critère pour départager les
486 sous-tâches.

487 5.4. Principes de l'ordonnancement global DP-Fair

488 Les techniques DP-Fair, pour « Deadline Partitioning Fair » reposent sur le concept
489 de *Deadline Partitioning* qui consiste à découper le temps en intervalles définis par
490 l'ensemble des échéances des tâches, associé à celui d'équité (Levin *et al.*, 2010). À la
491 fin de chaque intervalle, DP-Fair impose donc aux tâches de ne pas avoir de retard par
492 rapport à leur exécution fluide, soit à chaque date d'échéance $d_{i,j}$, $lag(\tau_k, d_{i,j}) = 0$
493 pour toute tâche τ_k de τ .

494 L'étape initiale est donc le découpage du temps en intervalles définis par les dates
495 d'échéance de l'ensemble des tâches. Plus formellement, en s'inspirant de la notation
496 utilisée par Levin *et al.*, on note $t_0 = 0$ et t_1, t_2, \dots les dates des différentes échéances
497 par ordre chronologique ($t_j < t_{j+1}$). Le $j^{\text{ème}}$ intervalle, noté σ_j , correspond donc à
498 l'intervalle $[t_{j-1}, t_j[$, et $L_j = t_j - t_{j-1}$ à sa longueur. Par la suite, i correspond à
499 l'indice de la tâche τ_i et j à l'indice de l'intervalle σ_j .

500 Pour chaque intervalle σ_j , l'ensemble des m processeurs met à disposition $m \cdot L_j$
501 unités temporelles d'exécution. Il convient alors de définir :

- 502 – la dotation temporelle pour chaque tâche τ_i , c'est-à-dire le nombre d'unités tem-
503 porelles qui seront allouées à son exécution dans σ_j ;
- 504 – la distribution temporelle, c'est-à-dire la manière dont les dotations temporelles
505 locales des tâches seront réparties sur les m processeurs pendant σ_j .

506 5.5. Algorithmes DP-Fair et variantes

507 Il existe plusieurs politiques de type DP-Fair, les différences se jouent principa-
508 lement sur la façon dont la dotation temporelle et la distribution temporelle sont dé-
509 cidées. Citons en particulier l'algorithme LLREF (Cho *et al.*, 2006), l'une des pre-
510 mières politiques DP-Fair. Cet algorithme affecte une durée d'exécution locale l_i^j à
511 chaque tâche τ_i sur l'intervalle σ_j telle que $l_i^j = u_i \cdot L_j$, l_i^j pouvant ainsi être réel.
512 La distribution temporelle des dotations sur chaque intervalle se base, quant à elle,
513 sur l'abstraction du TL-Plane (*Time and Local execution time domain Plane*) (Cho

514 *et al.*, 2006) qui pour chaque intervalle σ_j permet de représenter sous la forme d'un
 515 triangle rectangle isocèle, le domaine temporel au sein duquel l'ordonnancement des
 516 tâches (vu comme des trajectoires de jetons) doit s'inscrire pour être qualifié DP-Fair.
 517 La figure 5 illustre son fonctionnement dans l'exemple de l'exécution de 3 tâches sur
 518 2 processeurs dans l'intervalle σ_j . Au début de l'intervalle, les travaux disposant de
 519 la plus grande durée d'exécution locale l_i^j sont choisis et leurs jetons se déplacent
 520 parallèlement à l'hypoténuse du TL-Plane tandis que celui de τ_1 progresse horizon-
 521 talement. Lorsque le jeton de τ_1 atteint cette hypoténuse, soit quand sa laxité devient
 522 nulle, un événement de type C est généré et l'algorithme choisit à nouveau les deux
 523 tâches ayant la plus grande durée d'exécution locale restante. Lorsque le jeton de τ_2
 524 atteint l'axe horizontal du temps, c'est-à-dire que τ_2 a consommé sa durée d'exé-
 525 cution locale, un événement de type B est généré et l'algorithme choisit les deux tâches
 526 restantes. Ainsi, lors d'évènements B ou C, l'algorithme choisit à nouveau les travaux
 527 à exécuter.

528 La politique LRE-TL (Funk, Nanadur, 2009) se base sur LLREF mais réduit de
 529 manière significative le nombre de préemptions et migrations en ne choisissant pas les
 530 tâches ayant la plus grande durée d'exécution locale restante. De plus, avec quelques
 531 adaptations, LRE-TL supporte également les tâches sporadiques.

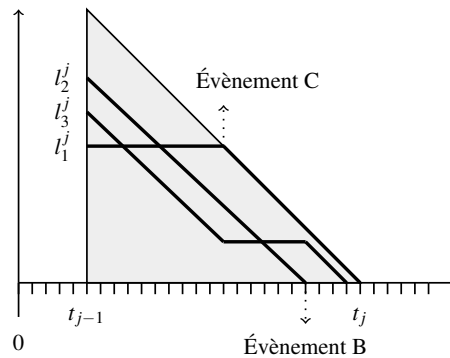


Figure 5. Exécution au sein d'un TL-plane sur l'intervalle σ_j des tâches τ_1 , τ_2 et τ_3 sur deux processeurs

532 Levin *et al.* ont présenté l'algorithme DP-WRAP (Levin *et al.*, 2010) qui a pour
 533 particularité d'utiliser l'algorithme de McNaughton pour effectuer la distribution tem-
 534 porelle. Cet algorithme est également capable de traiter des tâches sporadiques.

535 La technique BFair, pour *Boundary Fair*, est très similaire à la technique DP-Fair,
 536 mais réalise une dotation temporelle en nombres entiers. Dans cette catégorie, l'algo-
 537 rithme BF (Zhu *et al.*, 2003) a été proposé ainsi que BF² (Nelissen *et al.*, 2014) pour
 538 l'ordonnancement de tâches sporadiques.

539 Enfin, NVNLF (Funaoka *et al.*, 2008) est une variante *work-conserving* de LLREF
 540 qui permet de réduire de façon importante le nombre de préemptions et migrations
 541 tout en réduisant aussi les temps de réponse. L'algorithme NVNLF se base sur la

542 notion de TL-plane étendue afin de prendre en compte les ressources processeurs non
 543 utilisées. NVNLF alloue dans un premier temps le même budget que LLREF, puis
 544 distribue l'éventuel temps processeur restant aux tâches (non entièrement déjà dotées).
 545 Une tâche pourra donc continuer son exécution au-delà de son exécution fluide sans
 546 pénaliser les autres tâches.

547 6. Approches hybrides

548 6.1. Ordonnancement semi-partitionné

549 Les politiques d'ordonnancement semi-partitionné visent à améliorer les limites
 550 d'utilisation des algorithmes basés sur le partitionnement de tâches dans le cas où
 551 le système n'est pas partitionnable. Les premières politiques de ce genre sont EDF-
 552 fm (Anderson *et al.*, 2005) publié en 2005 par Anderson *et al.* et EKG (Andersson,
 553 Tovar, 2006) publié en 2006 par Andersson *et al.*. Dans le cas de ces deux ordonnan-
 554 ceurs, les tâches sont affectées aux processeurs à l'aide d'une variante de l'algorithme
 555 Next-Fit : les tâches sont affectées au processeur courant et lorsque celui-ci n'a pas
 556 assez de place, la tâche est découpée en deux parties. Ainsi, on obtient un partitionne-
 557 ment avec au maximum $m - 1$ tâches qui seront amenées à migrer d'un processeur à
 558 l'autre.

559 Afin de réduire le nombre de tâches migrantes, EKG propose de faire des regroupe-
 560 ments de processeurs de taille K et d'interdire les migrations en dehors de ces groupes.
 561 Les tâches dont le taux d'utilisation dépasse $\frac{K}{K+1}$ sont exécutées sur des processeurs
 562 dédiés. Un exemple est donné par la figure 6.

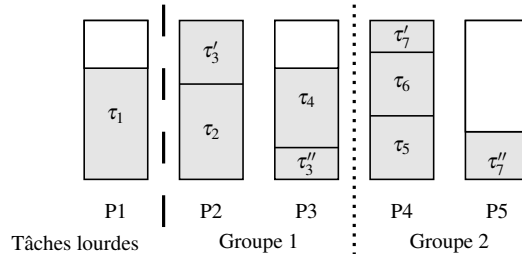


Figure 6. Affectation de 7 tâches sur 5 processeurs selon EKG avec $K = 2$,
 $u_1 = 0,7, u_2 = 0,6, u_3 = 0,6, u_4 = 0,5, u_5 = 0,4, u_6 = 0,4, u_7 = 0,5$

563 Les tâches migrantes s'exécutent selon le principe d'équité au début et en fin d'in-
 564 tervalles de temps définis par les dates de réveil et d'échéance des tâches d'un même
 565 groupe. Les autres tâches sont exécutées avec la politique EDF. EKG n'est optimal que
 566 pour des tâches périodiques et pour K égal au nombre de processeurs. Cependant, le
 567 nombre de préemptions et migrations augmente fortement dans ce cas, il convient
 568 donc de chercher la valeur de K la plus faible permettant d'ordonnancer le système
 569 de tâches considéré. EKG a été entendu aux tâches sporadiques (Andersson, Bletsas,
 570 2008) et aux tâches à échéances arbitraires (Andersson *et al.*, 2008).

571 L'algorithme EKG assigne les tâches migrantes et non-migrantes en même temps,
 572 mais d'autres politiques procèdent en deux étapes distinctes. Dans un premier temps,
 573 les tâches sont affectées selon un algorithme de partitionnement, puis dans un second
 574 temps, les tâches qui n'ont pas été affectées sont réparties sur plusieurs processeurs,
 575 suivant un second algorithme.

576 L'algorithme EDHS (Kato, Yamasaki, 2008d) utilise cette seconde approche. La
 577 figure 7 montre la technique de partitionnement de EDHS. Au cours de la seconde
 578 étape, chaque processeur ne peut accueillir qu'une seule tâche migrante.

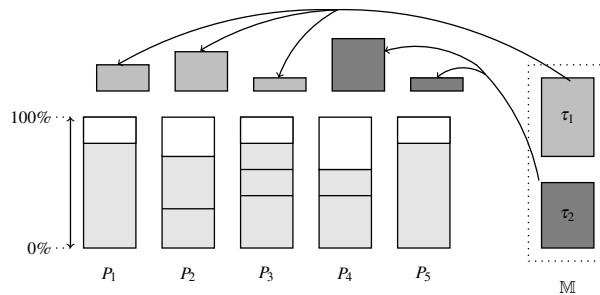


Figure 7. Semi-partitionnement selon EDHS

579 Les tâches migrantes s'exécutent successivement sur les différents processeurs
 580 pour une durée déterminée lors de la phase d'affectation des tâches. Les autres tâches
 581 s'exécutent sur leur processeur suivant l'ordonnancement EDF, mais en laissant la
 582 priorité aux tâches migrantes. Lorsqu'une tâche migrante a épuisé son temps d'exécution
 583 sur un processeur, elle continue son exécution immédiatement sur le processeur
 584 suivant en préemptant si nécessaire un travail en cours d'exécution.

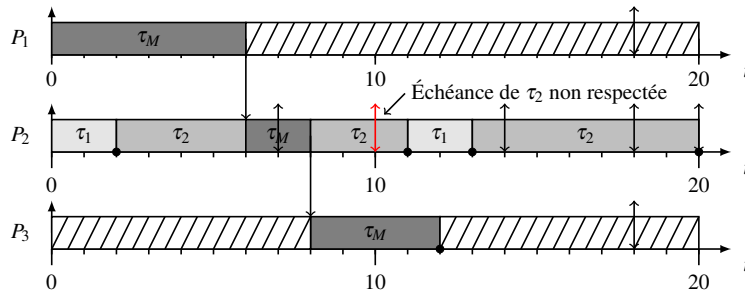


Figure 8. Ordonnancement EDHS des tâches $\tau_M(C_M = 12, T_M = 18)$,
 $\tau_1(C_1 = 2, T_1 = 7)$, $\tau_2(C_2 = 7, T_2 = 10)$

585 Il est intéressant de noter que tout système de tâches ordonnannable par partitionnement,
 586 l'est aussi par EDHS. Corollairement, EDHS ne provoque des migrations que
 587 pour des systèmes qui ne peuvent pas être ordonnancés par partitionnement.

588 La politique EDF-WM (Kato *et al.*, 2009) améliore EDHS en apportant une souplesse
 589 supplémentaire sur l'exécution des tâches migrantes. En effet, comme le montre

590 la figure 8, il existe des cas où EDHS provoque des erreurs qui auraient pu être évi-
 591 tées par plus de flexibilité. Ainsi, pour chaque portion de tâche migrante, une fenêtre
 592 d'exécution possible sur chaque processeur est définie. On obtient alors pour chaque
 593 portion de tâche une date d'arrivée et une date d'échéance. Ces « sous-tâches » sont
 594 ordonnancées comme les autres tâches selon EDF, mais en se basant sur les dates
 595 virtuelles d'activation et d'échéance.

596 Selon ce même principe de *task splitting* où au plus $m - 1$ tâches peuvent être
 597 découpées en deux parties s'exécutant sur deux processeurs fixés statiquement, Burns
 598 *et al.* (Burns *et al.*, 2012) propose une stratégie basée sur le schéma « C=D », à savoir :
 599 la première portion de toute tâche migrante reçoit une échéance identique à sa durée
 600 d'exécution, ce qui l'amène à s'exécuter sans préemption et laisse à la deuxième por-
 601 tion le délai maximum possible pour s'exécuter sur un autre processeur. L'ordonnan-
 602 cement des tâches non migrantes et parties de tâches migrantes relève ensuite d'EDF.
 603 Les auteurs mettent en avant la simplicité de mise en oeuvre d'une telle approche et
 604 donc l'économie de surcoûts système associés. En outre, une analyse sur la détermi-
 605 nation des durées d'exécution des premières portions de tâche ainsi qu'une évaluation
 606 expérimentale de performances vis-à-vis de différentes méthodes de bin-packing (dont
 607 dérivent les tâches migrantes) conduisent à des résultats prometteurs.

608 Enfin, dans (George *et al.*, 2011), une étude est faite de diverses déclinaisons d'al-
 609 gorithmes d'ordonnancement semi-partitionnés basés sur EDF. Pour la classe du *por-*
 610 *tioned scheduling*, plusieurs heuristiques sont étudiées pour calculer les échéances
 611 locales associées à chaque portion de tâche s'exécutant sur un processeur statique-
 612 ment choisi. Dans le cas de la *restricted migration*, une heuristique de type *round*
 613 *robin*, EDF-RRJM (*Round Robin Job Migration*), est proposée pour déterminer les
 614 processeurs qui hébergent les différents travaux d'une même tâche migrante. Outre
 615 une étude expérimentale visant à comparer les performances de ces différentes straté-
 616 gies vis-à-vis d'algorithmes d'ordonnancement globaux et partitionnés, des conditions
 617 d'ordonnançabilité pour des ensembles de tâches sporadiques sont aussi établies.

618 6.2. Algorithme RUN

619 L'algorithme RUN (Regnier *et al.*, 2011) permet d'ordonnancer de manière op-
 620 timale un ensemble de tâches périodiques indépendantes, synchrones et à échéances
 621 implicites sur une architecture multiprocesseur. L'optimalité est obtenue à l'aide d'une
 622 version d'équité dite « faible » qui a pour avantage de provoquer moins de préemp-
 623 tions et de migrations que les versions équitables abordées en section 5. L'algorithme
 624 SPRINT (Baldovin *et al.*, 2014) est une modification de RUN pour l'ordonnancement
 625 de tâches sporadiques avec échéances implicites.

626 La politique RUN procède en deux grandes étapes. La première étape, effectuée
 627 hors-ligne, permet de construire un arbre constitué des tâches à ordonnancer au niveau
 628 des feuilles et de *serveurs* au niveau des nœuds. La seconde étape, effectuée en-ligne,
 629 consiste à exécuter l'ordonnancement à partir de cet arbre en suivant un ensemble de
 630 règles basées sur EDF et la notion de *dualité*.

6.2.1. Partie hors-ligne

La construction de l'arbre se fait à partir de tâches à ordonnancer et de deux opérations : *pack* et *dual*. On suppose que le taux d'utilisation du système est égal au nombre de processeurs. Il est en effet simple d'ajouter des tâches *idle* pendant la construction de l'arbre. Avant tout, il convient d'introduire la notion de tâche *dual* : la tâche *dual* d'une tâche τ est notée τ^* et ne s'exécute que pendant les temps d'inactivité de la tâche τ et réciproquement. Celle-ci partage les mêmes échéances que τ , mais a une durée d'exécution complémentaire vis-à-vis de sa période.

Soit un ensemble de $m + k$ tâches et dont la somme des taux d'utilisation est égale à m . Les auteurs montrent alors que l'ordonnancement de cet ensemble de tâches sur m processeurs est possible si et seulement si il est possible d'ordonnancer leurs tâches *dual*es sur k processeurs. La figure 9 illustre ceci à travers un exemple simple.

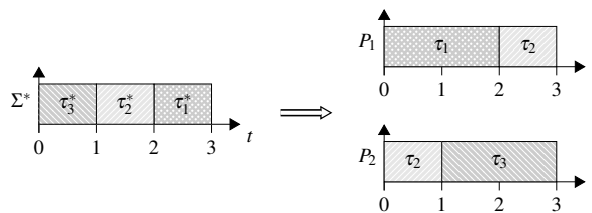


Figure 9. Équivalence entre l'ordonnancement des tâches τ_1 , τ_2 et τ_3 sur 2 processeurs et leurs tâches *dual*es τ_1^* , τ_2^* et τ_3^* sur 1 processeur. Les tâches ont pour période 3, échéance 3 et durée d'exécution 2

L'opération *pack* consiste à regrouper des tâches ou des serveurs avec un algorithme tel que Worst-Fit de telle sorte que l'utilisation totale ne dépasse pas un.

Les opérations de *dual* et *pack* sont enchaînées jusqu'à convergence au nœud racine. Les nœuds sont appelés *serveurs*. Les nœuds formés par un regroupement de serveurs sont appelés « serveur EDF » et les nœuds obtenus par l'opération *dual* sont appelés « serveur dual ». La figure 10 montre l'arbre de réduction pour un système composé de 6 tâches et dont les caractéristiques sont indiquées au niveau des feuilles.

Les échéances des nœuds sont déterminées au moment de la création de l'arbre, en fonction du type de nœud : i) les échéances d'un « serveur dual » sont identiques à celles de son nœud fils, ii) les échéances d'un « serveur EDF » sont l'union des échéances des nœuds fils.

Le taux d'utilisation des nœuds dépend également de leur type : i) le taux d'utilisation d'un « serveur dual » est égal à 1 moins l'utilisation de son nœud fils, ii) le taux d'utilisation d'un « serveur EDF » est égal à la somme des taux d'utilisation des nœuds fils.

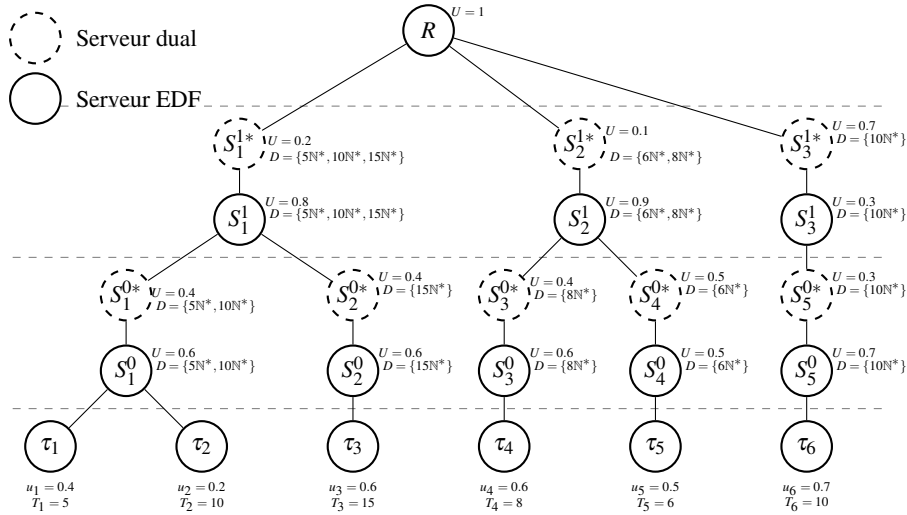


Figure 10. Arbres de réduction

658 6.2.2. Partie en-ligne

659 Un budget proportionnel à leur taux d'utilisation est affecté aux serveurs lors de
 660 l'activation d'une de leurs tâches clientes. À chaque activation, fin d'exécution ou
 661 lorsqu'un budget est écoulé, il est nécessaire de réordonner le système. Afin de
 662 déterminer l'ensemble des tâches à exécuter, l'arbre est parcouru depuis la racine jus-
 663 qu'aux feuilles en appliquant les deux règles suivantes :

- 664 1. Si un serveur EDF s'exécute, alors le nœud fils ayant l'échéance la plus proche
 665 s'exécute. Si un serveur EDF ne s'exécute pas, alors aucun de ses fils ne s'exécute.
 666 2. Si un serveur dual s'exécute, alors son fils ne s'exécute pas et réciproquement.

667 7. Conclusion

668 La vocation de cet article est de constituer un point d'entrée pour tout chercheur
 669 désireux de s'initier aux politiques d'ordonnancement temps réel multiprocesseur. En
 670 effet, au cours de ces vingt dernières années de très nombreux algorithmes ont été
 671 proposés. Nous avons identifié les grandes classes d'ordonnanceurs ainsi que leurs
 672 principaux algorithmes afin de permettre à un lecteur d'appréhender rapidement ce
 673 domaine de recherche.

674 Le choix d'une politique d'ordonnancement dépend de nombreux facteurs pouvant
 675 avoir un impact sur les performances du système tels que le nombre de tâches, leurs
 676 caractéristiques, le nombre de processeurs ou encore l'importance donnée à certains
 677 comportements (temps de réponse, consommation énergétique, préemptions, dépas-
 678 sements d'échéance, etc). Or, les résultats présentés dans la littérature se limitent à

679 un nombre réduit de politiques d’ordonnancement et l’agrégation des divers résultats
 680 obtenus est difficile en raison des différences dans les hypothèses, sur les données
 681 d’entrée, dans les implémentations ou même dans les valeurs mesurées. Ainsi, en pa-
 682 rallèle de notre travail de synthèse présenté ici, nous avons mis en œuvre une vingtaine
 683 de ces politiques d’ordonnancement au sein d’un simulateur appelé SimSo⁴ (Chéramy
 684 *et al.*, 2014). Implémenter ces algorithmes nous a permis de mieux comprendre leur
 685 fonctionnement, mais aussi de les évaluer comparativement dans des conditions expé-
 686 rimentales identiques (Chéramy, 2014). Cet exercice nous a montré en quoi l’écriture
 687 et la mise en œuvre des politiques d’ordonnancement est nécessaire à quiconque sou-
 688 hait étudier et maîtriser en profondeur ce domaine.

689 Soulignons que jusqu’à présent, aucune politique ne s’est véritablement imposée
 690 face aux autres. Les raisons en sont multiples : limites d’utilisation trop faibles,
 691 mise en œuvre trop complexe, hypothèses pas assez réalistes, surcoûts à l’exécution
 692 trop importants, etc. Des politiques telles que RUN ou U-EDF ont montré qu’il était
 693 possible de corriger en partie les défauts des politiques à base d’équité ou de semi-
 694 partitionnement tout en préservant l’optimalité. Il ne faut cependant pas oublier les
 695 politiques moins complexes telles que les variantes de EDF qui permettent d’ordon-
 696 nancer de nombreux systèmes et qui ont l’avantage de pouvoir être combinées avec
 697 des techniques de clustering lorsque le nombre de processeurs devient trop grand. En-
 698 fin, les ordonnanceurs à priorités fixes ne sont pas à exclure car leur fonctionnement
 699 plus simple permet d’étudier des systèmes plus complexes avec, par exemple, des dé-
 700 pendances entre les tâches.

701 Bibliographie

- 702 Anderson J., Bud V., Devi U. (2005). An EDF-based scheduling algorithm for multiprocessor
 703 soft real-time systems. In *Proceedings of ECRTS*.
- 704 Anderson J., Erickson J., Devi U., Casses B. (2014). Optimal semi-partitioned scheduling in
 705 soft real-time systems. In *Proceedings of the 20th RTCSA*.
- 706 Anderson J., Srinivasan A. (1999). *A new look at pfair priorities*. Rapport technique. TR00-023,
 707 University of North Carolina at Chapel Hill.
- 708 Anderson J., Srinivasan A. (2000a). Early-release fair scheduling. In *Proceedings of ECRTS*.
- 709 Anderson J., Srinivasan A. (2000b). Pfair scheduling: beyond periodic task systems. In *Pro-*
 710 *ceedings of the 7th RTCSA*.
- 711 Andersson B. (2003). *Static-priority scheduling on multiprocessors*. Thèse de doctorat non
 712 publiée, Chalmers University of Technology.
- 713 Andersson B. (2008). Global static-priority preemptive multiprocessor scheduling with utiliza-
 714 tion bound 38%. In *Principles of distributed systems*, vol. 5401. Springer.
- 715 Andersson B., Baruah S., Jonsson J. (2001). Static-priority scheduling on multiprocessors. In
 716 *Proceedings of the 22nd RTSS*.

4. L’outil est disponible sur <http://projects.laas.fr/simso/>

- 717 Andersson B., Bletsas K. (2008). Sporadic multiprocessor scheduling with few preemptions.
718 In *Proceedings of the 20th ECRTS*.
- 719 Andersson B., Bletsas K., Baruah S. (2008). Scheduling arbitrary-deadline sporadic task sys-
720 tems on multiprocessors. In *Proceedings of the 29th RTSS*.
- 721 Andersson B., Jonsson J. (2000). Fixed-priority preemptive multiprocessor scheduling: to
722 partition or not to partition. In *Proceedings of the 7th RTCSA*.
- 723 Andersson B., Tovar E. (2006). Multiprocessor scheduling with few preemptions. In *Procee-
724 dings of the 12th RTCSA*.
- 725 Baldovin A., Nelissen G., Vardanega T., Tovar E. (2014). SPRINT: Extending RUN to Schedule
726 Sporadic Tasks. In *Proceedings of the 22nd RTNS*.
- 727 Baruah S. (2004). Optimal utilization bounds for the fixed-priority scheduling of periodic
728 task systems on identical multiprocessors. *IEEE Transactions on Computers*, vol. 53, n° 6,
729 p. 781-784.
- 730 Baruah S., Cohen N., Plaxton C., Varvel D. (1996). Proportionate progress: A notion of fairness
731 in resource allocation. *Algorithmica*, vol. 15, p. 600-625.
- 732 Baruah S., Gehrke J., Plaxton C. (1995). Fast scheduling of periodic tasks on multiple resources.
733 In *International symposium on parallel processing*, p. 280-280.
- 734 Baruah S., Goossens J. (2003). Rate-monotonic scheduling on uniform multiprocessors. *IEEE
735 Transactions on Computers*, vol. 52, n° 7, p. 966-970.
- 736 Bastoni A., Brandenburg B., Anderson J. (2011). Is semi-partitioned scheduling practical? In
737 *Proceedings of the 23rd ECRTS*.
- 738 Bertogna M., Cirinei M., Lipari G. (2005). New schedulability tests for real-time task sets
739 scheduled by deadline monotonic on multiprocessors. In *Proceedings of the 9th opodis*,
740 p. 306-321.
- 741 Bletsas K., Andersson B. (2009). Preemption-light multiprocessor scheduling of sporadic tasks
742 with high utilisation bound. In *Proceedings of the 30th RTSS*.
- 743 Burns A., Baruah S. (2008). Sustainability in real-time scheduling. *Journal of Computing
744 Science and Engineering*, vol. 2, n° 1.
- 745 Burns A., Davis R., Wang P., Zhang F. (2012). Partitioned edf scheduling for multiprocessors
746 using a C=D task splitting scheme. *Real-Time Systems*, vol. 28, n° 1.
- 747 Calandrino J., Leontyev H., Block A., Devi U., Anderson J. (2006). LITMUS^{RT} : A testbed
748 for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th
749 RTSS*.
- 750 Calandrino J. M., Anderson J. H., Baumberger D. P. (2007). A hybrid real-time scheduling
751 approach for large-scale multicore platforms. In *Proceedings of the 19th ECRTS*, p. 247-
752 258.
- 753 Carpenter J., Funk S., Holman P., Srinivasan A., Anderson J. H., Baruah S. (2004). A catego-
754 rization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on
755 scheduling: Algorithms, methods, and models*, chap. 30. Chapman & Hall.
- 756 Chéramy M. (2014). *Étude et évaluation de politiques d'ordonnancement temps réel multipro-
757 cesseur*. Thèse de doctorat non publiée, Université de Toulouse.

- 758 Chéramy M., Hladik P.-E., Déplanche A.-M. (2014). SimSo: A simulation tool to evaluate
759 real-time multiprocessor scheduling algorithms. In *Proceedings of the 5th WATERS*.
- 760 Cho H., Ravindran B., Jensen E. (2006). An optimal real-time scheduling algorithm for multi-
761 processors. In *Proceedings of the 27th RTSS*, p. 101–110.
- 762 Coffman E., Csirik J. (2007). Performance guarantees for one-dimensional bin packing. In
763 T. F. Gonzalez (Ed.), *Handbook of approximation algorithms and metaheuristics*, chap. 32.
764 Chapman and Hall/CRC.
- 765 Davis R., Burns A. (2011). A survey of hard real-time scheduling for multiprocessor systems.
766 *ACM Comput. Surv.*, vol. 43, n° 4, p. 35-44.
- 767 Davis R., Kato S. (2012). FPSL, FPCL and FPZL schedulability analysis. *Real-Time Systems*,
768 vol. 48, n° 6, p. 750-788.
- 769 Devi U., Anderson J. (2005). Tardiness bounds under global EDF scheduling on a multiproces-
770 sor. In *Proceedings of the 26th RTSS*, p. 12.
- 771 Dhall S., Liu C. (1978). On a real-time scheduling problem. *Operations Research*, vol. 26, n° 1,
772 p. 127-140.
- 773 Erickson J., Anderson J. (2012). Fair Lateness Scheduling: Reducing Maximum Lateness in
774 G-EDF-Like Scheduling. In *Proceedings of the 24th ECRTS*.
- 775 Fan M., Quan G. (2012). Harmonic semi-partitioned scheduling for fixed-priority real-time
776 tasks on multi-core platform. In *Proceedings of DATE*, p. 503–508.
- 777 Funaoka K., Kato S., Yamasaki N. (2008). Work-conserving optimal real-time scheduling on
778 multiprocessors. In *Proceedings of the 20th ECRTS*.
- 779 Funk S. (2004). *EDF scheduling on heterogeneous multiprocessors*. Thèse de doctorat non
780 publiée, University of North Carolina at Chapel Hill.
- 781 Funk S., Nanadur V. (2009). LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for
782 Sporadic Task Sets. In *Proceedings of the 17th RTNS*.
- 783 George L., Courbin P., Sorel Y. (2011). Job vs. portioned partitioning for the earliest deadline
784 first semi-partitioned scheduling. *Journal of Systems Architecture*, vol. 57, n° 5.
- 785 Goossens J., Funk S., Baruah S. (2003). Priority-driven scheduling of periodic task systems on
786 multiprocessors. *Real-Time Systems*, vol. 25, n° 2-3, p. 187-205.
- 787 Guan N., Stigge M., Yi W., Yu G. (2010). Fixed-priority multiprocessor scheduling with liu
788 and layland’s utilization bound. In *Proceedings of the 16th RTAS*, p. 165 -174.
- 789 Ha R., Liu J. W. S. (1994). Validating timing constraints in multiprocessor and distributed real-
790 time systems. In *Proceedings of the 14th international conference on distributed computing
791 systems*, p. 162-171.
- 792 Holman P., Anderson J. (2005a). Adapting pfair scheduling for symmetric multiprocessors.
793 *Journal of Embedded Computing*.
- 794 Holman P., Anderson J. (2005b). Adapting Pfair scheduling for symmetric multiprocessors. *J.
795 Embedded Comput.*, vol. 1, p. 543–564.
- 796 Kato S., Yamasaki N. (2007). Real-time scheduling with task splitting on multiprocessors. In
797 *Proceedings of the 13th RTCSA*, p. 441-450.

- 798 Kato S., Yamasaki N. (2008a). Global EDF-based scheduling with efficient priority promotion.
799 In *Proceedings of the 14th RTCSA*, p. 197-206.
- 800 Kato S., Yamasaki N. (2008b). Portioned EDF-based scheduling on multiprocessors. In *Pro-*
801 *ceedings of the 8th acm international conference on emsoft*, p. 1-12. ACM.
- 802 Kato S., Yamasaki N. (2008c). Portioned static-priority scheduling on multiprocessors. In
803 *Proceedings of IPDPS*, p. 1 -12.
- 804 Kato S., Yamasaki N. (2008d). Semi-partitioning technique for multiprocessor real-time sched-
805 ularing. In *Proceedings of the 29th RTSS, work-in-progress session*.
- 806 Kato S., Yamasaki N. (2009). Semi-partitioned fixed-priority scheduling on multiprocessors.
807 In *Proceedings of the 15th RTAS*, p. 23-32.
- 808 Kato S., Yamasaki N., Ishikawa Y. (2009). Semi-partitioned scheduling of sporadic task systems
809 on multiprocessors. In *Proceedings of the 21st ECRTS*, p. 249-258.
- 810 Khemka A., Shyamasundar R. (1997). An optimal multiprocessor real-time scheduling algo-
811 rithm. *Journal of Parallel and Distributed Computing*, vol. 43, n° 1, p. 37-45.
- 812 Kim H., Cho Y. (2011). A new fair scheduling algorithm for periodic tasks on multiprocessors.
813 *Information Processing Letters*, vol. 111, n° 7, p. 301-309.
- 814 Lakshmanan K., Rajkumar R., Lehoczky J. (2009). Partitioned fixed-priority preemptive sched-
815 ularing for multi-core processors. In *Proceedings of the 21st ECRTS*, p. 239-248.
- 816 Lee C., Lee D. (1985). A simple on-line bin-packing algorithm. *J. ACM*, vol. 32, p. 562-572.
- 817 Lee J., Easwaran A., Shin I. (2010). LLF schedulability analysis on multiprocessor platforms.
818 In *Proceedings of the 31st RTSS*, p. 25-36.
- 819 Lee S. (1994). On-line multiprocessor scheduling algorithms for real-time tasks. In *Proceedings*
820 *of the IEEE region 10's ninth annual international conference*.
- 821 Leung J., Whitehead J. (1982). On the complexity of fixed-priority scheduling of periodic,
822 real-time tasks. *Performance Evaluation*, vol. 2, n° 4, p. 237-250.
- 823 Levin G., Funk S., Sadowski C., Pye I., Brandt S. (2010). DP-FAIR: A Simple Model for
824 Understanding Optimal Multiprocessor Scheduling. In *Proceedings of the 22nd ECRTS*,
825 p. 3 -13.
- 826 Liu C. L., Layland J. (1973). Scheduling algorithms for multiprogramming in a hard-real-time
827 environment. *J. ACM*, vol. 20, p. 46-61.
- 828 Massa E., Lima G. (2010). A bandwidth reservation strategy for multiprocessor real-time
829 scheduling. In *Proceedings of the 16th RTAS*, p. 175-183.
- 830 Massa E., Lima G., Regnier P., Levin G., Brandt S. (2014). Optimal and adaptive multiprocessor
831 real-time scheduling: The quasi-partitioning approach. In *Proceedings of the 26th ECRTS*.
- 832 Mok A. (1983). *Fundamental design problems of distributed systems for the hard-real-time*
833 *environment*. Rapport technique. Massachusetts Institute of Technology.
- 834 Nelissen G. (2012). *Efficient optimal multiprocessor scheduling algorithms for real-time sys-*
835 *tems*. Thèse de doctorat non publiée, École polytechnique de Bruxelles.
- 836 Nelissen G., Berten V., Nelis V., Goossens J., Milojevic D. (2012). U-EDF: An unfair but
837 optimal multiprocessor scheduling algorithm for sporadic tasks. In *Proceedings of the 24th*
838 *ECRTS*, p. 13-23.

- 839 Nelissen G., Su H., Guo Y., Zhu D., Nélis V., Goossens J. (2014). An optimal boundary fair
840 scheduling. *Real-Time Systems*, p. 1–53.
- 841 Oh D.-I., Bakker T. (1998). Utilization bounds for n-processor rate monotone scheduling with
842 static processor assignment. *Real-Time Systems*, vol. 15, p. 183-192.
- 843 Oh S.-H., Yang S.-M. (1998). A modified least-laxity-first scheduling algorithm for real-time
844 tasks. In *Proceedings of the 5th RTCSA*, p. 31-36.
- 845 Park M., Han S., Kim H., Cho S., Cho Y. (2005). Comparison of deadline-based scheduling al-
846 gorithms for periodic real-time tasks on multiprocessor. *IEICE transactions on information
847 and systems*, vol. 88, n° 3, p. 658-661.
- 848 Regnier P., Lima G., Massa E., Levin G., Brandt S. (2011). RUN: Optimal Multiprocessor
849 Real-Time Scheduling via Reduction to Uniprocessor. In *Proceedings of the 32nd RTSS*,
850 p. 104-115.
- 851 Shin I., Easwaran A., Lee I. (2008). Hierarchical scheduling framework for virtual clustering
852 of multiprocessors. In *Proceedings of ECRIS*, p. 181–190.
- 853 Sousa P. B., Souto P., Tovar E., Bletsas K. (2013). The Carousel-EDF scheduling algorithm for
854 multiprocessor systems. In *Proceedings of the 19th RTCSA*.
- 855 Srinivasan A., Baruah S. (2002). Deadline-based scheduling of periodic task systems on mul-
856 tiprocessors. *Inf. Process. Lett.*, vol. 84, p. 93–98.
- 857 Yang K., Anderson J. H. (2014). Optimal GEDF-based schedulers that allow intra-task paral-
858 lelism on heterogeneous multiprocessors. In *Proceedings of the 12th ESTIMedia*.
- 859 Zhu D., Mosse D., Melhem R. (2003). Multiple-resource periodic scheduling problem: how
860 much fairness is necessary? In *Proceedings of the 24th RTSS*, p. 142-151.

861

Article soumis le 23/03/2015

862

Accepté le 8/09/2015