THESE

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Faculté des Sciences Fondamentales et Appliquées
Ecole Doctorale des Sciences pour l'Ingénieur et Aéronautique

Diplôme National - Arrêté du 7 août 2006

Secteur de Recherche

Informatique

Présentée par

Bernard Chauvière

Des priorités fixes aux priorités dynamiques en ordonnancement temps-réel : étude comparative et calcul des priorités

---

Directeur de thèse : Dominique Geniet

Soutenue le 20 décembre 2007 devant la commission d'examen

JURY

Rapporteurs :

Ye-Qiong Song	Professeur, Nancy-Université, INPL et LORIA.
Patrick Martineau	Professeur, Ecole Polytechnique de l'Université de Tours, LI.

Examinateurs :

René Schott	Professeur, Université Henri Poincaré (Nancy 1), IECN et LORIA.
Joël Goossens	Professeur associé, Université Libre de Bruxelles, SSD.
Annie Choquet-Geniet	Professeur, Université de Poitiers, LISI.
Dominique Geniet	Maître de Conférences, Université de Poitiers, LISI.

Laboratoire d'Informatique Scientifique et Industrielle

Remerciements

Je remercie vivement M. Pierra, directeur du LISI, pour m'avoir permis d'intégrer l'équipe temps-réel, et aussi pour les moyens informatiques qu'il a mis à ma disposition.

Mes travaux ont été dirigés depuis plus de cinq ans par M. Geniet. Toute ma gratitude lui est destinée pour le rôle central qu'il a tenu pendant toutes ces années. Je tiens aussi à saluer nos complémentarités qui ont alimenté bon nombre de discussions passionnées. Mes remerciements s'adressent aussi à Mme Choquet-Geniet pour les nombreuses discussions que nous avons eues ensemble sur des thèmes de recherche communs.

Je tiens également à remercier les membres de l'équipe TRIO au LORIA qui m'a accueilli pendant plus de six mois : Mme Simonot-Lion, Mme Cucu, M. Navet, M. Zendra, M. Song, M. Rebeuf, et M. Grenier. Les deux séjours effectués ont été très enrichissants. Mon attention se porte tout particulièrement sur M. Schott qui a rendu ces séjours possibles et qui a dirigé les activités que j'ai menées là-bas. Notre collaboration a débuté lors de mon DEA, et pendant tout ce temps, il a toujours su prodiguer les conseils dont j'avais besoin, et apporter l'ouverture nécessaire.

Je tiens aussi à remercier l'équipe d'enseignement de l'ENSEM qui m'a accueilli, en particulier : M. Thomesse, M. Dufner, M. Brinzei, M. Song, M. Rebeuf, et M. Tricot. J'ai pu grâce à eux découvrir le monde de l'enseignement et notamment faire mes premières armes.

Je tiens à exprimer mes sincères remerciements :
- à M. Song et M. Martineau, pour avoir accepté la lourde charge de rapporteurs,
- à Mme Choquet-Geniet et M. Goossens, pour avoir accepté d'être examinateurs.

Finalement, je tiens à remercier toute l'équipe du LISI et plus particulièrement Mme Choquet-Geniet, M. Richard, M. Grolleau, Mme Largeteau, M. Ridouard, M. Sanou, M. Traoré et M. Carreau.

Table des matières

Introduction générale

Un système informatique dont le fonctionnement ne dépend pas seulement des résultats produits mais aussi des instants auxquels ces résultats sont produits est qualifié par le terme générique de *système temps-réel* [103]. Cette définition est très générale. Parmi les spécificités de ces systèmes, on reconnaît généralement leur besoin de *réactivité* et de *ponctualité*. La rapidité n'est pas recherchée en temps-réel, par contre le système doit toujours réagir à temps. Pour assurer cette propriété, leur comportement doit être *prévisible*. Les systèmes temps-réel sont surtout utilisés pour contrôler des appareils autonomes plus ou moins critiques. On les retrouve dans les hautes technologies (par exemple, l'aérospatiale, le contrôle de centrale nucléaire), et aussi au quotidien (par exemple, la téléphonie mobile, l'automobile).

Systèmes temps-réel. Les systèmes informatiques temps-réel sont décomposés en plusieurs tâches associées à des traitements particuliers. Ces traitements sont généralement déclenchés par des évènements. En fonction de la fréquence de ces évènements, on distingue plusieurs types de tâches. Nos travaux concernent principalement les tâches périodiques ordonnancées sur une architecture multiprocesseur. Pour garantir la réactivité d'un système temps-réel, on impose des échéances temporelles aux tâches : une fois activée par l'occurrence d'un évènement, une tâche dispose d'une durée limitée pour effectuer le traitement qui lui est associé. En conséquence, l'ordre selon lequel les tâches sont exécutées est primordial. Pour déterminer un ordre permettant de garantir le respect de toutes les échéances, on utilise une *politique d'ordonnancement en-ligne* qui attribue une priorité à chaque tâche. Celle dont la priorité est la plus élevée est alors choisie par l'algorithme pour être exécutée. On distingue généralement deux types de politiques d'ordonnancement : celles à priorités fixes, et celles à priorités dynamiques. Les mécanismes attribuant les priorités doivent avoir une complexité faible (généralement linéaire) puisqu'ils sont exécutés fréquemment et en concurrence des tâches constituant le système : le temps pris pour choisir la tâche à exécuter doit être négligeable en comparaison de la durée d'exécution des tâches.

Ordonnancement temps-réel. Les politiques d'ordonnancement ont été abondamment étudiées. En fonction des contraintes imposées aux systèmes temps-réel, on dispose ou non de politiques d'ordonnancement optimales - i.e. dès que le système est ordonnançable, la politique optimale l'ordonnance correctement. Toutefois, dès que plusieurs ressources sont partagées, il n'existe pas de politique optimale[34], et le problème consistant à trouver une séquence d'exécution valide est NP-dur[13, 70, 71]. Pour ces contextes, des approches dites hors-ligne ont été développées. Une des plus connues est sans doute celle présentée dans [9] : elle permet de trouver une configuration de priorité fixe ordonnançant un système de tâches dès qu'il en existe une. Toutefois, cet algorithme n'est optimal que pour les systèmes monoprocesseurs.

D'autres méthodes hors-ligne dites exactes ont été développées, elles sont basées sur des modèles à états, par exemple automates finis [2, 3, 4, 63], réseaux de Petri [18, 45, 72], géométrie discrète [62, 64]. Elles reposent sur une exploration exhaustive de toutes les séquences d'exécution. En conséquence, leur complexité est exponentielle et les rend souvent inutilisables pour des systèmes de tâches de taille industrielle.

Pour obtenir une solution par un calcul de complexité polynomiale, plusieurs méthodes approchées ont été élaborées. Elles reposent généralement sur la descente de gradient, le recuit simulé, la

méthode taboue, les algorithmes génériques, ou encore les processus de décision markoviens. Elles ont déjà été appliquées avec succès au problème de l'allocation de fréquence [53, 54] et à celui de la planification de trajectoire en robotique [65]. A notre connaissance, seuls les algorithmes génétiques ont été appliqués à l'ordonnancement des systèmes temps-réel [87].

Validation des systèmes temps-réel. Avant d'exploiter un système temps-réel, on doit garantir qu'il respecte les contraintes temporelles imposées par sa spécification : c'est l'étape de *validation*. De nombreuses méthodes ont été élaborées pour répondre à cet objectif. Dans certains cas, on dispose de conditions analytiques permettant de garantir le comportement d'un système temps-réel. Elles sont généralement de faible complexité (polynomiale), mais aussi pessimistes dans le sens où elles ne sont que suffisantes : elles peuvent ne pas être satisfaites bien que le système soit valide. Lorsqu'aucune condition analytique ne permet de garantir le bon fonctionnement d'un système temps-réel, on est généralement amené à effectuer une simulation. Cette approche est souvent utilisée, cependant elle n'est pas exempte de défauts. Par exemple, les simulations permettent de valider un certain nombre de situations, mais elles ne permettent généralement pas de garantir à 100% le fonctionnement d'un système temps-réel. Toutefois, elles constituent un outil intéressant pour étudier le comportement d'un système de tâches.

La durée de vie d'un système temps-réel est généralement considérée comme infinie. Toutefois, on ne peut pas réaliser de simulations de durée infinie. Pour étudier un système temps-réel, on doit donc déterminer un intervalle d'étude, c'est-à-dire un intervalle temporel représentatif de l'ensemble de la vie d'un système temps-réel. Ces intervalles ont été beaucoup étudiés. En contexte monoprocesseur, on dispose d'une borne de la durée de simulation nécessaire, on peut donc effectuer une simulation pour étudier ce type de systèmes. En contexte multiprocesseur, seuls les politiques d'ordonnancement à priorités fixes ont été étudiées jusqu'à maintenant [22, 24, 25].

Diagnostic des systèmes temps-réel. Le respect des échéances des tâches permet de garantir que le système temps-réel est réactif. Cette propriété est indispensable puisqu'elle assure qu'il évolue à la vitesse du système contrôlé. Cependant, d'autres propriétés portant sur le fonctionnement d'un système temps-réel sont importantes, elles constituent la *qualité de service* (QoS) fournie par le système. Par exemple, on peut chercher à minimiser la *gigue* : cela permet d'assurer la régularité de certains traitements périodiques. L'étude de la qualité de service est importante dans le cadre du temps-réel souple - i.e. le dépassement d'une échéance n'entraîne pas de conséquence grave pour le système contrôlé. Par exemple, elle est très utilisée pour caractériser les performances des transferts d'informations multimédias dans les réseaux informatiques.

Lorsqu'un système de tâches n'est pas ordonnançable, on peut réagir de différentes manières. Par exemple, on peut augmenter la puissance de traitement du système informatique hébergeant le système de tâches. Toutefois, en fonction des contraintes financières ou énergétiques, cette solution n'est pas toujours envisageable. On peut aussi modifier la spécification ou la conception du système temps-réel. Dans ce cas, les informations que l'on peut fournir au concepteur du système sont alors primordiales puisqu'elles vont guider les corrections à apporter.

Contributions. Nous avons suivi trois axes de recherche. Tout d'abord, nous avons développé différentes méthodes attribuant des priorités aux tâches de manière à respecter les contraintes imposées à un système de tâches. Cette étude nous a ensuite amenés à étudier certains problèmes théoriques propres à l'ordonnancement des tâches périodiques, comme par exemple la cyclicité et la durée de montée en charge. Ce type de résultat est important en temps-réel puisqu'il permet d'obtenir une durée d'exécution permettant l'étude du système temps-réel à l'aide d'une simulation. On peut ainsi faire ressortir certaines propriétés sur la fiabilité d'un système temps-réel. Finalement, nous avons mené une étude dont l'objectif est double : obtenir des informations sur la qualité de service fournie par un système temps-réel, et aider le concepteur à modifier la spécification d'un système temps-réel lorsque celui-ci n'est pas ordonnançable.

Initialement, nous avons repris l'approche suivie dans [9] pour attribuer des priorités fixes aux tâches. La recherche de politique en-ligne optimale correspond à une approche généraliste : on détermine une solution utilisable pour tous les systèmes de tâches. Dès que plusieurs ressources sont partagées, une telle solution n'existe plus [34]. L'approche de [9] consiste à rechercher une attribution des priorités propres à un système de tâches : ses spécificités peuvent alors être prise en compte. Pour les systèmes monoprocesseurs, elle a permis la mise en place d'une méthode optimale pour attribuer des priorités fixes. Cependant, il a été montré dans [6] qu'elle n'est plus optimale dans le cas multiprocesseur. Nous avons alors recherché une autre méthode permettant de poursuivre l'approche initiée par [9]. Après avoir caractérisé les relations entre les priorités des tâches propres à une séquence d'exécution donnée, nous avons établi un algorithme fournissant les configurations de priorité fixe produisant cette même séquence. Les séquences traitées peuvent être produites par n'importe quelle méthode et notamment par les méthodes hors-ligne. Notre résultat permet ensuite de les implanter dans les exécutifs temps-réel sans utiliser de séquenceur, donc en conservant la souplesse propre à l'ordonnancement en-ligne. En poursuivant cette approche, nous avons aussi obtenu un algorithme permettant d'obtenir toutes les configurations de priorité fixe valides. Il est donc optimal pour attribuer des priorités fixes pour les systèmes multiprocesseurs. En cela, il prolonge l'approche initiée par [9]. En utilisant notre algorithme, nous avons pu évaluer expérimentalement la puissance d'ordonnancement des exécutifs temps-réel à priorités fixes. Nous l'avons aussi comparée à celle des politiques en-ligne classiques. Nous avons constaté que l'ordonnancement en priorités fixes fournit de bons résultats pour les systèmes dont la charge n'est pas trop élevée. Toutefois, ses performances s'effondrent rapidement lorsque la charge augmente.

Pour dépasser la puissance d'ordonnancement fournie par l'ordonnancement en priorités fixes, nous avons étudié une classe de politiques d'ordonnancement présentée dans [20]. Elle se situe entre les priorités fixes et les priorités dynamiques. On considère que la priorité d'une tâche est calculée à son activation, elle conserve ensuite cette priorité jusqu'à sa terminaison. La priorité attribuée à une tâche peut donc changer entre deux activations. Dans la suite, nous disons que les politiques appartenant à cette classe sont à priorités fixes par instances. Ainsi définie, cette classe contient une infinité de configurations de priorité, il n'est alors plus possible de rechercher toutes celles qui sont valides. En outre, certaines configurations produisent des séquences d'exécution aux propriétés surprenantes : certaines peuvent être acycliques ! Nous nous sommes donc restreints à une sous-classe, notée $\overline{\overline{\text{PFI}}}$, en considérant que deux instances d'une même tâche et distantes d'exactement une métapériode ont la même priorité. Nous avons tout d'abord étudié les propriétés théoriques de ce contexte, et nous avons montré qu'il est optimal en monoprocesseur : dès qu'un système de tâches est ordonnançable en monoprocesseur, alors il existe une configuration de priorité $\overline{\overline{\text{PFI}}}$ qui l'ordonnance. Ensuite, nous avons étendu à cette classe la méthode que nous avons élaborée pour les politiques à priorités fixes. Cela nous a amenés à proposer trois algorithmes permettant de résoudre les problèmes suivants pour un système de tâches donné :

1. existence d'une configuration de priorité fixe par instances,
2. calcul de toutes les configurations de priorité $\overline{\overline{\text{PFI}}}$ engendrant une séquence d'exécution donnée,
3. calcul de toutes les configurations de priorité $\overline{\overline{\text{PFI}}}$ valides.

Les algorithmes (2) et (3) permettent de produire des solutions d'ordonnancement pour un système de tâches donné. A l'aide de l'algorithme (3), nous avons pu évaluer expérimentalement la puissance d'ordonnancement de la classe des politiques à priorités fixes par instance. Cette classe de politiques d'ordonnancement permet d'atteindre une puissance d'ordonnancement bien supérieure à celle des politiques à priorités fixes.

Pour obtenir des solutions d'ordonnancement, nous avons aussi utilisé des méthodes d'approximation : elles permettent d'obtenir une solution approchée d'un problème NP en un temps polynomial. Dans le cadre d'une recherche menée au LORIA, nous nous sommes intéressés à l'ordonnancement des systèmes de tâches liées par des relations de précédence. Nous avons commencé par étudier les algorithmes génétiques proposés par [52, 23]. Ces deux algorithmes sont relativement différents. Celui proposé par [23] fournit de meilleures solutions que celui proposé par [52], mais

en utilisant un temps de calcul nettement supérieur. Pour obtenir une complexité de calcul assez faible, nous avons tout d'abord proposé plusieurs variantes de celui de [52]. Nous avons constaté expérimentalement qu'elles permettent d'augmenter ses performances. Nous avons ensuite élaboré un algorithme reposant sur la méthode taboue qui offre des performances supérieures et dépassant même celles de l'algorithme génétique proposé par [23].

Lors de notre étude des politiques d'ordonnancement à priorités fixes et de celles à priorités fixes par instances, nous avons été confrontés à plusieurs problèmes théoriques. Effectivement, pour déterminer les configurations de priorité valides, il nous a fallu montrer à partir d'une durée d'étude finie que les configurations obtenues étaient indéfiniment valides. Pour cela, nous avons étudié les intervalles de faisabilité et les deux notions sous-jacentes : la cyclicité des séquences d'exécution et la durée de montée en charge - i.e. la durée permettant d'atteindre le régime permanent. Notre étude concerne la sous-classe, notée $\overline{\text{PFI}}$, des politiques à priorités fixes par instances restreinte à celles qui conservent entre les différentes métapériodes les relations de priorité entre les instances. Nous avons montré que les politiques de $\overline{\text{PFI}}$ produisent en multiprocesseur des séquences d'exécution périodiques, et nous avons aussi borné leur durée de montée en charge. Nous en avons alors déduit un intervalle de faisabilité valable pour toutes les politiques $\overline{\text{PFI}}$. Notre étude s'applique par exemple aux politiques Rate Monotonic (RM), Deadline Monotonic (DM) et Earliest Deadline First (EDF). A notre connaissance, notre intervalle de faisabilité est le premier connu pour EDF.

Pour obtenir un intervalle de faisabilité plus précis, nous nous sommes ensuite focalisés sur les politiques à priorités fixes. Plusieurs résultats concernant ce type de séquences d'exécution existent déjà [24, 25], ils fournissent entre autres des majorations de la durée de montée en charge. En affinant l'étude menée dans [24], nous avons déterminé la durée exacte de montée en charge des séquences d'exécution multiprocesseurs produites par les politiques à priorités fixes. Son calcul est de complexité exponentielle (au pire), nous avons donc proposé plusieurs majorations de la durée de montée en charge. Nous les avons comparées expérimentalement avec celle fournie par [24]. Les résultats montrent que nos majorations apportent une amélioration significative. Nous avons aussi constaté que la complexité moyenne du calcul de la durée exacte de montée en charge est en moyenne plus proche d'une complexité polynomiale que d'une complexité exponentielle.

Les problèmes liés à l'ordonnancement et à la validation des systèmes de tâches sont centraux dans l'étude des systèmes temps-réel. Toutefois, le diagnostic de ces systèmes et l'étude de la qualité de service qu'ils offrent sont aussi des thèmes de recherche importants. Le comportement d'un système temps-réel est influencé par les choix d'ordonnancement réalisés. L'étude de la qualité de service permet de quantifier l'influence de ces choix. Parfois, elle peut servir à départager deux politiques d'ordonnancement valides, elle peut aussi montrer qu'une politique bien qu'étant valide produit une séquence d'exécution de très mauvaise qualité, par exemple en termes de nombre de changements de contexte. Il est donc intéressant de réaliser une étude plus poussée que celle consistant à déterminer une solution d'ordonnancement valide. Les informations fournies doivent aussi se révéler intéressantes dans le cas où le système n'est pas ordonnançable : elles pourront alors aider à résoudre les problèmes rencontrés. Pour aborder cette problématique, nous avons élaboré un modèle basé sur les chaînes de Markov. Il nous a permis de calculer les distributions de probabilité du temps de réponse de chaque instance en fonction de l'exécutif ciblé. Ces informations permettent d'obtenir un diagnostic permettant de faire ressortir les parties critiques d'un ordonnancement, elles permettent alors d'identifier les points faibles d'un système temps-réel qui pourront guider le concepteur pour effectuer ses modifications. A partir des distributions de probabilité du temps de réponse des instances, on peut aussi obtenir d'autres indicateurs plus orientés vers la qualité de service, comme par exemple la gigue. Notre méthode peut donc servir à l'analyse de la qualité de service fournie par un système temps-réel.

L'approche que nous avons suivie en utilisant les chaînes de Markov consiste à étudier l'ensemble des ordonnancements valides. Plus précisément, on se restreint à ceux dont la probabilité est non-nulle. Pour réaliser une étude finie, on utilise un intervalle d'étude. Celui-ci est très similaire aux intervalles de faisabilité utilisés pour la simulation d'une politique d'ordonnancement parti-

culière. Toutefois, il doit être valable pour tous les ordonnancements traités. Or, pour certaines politiques d'ordonnancement, comme Least Laxity First (LLF) par exemple, on ne connaît pas d'intervalle de faisabilité valable en contexte multiprocesseur. Pour obtenir un intervalle d'étude, nous avons élaboré une nouvelle approche qui ne dépend pas de la cyclicité des séquences d'exécution concernées mais de la cyclicité des états qu'elles permettent d'atteindre. Les résultats obtenus sont surprenants : l'intervalle d'étude est parfois plus petit que l'intervalle de faisabilité de certaines des séquences d'exécution concernées. Ainsi, le fait de considérer simultanément plusieurs séquences d'exécution permet de raccourcir l'intervalle d'étude.

Plan du mémoire. Dans la première partie, nous présentons tout d'abord les systèmes temps-réel auxquels nous nous sommes consacrés (chapitre 1). Ensuite, nous présentons les principaux résultats connus (chapitre 2).

Les méthodes que nous avons élaborées reposent toutes sur l'étude de la cyclicité des séquences d'exécution multiprocesseurs que nous avons menée. Nous présentons donc ces résultats dès la seconde partie. Nous commençons celle-ci en définissant les sous-classes des politiques à priorités fixes par instances que nous avons étudiées ($\overline{\text{PFI}}$ et $\overline{\overline{\text{PFI}}}$). Ensuite, nous donnons quelques exemples illustrant leurs propriétés spécifiques. Ces deux points font l'objet du chapitre 3. Nos résultats concernant la cyclicité des séquences $\overline{\text{PFI}}$ sont présentés dans le chapitre 4, et ceux concernant les politiques à priorités fixes dans le chapitre 5. Nous terminons cette partie en déterminant la durée d'étude propre au modèle basé sur les chaînes de Markov que nous avons mis en place (chapitre 6).

La troisième partie est consacrée aux méthodes fournissant des solutions d'ordonnancement. Dans le chapitre 7, nous présentons celle que nous avons élaborée pour déterminer les configurations de priorité fixe ordonnançant un système de tâches donné. L'étude menée sur les configurations de priorité fixe par instances est abordée dans le chapitre 8. Nous terminons cette partie en exposant les méthodes d'approximation que nous avons développées pour les systèmes de tâches liées par des relations de précédence (chapitre 9).

Dans la quatrième partie, nous proposons une étude permettant de diagnostiquer le comportement des systèmes temps-réel. Nous commençons par définir les chaînes de Markov que nous avons utilisées. Ensuite, nous établissons une méthode permettant d'intégrer les contraintes d'exécution dues aux interactions entre les tâches (par exemple, précédence, exclusion mutuelle) et aussi celles dues aux spécificités de l'exécutif temps-réel ciblé (par exemple, préemption, migration). Cette étape de modélisation est présentée dans le chapitre 10. L'évaluation du modèle que nous avons défini nécessite une complexité importante, nous avons alors conçu un algorithme adapté que nous présentons dans le chapitre 11. Nous terminons cette partie en présentant les résultats que nous avons obtenus sur l'étude de la probabilité du temps de réponse de chaque instance (chapitre 12). A l'aide d'un exemple, nous illustrons les diagnostics que nos résultats permettent de formuler.

Première partie

Contexte d'étude et état de l'art

Chapitre 1

Systèmes temps-réel

1.1 Domaine d'application

L'informatique temps-réel est apparue avec le développement des systèmes embarqués. Ces systèmes sont généralement *autonomes*, parfois il est même impossible d'intervenir directement dessus, comme par exemple dans l'aérospatial. Leur pilotage est alors assuré par un programme informatique appelé *application temps-réel*. On retrouve ce type de programme dans de nombreux domaines, citons par exemple l'aviation, l'automobile, la téléphonie portable, les chaînes de montage industrielles, etc. Plus récemment, l'informatique temps-réel a aussi été utilisé dans le domaine des réseaux, notamment pour le transfert d'informations multimédia.

Le qualificatif *temps-réel* désigne toute application mettant en œuvre un système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état d'un environnement (appelé procédé) qui lui est connecté et dont il doit contrôler le comportement [29].

Le rôle d'une application temps-réel est donc de contrôler un procédé. Pour cela, elle doit être *réactive* et donc évoluer à la vitesse de l'appareil piloté. Le rapport au temps de ces systèmes est critique et constitue le principal enjeu lors de leur conception. Toutefois, toutes les opérations réalisées par une application temps-réel ne sont pas vitales : on distingue les contraintes *strictes* des contraintes *souples*. Lorsque certaines contraintes imposées à un système temps-réel sont strictes, on parle de *temps-réel dur*, par exemple dans l'aérospatial ; dans le cas contraire, on parle de *temps-réel mou*, par exemple pour le multimédia. Une autre caractéristique importante de ces systèmes est leur autonomie. Dans certains cas, aucune intervention extérieure n'est possible en cas de difficultés, le système doit alors être *tolérant aux fautes*. Deux situations correspondent à cette particularité : premièrement, si une partie peu importante du système devient défaillante, cela ne doit pas empêcher le reste du système de fonctionner normalement, deuxièmement, si le système se retrouve dans une situation indésirable, il doit être capable de retourner dans un état stable appartenant à un schéma de fonctionnement normal.

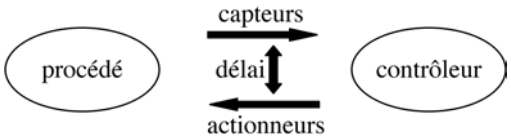


FIG. 1.1 – Interactions entre le procédé et le contrôleur

1.2 Contrôle de procédé

Les systèmes temps-réel sont constitués de deux entités distinctes. Le *procédé* regroupe l'appareillage physique du système, et le *contrôleur* est chargé d'assurer le bon comportement du système. La coopération de ces deux entités suit généralement le schéma présenté sur la figure 1.1. En résumé, le procédé recueille une information via un *capteur* et la transmet au contrôleur qui la traite et décide ou non de réagir. Dans le cas positif, il transmet un ordre au procédé qui le met en œuvre via des *actionneurs*.

Le délai séparant l'arrivée d'une information dans le contrôleur et l'émission de l'action résultante correspond au *temps de réponse* du système temps-réel. En fonction des applications et des exigences de l'environnement, le système doit être plus ou moins *réactif* - i.e. le temps de réponse doit être plus ou moins court. La validation des systèmes temps-réel porte sur l'étude de leur temps de réponse afin de déterminer s'il respecte les exigences de leur spécification.

1.3 Approches synchrone et asynchrone

Pour implémenter un système temps-réel, on peut suivre soit l'approche *synchrone*, soit l'approche *asynchrone*. Pour certains systèmes, le temps de traitement utilisé par le contrôleur est négligeable par rapport à la vitesse d'évolution de l'environnement. Dans ce cas, on peut supposer que le temps de traitement est nul, et donc que la réaction du procédé est immédiate : c'est l'hypothèse synchrone. Pour d'autres systèmes, soit en vertu des exigences de l'environnement, soit en vertu de la masse d'informations à traiter, le temps de traitement utilisé par le contrôleur n'est pas négligeable : c'est l'hypothèse asynchrone.

L'approche synchrone a abouti au développement de langages de programmation propres aux systèmes temps-réel, par exemple, ESTEREL[15, 14], LUSTRE[49, 51] et SIGNAL[46]. Le temps de traitement étant considéré comme nul, cette approche présente d'indiscutables avantages :

1. l'ordre dans lequel sont effectués les traitements n'a pas d'importance,
2. il n'est pas nécessaire d'interrompre certains traitements au profit d'autres plus urgents, en vertu de (1),
3. les problèmes posés par l'exclusion mutuelle de certains traitements sont évités, en vertu de (2).

Toutefois, cette approche revient à supposer que l'architecture matérielle est surdimentionnée par rapport à la complexité des traitements du contrôleur. En fonction des coûts de fabrication et des contraintes énergétiques, elle n'est pas toujours utilisable.

Sous l'hypothèse asynchrone, le temps de traitement utilisé par le contrôleur n'est pas négligeable. Avant d'implanter le système temps-réel, il est alors nécessaire de vérifier que ses temps de réponse vérifient les contraintes imposées par sa spécification. On doit aussi prendre en compte les interactions possibles entre les différents traitements, par exemple : accès au processeur, exclusion mutuelle. Parfois, il est alors nécessaire d'interrompre des traitements longs au profit d'autres qui sont plus urgents : l'ordre selon lequel sont effectués les différents traitements est alors *primordial*. Pour implanter de tels systèmes, on utilise des *politiques d'ordonnancement* pour déterminer l'ordre selon lequel les différents traitements doivent être effectués.

1.4 Tâches temps-réel

Un système temps-réel est composé d'un ensemble de tâches. Chaque tâche correspond à un traitement particulier déclenché par l'occurrence d'un évènement spécifique. Lorsqu'un évènement se produit périodiquement (par exemple, un envoi de donnée depuis un capteur), la tâche associée est dite périodique ; dans le cas contraire elle est dite apériodique (par exemple, un signal d'alarme). Parmi les tâches apériodiques, on distingue les tâches sporadiques : la durée séparant deux occurrences successives de l'évènement associé admet alors une borne minimum.

Dans toute la suite, nous désignons par τ un système de n tâches temps-réel, et par τ_k la $k^{\grave{e}}$ tâche de ce système. Chaque occurrence de l'évènement associé à une tâche τ_k déclenche la création d'un processus exécutant le programme associé à τ_k. On dit aussi que ces processus sont des *instances* de la tâche τ_k. On désigne par $\tau_{k,i}$ la $i^{\grave{e}}$ instance de la tâche τ_k, et par $\overline{\tau}$ l'ensemble des instances engendrées par les tâches de τ.

1.4.1 Tâches périodiques

Les tâches périodiques sont généralement représentées par quatre paramètres temporels (voir figure 1.2) :

- $r_k \in \mathbb{N}$: date de première activation,
- $C_k \in \mathbb{N}$: pire durée d'exécution,
- $D_k \in \mathbb{N}$: délai critique,
- $T_k \in \mathbb{N}$: période.

Le paramètre r_k indique la date de la première occurrence de l'évènement associé à τ_k. Ensuite, toutes les T_k unités de temps, une instance de τ_k est générée. Lorsque $r_k > 0$, la tâche est à *départ différé*. Lorsque toutes les tâches τ_k vérifient $r_k = 0$, le système de tâches est dit à *départs simultanés*.

Pour la validation d'un système de tâches, on ne s'intéresse pas directement au programme exécuté par chaque tâche, on ne prend en compte que sa durée d'exécution. Toutefois, cette durée est délicate à évaluer pour des raisons tant matérielles que logicielles [91]. Tout d'abord, elle dépend du matériel et notamment des mécanismes d'optimisation utilisés, par exemple : mémoire cache [107], pipeline [110], prédicteur de branchement [30]. Ensuite, la durée d'exécution d'un programme dépend de ses paramètres d'entrées : il n'est pas constant. Ceci est dû aux structures conditionnelles et aux boucles non-statiques. Ne pouvant pas prédire avec certitude la durée d'exécution d'une instance $\tau_{k,i}$, on utilise une durée correspondant au pire cas : Worst Case Execution Time (WCET), noté C_k.

Le paramètre D_k indique la durée maximale autorisée pour exécuter une instance de τ_k : si une instance de τ est activée à l'instant t, alors elle doit être terminée à l'instant $t + D_k$. Dans le cas où une instance ne respecte pas son échéance (elle se termine après $t + D_k$), une faute temporelle est commise, et le bon comportement du procédé n'est plus assuré.

Une même instance n'est pas parallélisable, on a donc $C_k \leq D_k$ pour chaque tâche $\tau_k \in \tau$. En fonction du rapport entre D_k et T_k, on dit que la tâche τ_k est :

- $D_k < T_k$: à échéances avant requête,
- $D_k = T_k$: à échéances sur requête,
- $D_k > T_k$: à échéances après requête.

Lorsqu'une tâche est à échéance après requête, plusieurs instances de cette tâche peuvent exister simultanément sans que les contraintes temporelles soient enfreintes, le programme associé à cette tâche doit alors être *réentrant*.

Les paramètres temporels d'une tâche τ_k permettent de caractériser le comportement de cette tâche. Pour décrire celui d'un système composé de tâches périodiques, on utilise les paramètres suivants :

- $r = \max\{r_1, \ldots, r_n\}$: la plus tardive date de première activation,

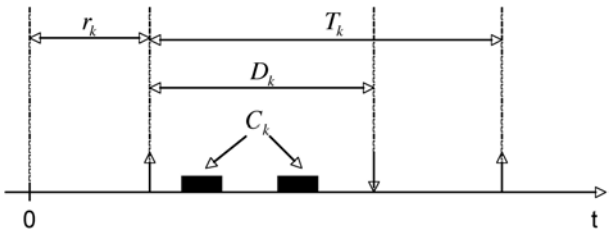


FIG. 1.2 – Paramètres temporels d'une tâche périodique

- $C = \max\{C_1, \ldots, C_n\}$: la plus grande durée d'exécution,
- $D = \max\{D_1, \ldots, D_n\}$: la plus grande échéance relative,
- $T = \max\{T_1, \ldots, T_n\}$: la plus grande période,
- $P = PPCM\{T_1, \ldots, T_n\}$: la métapériode.

Par abus de langage, nous appelons aussi métapériode tout intervalle temporel de la forme $[t, t + P[$. De plus, nous notons δ_k le nombre P/T_k d'instances de la tâche τ_k générées lors de chaque métapériode.

On associe aussi chaque instance $\tau_{k,i}$ avec ses propres paramètres temporels :

- $r_{k,i} = r_k + i.T_k$: date d'activation,
- $d_{k,i} = r_{k,i} + D_k$: échéance absolue.

Dans la suite, nous sommes souvent amenés à comparer les différentes instances d'une même tâche. Cependant, nous ne comparons ensemble que des instances dont les dates d'activation sont distantes d'un nombre entier de métapériodes. Par exemple, considérons une tâche de période 5 appartenant à un système de tâches dont la métapériode est 20. Alors, nous comparons ensemble la $3^{è}$, la $7^{è}$, la $11^{è}$ instances de cette tâche, mais jamais la $3^{è}$ avec la $8^{è}$. Dans la suite, nous utilisons cette propriété par une relation d'équivalence entre les instances.

Définition 1.1

- *Une instance $\tau_{k,i} \in \overline{\tau}$ est équivalente à une instance $\tau_{k',i'} \in \overline{\tau}$ à m métapériodes près, noté $\tau_{k,i} \equiv_m \tau_{k',i'}$, si et seulement si $k = k'$ et $i' - i = m.\delta_k$.*
- *Deux ensembles d'instances A et B sont équivalents à m métapériodes près si et seulement si les instances qu'ils contiennent sont deux à deux équivalentes à m métapériodes près.*

1.4.2 Tâches sporadiques

Les tâches sporadiques sont généralement représentées par trois paramètres temporels (voir figure 1.3) :

- $C_k \in \mathbb{N}$: pire durée d'exécution,
- $D_k \in \mathbb{N}$: délai critique,
- $T_k \in \mathbb{N}$: durée minimale entre deux activations successives.

Comme pour les tâches périodiques, on associe chaque instance $\tau_{k,i}$ d'une tâche sporadique avec ses propres paramètres temporels :

- $r_{k,i}$: date d'activation,
- $d_{k,i} = r_{k,i} + D_k$: échéance absolue.

Contrairement aux instances des tâches périodiques, les paramètres $r_{k,i}$ des instances de tâches sporadiques ne sont pas connus a priori : ils dépendent de la dynamique du procédé ainsi que de son environnement. Par contre, on sait que les dates d'activation respectent la propriété suivante :

$$\forall \tau_k \in \tau, \forall i \in \mathbb{N}, r_{k,i+1} - r_{k,i} \geq T_k$$

L'indéterminisme des dates d'activation rend la validation des systèmes de tâches sporadiques difficile. Sous certaines conditions, on peut réaliser une validation au pire cas : on suppose que les tâches sporadiques sont activées le plus souvent possible, on se ramène alors à un système

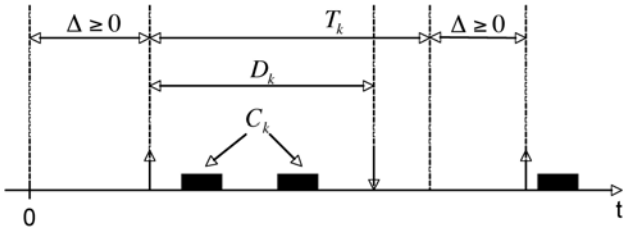


FIG. 1.3 – Paramètres temporels d'une tâche sporadique

de tâches périodiques. Cependant, une telle approche n'est pas toujours possible, notamment en contexte multiprocesseur [7]. En général, on peut difficilement garantir des échéances temporelles strictes pour les tâches sporadiques.

1.4.3 Tâches apériodiques

Les tâches apériodiques sont utilisées lorsque l'on ne peut pas garantir l'existence d'une durée minimale entre deux occurrences successives. Les difficultés de validation sont alors encore aggravées par rapport au cas des tâches sporadiques. En général, on ne peut pas garantir des échéances strictes pour ce type de tâches. Elles sont alors seulement utilisées dans le cadre de contraintes temporelles souples.

1.5 Interactions entre les tâches

1.5.1 Communication

Les différentes tâches composant un système temps-réel sont souvent amenées à échanger des informations. Pour cela, plusieurs méthodes de communication ont été développées. On distingue deux types de communication : synchrones (e.g. conversation téléphonique) ou asynchrones (e.g. courrier postal). Les communications asynchrones sont généralement implantées par des échanges de messages via des *boîtes aux lettres*, l'envoi et la réception d'un message n'étant pas simultanés. Lors d'une communication asynchrone, les messages peuvent être déposés ou retirés à tout moment. Lors d'une communication synchrone, les deux tâches concernées préparent la communication, et celle-ci est effectuée seulement lorsque les deux tâches sont prêtes. On implante généralement ce type de communication par des *rendez-vous*.

1.5.2 Synchronisation

La synchronisation peut être vue comme un cas particulier de communication où le seul fait de recevoir un message fournit toutes les informations nécessaires à son traitement. On parle alors d'évènement au lieu de message. Cette méthode est généralement utilisée dans le cas des systèmes temps-réel étudiés selon l'approche synchrone. Celle-ci étant guidée par les évènements, elle se prête bien à ce type de communication. La synchronisation est généralement implanté à l'aide de *contraintes de précédence* : une tâche doit attendre la terminaison d'une tâche antérieure avant de débuter son exécution. Les contraintes de précédence sont généralement représentées par un graphe orienté et acyclique.

1.5.3 Partage de ressources

Une ressource est considérée comme critique si elle ne peut pas être utilisée simultanément par plusieurs tâches. Une ressource peut être un composant matériel (e.g. périphérique, capteur, actionneur) ou logiciel (e.g. zone de mémoire partagée). On distingue généralement deux types d'accès : ceux en lecture qui consistent à récupérer une information, et ceux en écriture qui modifient l'état de la ressource. Naturellement, plusieurs accès simultanés en écriture ne sont pas autorisés, par contre, les accès en lecture ne modifiant pas l'état de la ressource, on peut généralement en réaliser plusieurs simultanément. Les portions de programme utilisant des ressources sont appelées des *sections critiques*.

Le mécanisme permettant de réguler les accès aux ressources est *l'exclusion mutuelle* des sections critiques utilisant des ressources communes. Il est généralement implémenté à l'aide de *sémaphores*. Chaque ressource est associée à un sémaphore. Avant d'utiliser une ressource, les tâches doivent faire une requête pour *prendre* le sémaphore correspondant. Si celui-ci est libre, alors la ressource est attribuée à cette tâche ; dans le cas contraire, la tâche est mise en attente jusqu'à ce que le sémaphore soit disponible. Après avoir utilisé une ressource, les tâches doivent *rendre*

le sémaphore. L'utilisation des sémaphores permet de modéliser un grand nombre d'interactions entre les tâches.

1.6 Exécutif temps-réel

L'exploitation des applications temps-réel requiert un système d'exploitation spécifique. Les *exécutifs temps-réel* se distinguent des autres systèmes par plusieurs caractéristiques. Tout d'abord, l'architecture même des applications temps-réel impose que ces systèmes soient multitâches : plusieurs processus peuvent être actifs simultanément et partagent l'accès au processeur. Ils doivent aussi fournir les services spécifiques aux applications temps-réel : communication, synchronisation, contrôle du temps, etc. Ensuite, ces systèmes doivent être *prédictibles*, propriété fondamentale puisqu'elle rend la validation possible. Pour qu'un système soit prédictible, la durée d'exécution des appels au noyau doit être connue, ou du moins prévisible. Il est alors possible d'évaluer la durée d'exécution des tâches, et par suite, de valider ou d'invalider un système temps-réel. Plusieurs exécutifs temps-réel ont été développés à ce jour. Certains sont spécifiques à un domaine d'application, par exemple OSEK/VDX [80] est très utilisé par l'industrie automobile. D'autres sont dédiés à un usage généraliste comme par exemple VxWorks [106], RT-Linux [95] et RTAI [94].

Les instances des tâches générées par l'arrivée des évènements sont implémentées au sein des exécutifs temps-réel par des processus. Ceux-ci sont caractérisés par des paramètres dynamiques, par exemple : durée avant échéance et allocation CPU déjà obtenue. Ces paramètres sont utilisés par l'ordonnanceur pour déterminer les processus qui doivent être exécutés en priorité.

Un processus peut se trouver dans différents états (voir figure 1.4) :

- *prêt* : le processus est prêt à être exécuté et attend l'accès à un processeur,
- *exécuté* : le processus est attribué à un processeur et est en cours d'exécution,
- *bloqué* : le processus est en attente d'un message, d'un évènement ou de l'accès à une ressource critique.

Un processus est *actif* lorsqu'il est soit prêt, soit en cours d'exécution, soit bloqué. Lorsqu'un processus est créé, il est dans l'état prêt. Lorsque l'ordonnanceur lui attribuera un processeur, il passera dans l'état exécuté. Lors de l'exécution d'un processus, plusieurs scénarios sont possibles. Premièrement, le processus peut atteindre la fin de son programme, il est alors détruit. Deuxièmement, l'ordonnanceur peut décider de lui retirer l'accès au processeur au profit d'un autre, le processus est alors suspendu et retourne dans l'état prêt : c'est une *préemption*. Troisièmement, le processus peut demander par exemple l'accès à une ressource, attendre un évènement. Si sa demande est satisfaite, alors il peut poursuivre son exécution, sinon il change d'état et devient bloqué. Lorsque sa demande pourra être satisfaite, alors il retournera à l'état prêt.

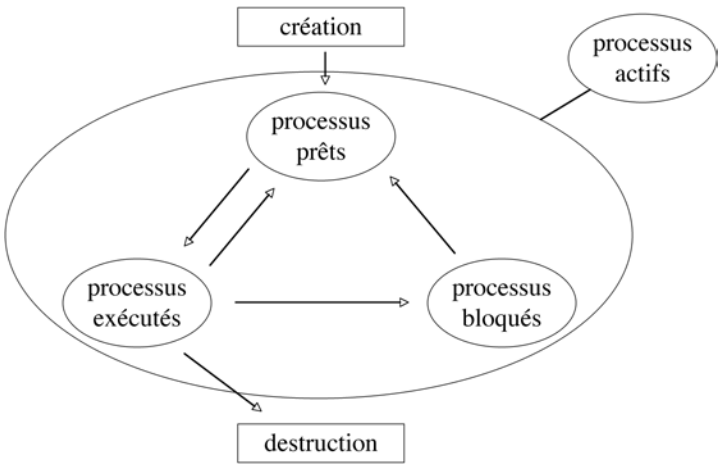


FIG. 1.4 – Evolution de l'état des processus

Certains exécutifs temps-réel ne permettent pas qu'une tâche soit suspendue au profit d'une autre, ces systèmes sont *non-préemptifs*. La gestion des processus s'en trouve simplifiée. Considérons par exemple un processus qui ne se met pas en attente lui-même (par l'attente d'un message ou d'un évènement par exemple). Lorsque l'ordonnanceur lui attribue le processeur, ce processus est alors exécuté sans interruption jusqu'à sa terminaison. En contexte monoprocesseur, cette propriété permet d'éviter de nombreux problèmes dus aux accès aux ressources critiques : inversion de priorités, interblocages, etc.

1.7 Architecture matérielle

Les applications temps-réel sont exécutées sur un support informatique. Sa structure détermine *l'architecture matérielle* du système temps-réel. On distingue généralement trois modèles d'architectures :

- monoprocesseur : un unique processeur est relié à une unique mémoire,
- multiprocesseur : plusieurs processeurs sont reliés par un bus, ils partagent une mémoire commune et peuvent communiquer en temps constant,
- distribué : plusieurs processeurs sont reliés par un réseau, leurs communications dépendent alors du débit du réseau, du trafic présent, des perturbations pouvant provoquer des erreurs de transmissions, etc.

Pour les architectures multiprocesseur et distribué, différentes hypothèses coexistent quant à l'attribution des processus aux processeurs. Dans le cas le plus simple, chaque tâche est attribuée statiquement à un processeur, ainsi toutes les instances d'une tâche sont exécutées sur le même processeur : c'est *l'approche partitionnée*. Une autre solution consiste à attribuer les processeurs aux instances plutôt qu'aux tâches, ainsi différentes instances d'une même tâche peuvent être exécutées sur des processeurs différents, c'est la *migration partielle*. Finalement, on peut aussi autoriser les instances à changer de processeurs en cours d'exécution, c'est la *migration totale*. Pour les architectures distribuées, on suit généralement l'approche partitionnée : seul le transit d'informations est autorisé sur le réseau.

Chapitre 2

Politiques d'ordonnancement

2.1 Classement des politiques d'ordonnancement

L'ordonnanceur d'un exécutif temps-réel détermine à chaque instant parmi les instances actives celles qui doivent être exécutées. Deux approches ont été suivies pour remplir cette fonction. L'ordonnancement *en-ligne* repose sur l'utilisation d'une *politique d'ordonnancement*. Son rôle est d'attribuer une priorité à chaque instance active, les processeurs étant ensuite attribués aux instances les plus prioritaires. Les décisions sont donc prises dynamiquement en fonction de l'état du système. L'ordonnancement *hors-ligne* est basé sur une *table d'ordonnancement* destinée à être utilisée par un *séquenceur*. A chaque instant, le séquenceur consulte la table d'ordonnancement pour choisir les instances qui doivent être exécutées. Pour ordonnancer des systèmes avec des interactions complexes, cette approche est intéressante puisque la table d'ordonnancement est définie et validée avant la mise en service de l'application temps-réel. Ainsi, de nombreuses techniques peuvent être utilisées pour constituer cette table. Toutefois, cette approche est moins souple que celle en-ligne puisque les décisions d'ordonnancement sont prises statiquement, le système est donc moins apte à réagir à une situation nouvelle, par exemple, l'arrivée imprévue d'une tâche sporadique.

De nombreuses méthodes peuvent être utilisées pour attribuer des priorités, et par suite, pour définir des politiques d'ordonnancement. On utilise ici la classification de [20] :

- PFX : politiques à priorités fixes, toutes les instances d'une même tâche partagent la même priorité qui n'évolue pas au cours du temps,
- PFI : politiques à priorités fixes par instances, chaque instance possède sa propre priorité qui n'évolue pas au cours du temps,
- DYN : politiques à priorités dynamiques, chaque instance a sa propre priorité et celle-ci peut évoluer au cours du temps.

Dans [21], une autre classe de politiques d'ordonnancement est introduite : la priorité à l'instant t d'une instance $\tau_{k,i}$ dans une séquence s est attribuée en fonction de seulement deux informations :

- sa charge déjà exécutée : $CE_s(\tau_{k,i}, t)$ (voir section suivante),
- la distance à son échéance : $d_{k,i} - t$.

Dans la suite, nous notons DCL cette classe de politiques d'ordonnancement. Ces quatre classes vérifient les relations suivantes :

$$\begin{cases} \text{PFX} \subset \text{PFI} \subset \text{DYN} \\ \text{DCL} \subset \text{DYN} \end{cases}$$

A notre connaissance, les classes PFI et DCL n'ont jamais été comparées à ce jour. Dans le chapitre 3, nous donnons quelques éléments de réponse. Les notions suivantes servent à comparer les politiques d'ordonnancement d'entre elles.

Un système de tâches est *ordonnançable* (ou *faisable*) par une politique d'ordonnancement si la séquence d'exécution produite par cette politique est valide (i.e. respecte toutes les contraintes du système). Par extension, un système de tâches est faisable s'il existe une politique pour laquelle il est ordonnançable. Une politique d'ordonnancement est *plus puissante* qu'une autre si elle produit

une séquence d'exécution valide pour tous les systèmes ordonnançables par l'autre.

Une politique d'ordonnancement est *optimale* si elle produit une séquence valide pour tous les systèmes ordonnançables. Plus précisément, une politique d'ordonnancement est optimale pour ordonnancer les systèmes appartenant à un ensemble E parmi les politiques appartenant à une classe C si et seulement si elle ordonnance tous les systèmes de E faisables par au moins une politique de C.

2.2 Séquences d'exécution

Une séquence d'exécution modélise l'exécution d'un système de tâches : à chaque instant, elle définit les instances qui disposent d'un processeur et donc qui sont en cours d'exécution. Dans la suite, nous manipulons les séquences d'exécution $(s_t)_{t\in\mathbb{N}}$ à l'aide des notations suivantes (voir figure 2.1) :

- s_t : ensemble des instances en cours d'exécution dans l'intervalle $[t, t+1[$,
- $s_t^\star$: ensemble des instances actives à l'instant t,
- $e_s(\tau_{k,i})$: date de terminaison de $\tau_{k,i}$,
- $TR_s(\tau_{k,i})$: temps de réponse de $\tau_{k,i}$,
- $CE_s(\tau_{k,i}, t)$: charge de $\tau_{k,i}$ exécutée dans l'intervalle $[0, t[$,
- $LX_s(\tau_{k,i}, t)$: laxité de $\tau_{k,i}$ à l'instant t - i.e. la durée maximale de suspension de $\tau_{k,i}$ sans dépasser son échéance.

Ces notions sont liées entre elles par les propriétés suivantes :

$$TR_s(\tau_{k,i}) = e_s(\tau_{k,i}) - r_{k,i}$$

$$CE_s(\tau_{k,i}, t) = \sum_{u=0}^{t-1} |s_u \cap \{\tau_{k,i}\}|$$

$$LX_s(\tau_{k,i}, t) = (d_{k,i} - t) - (C_k - CE_s(\tau_{k,i}, t))$$

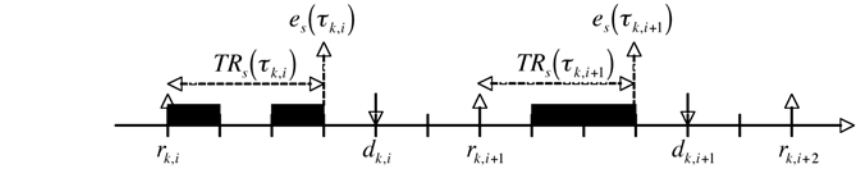
$$e_s(\tau_{k,i}) = t \Leftrightarrow [CE_s(\tau_{k,i}, t-1) = C_k - 1 \wedge CE_s(\tau_{k,i}, t) = C_k]$$

Une séquence d'exécution est *conservative* si aucun processeur n'est inactif dès qu'il existe une tâche active qui n'est pas déjà attribuée à un autre processeur. Formellement, on exprime cette propriété ainsi :

$$\forall t \in \mathbb{N}, |s_t| = min\{p, |s_t^\star|\}$$

Chaque tâche requiert périodiquement l'accès à un processeur. La charge engendrée par un système de tâches est généralement notée U et est définie par :

$$U = \sum_{\tau_k \in \tau} \frac{C_k}{T_k}$$



s_t	$\tau_{k,i}$	$\emptyset$	$\tau_{k,i}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\tau_{k,i+1}$	$\tau_{k,i+1}$	$\emptyset$	$\emptyset$	$\emptyset$	
s_t^*	$\tau_{k,i}$	$\tau_{k,i}$	$\tau_{k,i}$	$\emptyset$	$\emptyset$	$\emptyset$	$\tau_{k,i+1}$	$\tau_{k,i+1}$	$\tau_{k,i+1}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$CE_s(\tau_{k,i},t)$	0	1	1	2	2	2	2	2	2	2	2	2	2
$CE_s(\tau_{k,i+1},t)$	0	0	0	0	0	0	0	0	1	2	2	2	2

FIG. 2.1 – Notations relatives à une séquence d'exécution s

Pour une séquence d'exécution s, on distingue les notions suivantes (voir figure 2.2) :
- $U_s(t) = \sum_{\tau_{k,i} \in \overline{\tau} | r_{k,i} \leq t} C_k$: charge injectée dans l'intervalle $[0, t]$,
- $\bar{U}_s(t) = \sum_{u=0}^{t-1} |s_u|$: charge absorbée dans l'intervalle $[0, t[$,
- $\dot{U}_s(t) = U_s(t) - \bar{U}_s(t)$: charge restante à l'instant t.

Remarquons que $U_s(t)$ ne dépend pas de la séquence s mais seulement des paramètres temporels des tâches. Le système est *saturé* à l'instant t dans la séquence s si et seulement si on a $\bar{U}_s(t+1) - \bar{U}_s(t) = p$.

Lemme 2.1 *Soit s une séquence d'exécution d'un système de tâches périodiques. Pour tout $t \geq max\{r_k\}$, on a $U_s(t + P - 1) - U_s(t) = U.P$.*

Démonstration :
Soit s une séquence d'exécution et $t \geq max\{r_k\}$.
Pour chaque tâche $\tau_k \in \tau$, déterminons le nombre de $r_{k,i}$ appartenant à l'intervalle allant de t à $t + P - 1$. Ce nombre correspond au nombre d'instances de τ_k générées dans cet intervalle.
Soit $i \in \mathbb{N}$ tel que $r_{k,i} = min\{r_{k,j} | r_{k,j} \geq t\}$.
On a $r_{k,i+\delta_k} = r_{k,i} + \delta_k.T_k = r_{k,i} + P \geq t + P$.
Ainsi, l'instance $\tau_{k,i+\delta_k}$ est générée après $t + P - 1$.
On a aussi $r_{k,i-1} = r_{k,i} - T_k < t$ puisque $\tau_{k,i}$ est la première instance de τ_k générée dans cet intervalle. On obtient donc $r_{k,i-1+\delta_k} = r_{k,i} - T_k + P < t + P$.
Ainsi, l'instance $\tau_{k,i-1+\delta_k}$ est générée dans l'intervalle allant de t à $t + P - 1$.
Donc, le nombre d'instances de τ_k générées dans cet intervalle est δ_k.
On obtient alors :

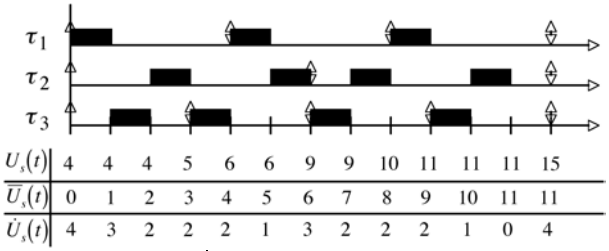
$$U_s(t + P - 1) - U_s(t) = \sum_{\tau_k \in \tau} \delta_k.C_k = P. \sum_{\tau_k \in \tau} C_k/T_k = P.U$$

CQFD.

2.3 Systèmes de tâches indépendantes

2.3.1 Rate Monotonic

La politique *Rate Monotonic* (RM) [96, 74] attribue aux tâches τ_k une priorité inversement proportionnelle à leur période $(1/T_k)$. Cette politique est à priorités fixes, et elle appartient donc à PFX. La figure 2.3 donne un exemple d'ordonnancement produit par RM. Cette politique est très intéressante dans certains contextes puisque l'on dispose du résultat d'optimalité suivant.



	0	1	2	3	4	5	6	7	8	9	10	11	12
$U_s(t)$	4	4	4	5	6	6	9	9	10	11	11	11	15
$\bar{U}_s(t)$	0	1	2	3	4	5	6	7	8	9	10	11	11
$\dot{U}_s(t)$	4	3	2	2	2	1	3	2	2	2	1	0	4

	r_k	C_k	D_k	T_k
τ_1	0	1	4	4
τ_2	0	2	6	6
τ_3	0	1	3	3

FIG. 2.2 – Evolution des charges injectée, absorbée et restante dans une séquence d'exécution s

Théorème 2.1 *[74] La politique RM est optimale parmi les politiques à priorités fixes pour ordonnancer, en contexte préemptif et monoprocesseur, les systèmes de tâches à départs simultanés et à échéances sur requête.*

Cet algorithme est aussi intéressant d'un autre point de vue, lorsque les tâches sont à départs différés, on dispose d'une condition suffisante à l'ordonnançabilité d'un système de tâches par RM.

Théorème 2.2 *[74] Un système de tâches à échéances sur requêtes est ordonnançable par RM en contexte préemptif et monoprocesseur si :*

$$U = \sum_{k=1}^{n} \frac{C_k}{T_k} \leq n(2^{1/n} - 1)$$

Le terme $n(2^{1/n} - 1)$ *tend vers* $ln(2) \approx 0,69$ *lorsque* n *tend vers l'infini.*

Cette condition est très intéressante puisqu'elle permet d'établir l'ordonnançabilité d'un système de tâches par RM en évaluant simplement sa charge, calcul dont la complexité est en $O(n)$. Cette condition n'est pas nécessaire, certains systèmes de tâches peuvent donc être ordonnançables par RM sans qu'elle soit vérifiée. [67] a déterminé expérimentalement la borne supérieure de la charge d'un système de tâches pour que celui-ci puisse être ordonnançable par RM, cette borne se situe autour de 88%. On peut alors constituer l'échelle suivante :

Taux de charge	$U \leq 0,69$	$0,69 < U \leq 0,88$	$U > 0,88$
Ordonnançable par RM	oui	possible	quasi-impossible

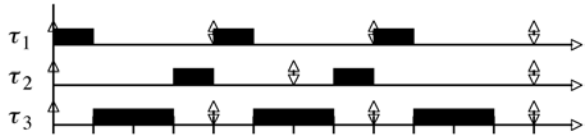
Lorsque les tâches sont à départs simultanés, [67] a établi une condition nécessaire et suffisante à l'ordonnançabilité d'un système de tâches par RM. Sa complexité est équivalente à réaliser une simulation d'une durée égale à la plus longue des périodes - i.e. de l'ordre de $O(n^2.\max_{\tau_k \in \tau}\{T_k\})$.

Théorème 2.3 *[67] Un système de tâches à échéances sur requête et à départs simultanés vérifiant* $T_1 \leq \cdots \leq T_n$ *est ordonnançable par RM en contexte préemptif et monoprocesseur si et seulement si :*

$$\forall k \in \{1, \ldots, n\}, \min_{t \in S_k} \sum_{i=1}^{k} \frac{C_i}{t} \left\lceil \frac{t}{T_i} \right\rceil \leq 1$$

où $S_k = \{i.P_j | 1 \leq j \leq k \wedge 1 \leq i \leq \lfloor \frac{T_k}{T_j} \rfloor\}\}$.

De nombreux autres résultats ont été obtenus sur RM, le livre traitant de la Rate Monotonic Analysis (RMA) [57] reste une référence pour ceux qui souhaitent utiliser ou étudier RM.



	r_k	C_k	D_k	T_k
τ_1	0	1	4	4
τ_2	0	1	6	6
τ_3	0	2	4	4

FIG. 2.3 – Exemple d'ordonnancement par RM

2.3.2 Deadline Monotonic

Pour ordonnancer les systèmes de tâches à échéances avant requête, on utilise généralement la politique *Deadline Monotonic* (DM) [71]. Elle attribue des priorités aux tâches τ_k qui sont inversement proportionnelles à leur échéance $(1/D_k)$. Cette politique est à priorités fixes et appartient donc à PFX. La figure 2.4 donne un exemple d'ordonnancement produit par DM ; remarquons que RM échoue à ordonnancer ce système de tâches. Comme pour RM, on dispose d'un résultat d'optimalité.

Théorème 2.4 *[71] La politique DM est optimale parmi les politiques à priorités fixes pour ordonnancer, en contexte préemptif et monoprocesseur, les systèmes de tâches à départs simultanés.*

Précisons toutefois que ce résultat d'ordonnançabilité concerne seulement les systèmes de tâches à échéances avant ou sur requête. Dans le cas où il existe une tâche τ_k vérifiant $D_k > T_k$, [68] a montré que DM n'est plus optimal. Plus généralement, les résultats que nous donnons dans la suite de cette section supposent tous que les tâches sont à échéances avant ou sur requête.

Comme pour RM, plusieurs tests d'ordonnançabilité existent pour DM. Dans [71], Leung and Whitehead ont montré que DM vérifie le théorème de l'instant critique (voir [74]), ainsi le résultat correspondant au théorème 2.2 s'applique aussi à DM.

Théorème 2.5 *Un système de tâches est ordonnançable par DM en contexte préemptif et monoprocesseur si :*

$$\sum_{k=1}^{n} \frac{C_k}{D_k} \leq n(2^{1/n} - 1)$$

Cette condition est facile à évaluer, cependant elle est très pessimiste. On utilise préférentiellement une condition obtenue par [69] à partir de celle du théorème 2.3.

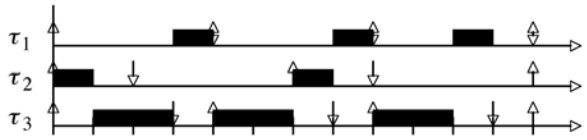
Théorème 2.6 *[69] Un système de tâches à départs simultanés vérifiant $D_1 \leq \cdots \leq D_n$ est ordonnançable par DM en contexte préemptif et monoprocesseur si et seulement si :*

$$\forall k \in \{1, \ldots, n\}, \min_{t \in S_k} \sum_{i=1}^{k} \frac{C_i}{t} \left\lceil \frac{t}{T_i} \right\rceil \leq 1$$

où $S_k = \{D_k\} \cup \{i.P_j | 1 \leq j \leq k \wedge 1 \leq i \leq \lfloor \frac{D_k}{T_j} \rfloor\}\}$.

2.3.3 Earliest Deadline First

La politique d'ordonnancement *Earliest Deadline First* (EDF) [96, 74] attribue des priorités dynamiques aux tâches : à chaque instant, la priorité des tâches est inversement proportionnelle à la durée les séparant de leur échéance. De manière équivalente, on peut dire que la priorité attribuée



	r_k	C_k	D_k	T_k
τ_1	0	1	4	4
τ_2	0	1	2	6
τ_3	0	2	3	4

FIG. 2.4 – Exemple d'ordonnancement par DM

à une instance $\tau_{k,i}$ est $1/d_{k,i}$. Cette politique appartient donc à PFI $\cap$ DCL. Elle a été inspirée par les travaux [55, 81, 32]. La figure 2.5 donne un exemple d'ordonnancement produit par EDF.

La politique EDF possède des propriétés remarquables pour l'ordonnancement en monoprocesseur. Tout comme RM et DM, elle a été beaucoup étudiée, [19, 104] constituent de bons ouvrages de référence. Le domaine d'optimalité de EDF a été étudié simultanément par [33, 61].

Théorème 2.7 *[33, 61] La politique EDF est optimale pour ordonnancer en contexte préemptif et monoprocesseur.*

On dispose aussi d'une condition nécessaire et suffisante à l'ordonnançabilité d'un système de tâches par EDF due à [74].

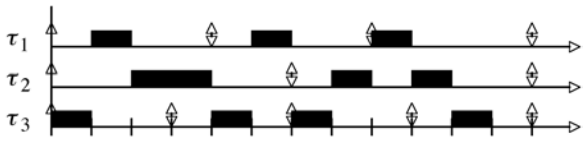
Théorème 2.8 *[74] Un système de tâches à échéances sur requête est ordonnançable par EDF en contexte préemptif et monoprocesseur si et seulement si :*

$$U = \sum_{k=1}^{n} \frac{C_k}{T_k} \leq 1$$

2.3.4 Least Laxity First

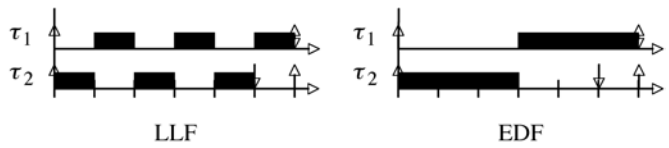
La politique d'ordonnancement *Least Laxity First* (LLF) [86, 34] est une politique à priorités dynamiques. A chaque instant, la priorité attribuée aux tâches est inversement proportionnelle à leur laxité ($1/LX_s(\tau_{k,i}, t)$). Cette politique appartient donc à DCL. Tout comme EDF, la politique LLF est optimale pour ordonnancer en monoprocesseur [86, 34]. Ainsi, la condition d'ordonnançabilité énoncée dans le théorème 2.8 reste valable pour LLF.

Théorème 2.9 *[86, 34] La politique LLF est optimale pour ordonnancer en contexte préemptif et monoprocesseur.*



	r_k	C_k	D_k	T_k
τ_1	0	1	4	4
τ_2	0	2	6	6
τ_3	0	1	3	3

FIG. 2.5 – Exemple d'ordonnancement par EDF



	r_i	C_i	D_i	T_i
τ_1	0	3	6	6
τ_2	0	3	5	6

FIG. 2.6 – Changements de contexte provoqués par EDF et LLF

Toutefois, la politique LLF vise une répartition équitable des accès au processeur tout en donnant la main en priorité aux tâches les plus urgentes - i.e. de plus faible laxité. La figure 2.6 montre que LLF provoque de nombreux changements de contexte inutiles, elle engendre ainsi un surcoût non négligeable. De plus, l'ordre des tâches les plus prioritaires peut changer à tout instant, les priorités doivent alors être évaluées en permanence. L'implémentation de cette politique est donc délicate. La tâche élue par EDF ne peut changer qu'à la terminaison ou à l'activation d'une tâche, le nombre de changements de contexte induits par EDF est donc beaucoup moins important qu'avec LLF. Pour toutes ces raisons, LLF est peu utilisée en pratique ; toutefois, comme nous le verrons dans la suite, elle offre de bonnes performances en environnement multiprocesseur.

2.4 Serveur de tâches sporadiques

Pour ordonnancer les tâches sporadiques, on peut les traiter en arrière plan des tâches périodiques : les instants auxquels aucune tâche périodique n'est active sont attribués à l'exécution des tâches sporadiques. Cette méthode a l'avantage de ne pas modifier le comportement des tâches périodiques, ainsi les politiques d'ordonnancement des tâches périodiques peuvent être utilisées dans ce contexte sans modification. Toutefois, elle ne permet pas de garantir des contraintes strictes sur les échéances des tâches sporadiques, cependant elle fonctionne correctement si la charge CPU n'est pas trop élevée. Pour remédier à ce problème, on utilise un *serveur de tâches sporadiques.*

Afin de réutiliser les résultats concernant l'ordonnancement des tâches périodiques, les serveurs de tâches sporadiques sont implémentés sous la forme d'une tâche périodique dont les paramètres temporels sont choisis en fonction des tâches sporadiques à exécuter. Le plus simple d'entre eux est le serveur à scrutation : à chaque activation du serveur, il traite les tâches sporadiques en attente, s'il n'y en a pas, il se suspend jusqu'à sa prochaine activation. Considérons une tâche sporadique τ_k. Pour assurer que cette tâche respecte ses échéances, les paramètres du serveur à scrutation associé τ_{serv} doivent vérifier :

- $C_{serv} \geq C_k$, la charge CPU du serveur permet au moins d'absorber celle de la tâche sporadique,
- $D_{serv} + T_{serv} \leq D_k$, quel que soit le moment où la tâche sporadique est activée, le serveur doit être réactivé suffisamment rapidement pour que son échéance soit inférieure ou égale à celle de la tâche sporadique.

Le serveur à scrutation présente un inconvénient : lorsqu'une tâche sporadique est activée juste après l'activation du serveur, elle doit attendre la période suivante du serveur avant d'être traitée, alors que le serveur peut être oisif. C'est ce fait qui induit la contrainte $D_{serv} + T_{serv} \leq D_k$ pour les paramètres temporels du serveur. Pour remédier à ce problème, [66] a proposé le *serveur ajournable.* Celui-ci conserve sa capacité d'exécution durant toute sa période, ainsi les tâches sporadiques activées après son activation peuvent être traitées sans attendre qu'il soit réactivé. Toutefois, cette règle déroge au principe de l'ordonnancement par priorité : toute tâche active et de plus forte priorité doit être exécutée sans attendre. Ce problème peut amener des tâches périodiques à manquer leur échéance bien que les conditions d'ordonnançabilité soient satisfaites.

Le *serveur sporadique* a été proposé par [99, 100] pour les politiques RM et DM, et il a été adapté à EDF par [102]. Il est basé sur le serveur ajournable mais le rechargement de la capacité

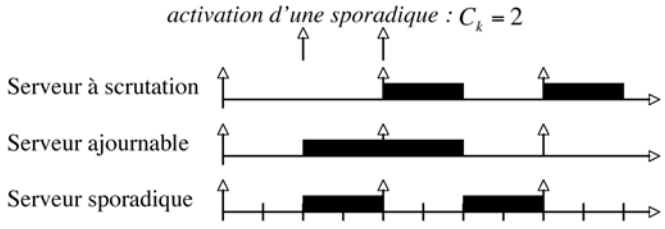


FIG. 2.7 – Différence entre les serveurs de tâches sporadiques

d'exécution du serveur n'est pas immédiat à sa réactivation, mais dépend des instants où il a utilisé le processeur lors de sa période précédente. Cette propriété permet d'assurer que le serveur sporadique respecte localement son taux de charge. Un serveur ajournable peut être exécuté d'une traite pendant au pire $2C_{spor}$ unités de temps (voir figure 2.7). C'est ce scénario qui peut conduire à des dépassements d'échéance pour les tâches périodiques. Dans chaque intervalle de longueur T_{serv}, le mécanisme de rechargement du serveur sporadique assure que celui-ci n'accède jamais plus de C_{serv} fois au processeur. En conséquence, les conditions d'ordonnançabilité des tâches périodiques restent valides lorsque l'on utilise un serveur sporadique.

2.5 Systèmes de tâches communicantes

Les politiques d'ordonnancement définies dans la section 2.3 sont conçues pour ordonnancer des systèmes de tâches indépendantes : toute interaction entre les tâches est proscrite. Naturellement, les tâches composant une application temps-réel ne sont jamais totalement indépendantes : soit elles communiquent entre elles, soit elles partagent des ressources communes. Toutefois, l'ordonnancement des tâches communicantes peut être réalisé à partir des politiques précédentes en transformant le système de tâches. En particulier, les synchronisations entre les tâches peuvent être traitées de la même manière.

Il existe une analogie profonde entre la communication et la notion de précédence. Prenons l'exemple d'une tâche τ_1 qui produit un résultat et qui le communique à une autre τ_2. Du point de vue de l'ordonnancement, on peut considérer que la tâche τ_1 *précède* la tâche τ_2, c'est-à-dire que l'exécution de la tâche τ_2 ne peut commencer que lorsque la tâche τ_1 est terminée. On note généralement $\tau_1 \succ \tau_2$ pour indiquer que τ_1 précède τ_2.

Le *découpage en forme normale* d'un système de tâches communicantes consiste à remplacer les communications par des relations de précédence en décomposant les tâches autour des communications de telle manière que les envois de messages soient toujours en fin de tâche et que les réceptions de messages soient toujours en début de tâche (voir figure 2.8).

Cette transformation permet de ramener le problème de l'ordonnancement des tâches communicantes à celui des tâches liées par des relations de précédence. Pour l'algorithme EDF, Blazewicz a montré comment modifier les dates de réveil et les échéances des sous-tâches obtenues lors du découpage en forme normale, pour que les relations de précédence soient implicitement vérifiées.

Théorème 2.10 *[16] La politique EDF est optimale pour ordonnancer en contexte préemptif et monoprocesseur des systèmes de tâches communicantes si après la décomposition en forme normale, les paramètres temporels sont attribués selon la règle suivante :*

- $r_{i,j} = \max\{r_i, \max_{\tau_{k,l} \succ \tau_{i,j}}\{D_{k,l} - C_k\}\}$, *en commençant par les tâches sans successeurs.*
- $D_{i,j} = \min\{D_i, \min_{\tau_{i,j} \succ \tau_{k,l}}\{r_{k,l} + C_k\}\}$, *en commençant par les tâches sans prédécesseurs.*

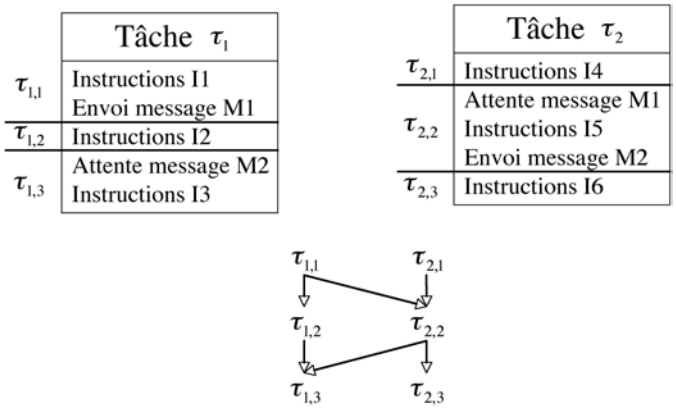


FIG. 2.8 – Exemple de découpage en forme normale

Cette méthode d'attribution des paramètres temporels aux sous-tâches peut être adaptée pour d'autres algorithmes d'ordonnancement, par exemple RM et DM. D'autres extensions ont aussi été développées, notamment lorsque les tâches communicantes n'ont pas la même période [92], ou pour traiter les relations de précédence avec des tâches apériodiques [28].

2.6 Protocoles de partage de ressources

2.6.1 Problèmes issus du partage de ressources

Le partage de ressources donne lieu à différents phénomènes pouvant nuire à l'ordonnancement d'un système de tâches. Le plus simple d'entre eux est *l'interblocage*. Considérons un système de tâches partageant deux ressources R1 et R2, et examinons la séquence d'exécution indiquée sur la figure 2.9. Lorsque la tâche τ_2 est activée, elle prend la ressource R1, ensuite elle est suspendue lorsque τ_1 est activée puisqu'elle est moins prioritaire. Celle-ci prend alors la ressource R2 et demande aussi R1. La ressource R1 étant déjà attribuée, la tâche τ_1 est donc bloquée. Alors, la tâche τ_2 accède à nouveau au processeur et fait une demande pour la ressource R2. Celle-ci étant déjà attribuée, la tâche τ_2 est aussi bloquée. Les tâches τ_1 et τ_2 se bloquent mutuellement pour accéder aux ressources R1 et R2 : c'est l'interblocage. Ce phénomène ne peut se produire qu'en présence d'appels imbriqués à plusieurs ressources. L'utilisation des protocoles de gestion de ressources permet d'éviter ce phénomène.

Lorsque les tâches partagent des ressources, une tâche peut être bloquée par d'autres moins prioritaires mais détenant des ressources. Ce phénomène peut amener la tâche la plus prioritaire à être exécutée en dernier : c'est *l'inversion de priorités*. Considérons un système de trois tâches partageant une ressource R. Supposons que les tâches sont ordonnées par ordre de priorités ($prio(\tau_1) > prio(\tau_2) > prio(\tau_3)$), et examinons la séquence d'exécution indiquée sur la figure 2.10. La tâche τ_3 commence son exécution en prenant l'accès à R. Lorsque τ_1 est activée, elle est exécutée jusqu'à ce qu'elle demande la ressource. A ce moment-là, elle est alors bloquée par τ_3, et doit attendre que τ_3 libère la ressource. L'arrivée de la tâche τ_2 montre que toutes les tâches de priorités intermédiaires (entre τ_1 et τ_3) peuvent retarder la libération de la ressource, et par suite, sont exécutées avant τ_1 qui est pourtant plus prioritaire : c'est l'inversion de priorités. Cet exemple montre que le temps de blocage d'une tâche ne dépend pas que des tâches plus prioritaires,

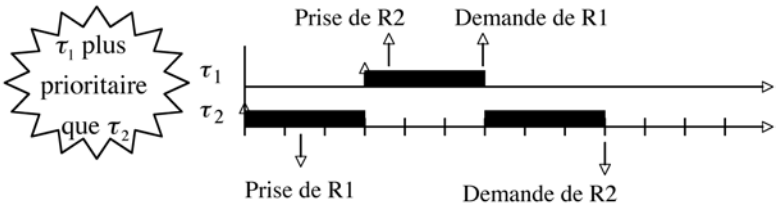


FIG. 2.9 – Exemple d'interblocage

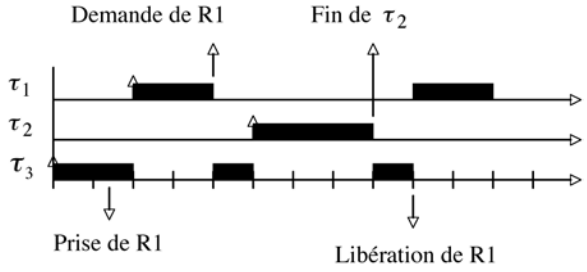


FIG. 2.10 – Exemple d'inversion de priorités

potentiellement, il peut dépendre de toutes les autres tâches. En général, on ne peut pas borner le temps de blocage d'une tâche. L'utilisation des protocoles de gestion de ressources permet de remédier à ce problème.

Une dernière particularité de l'ordonnancement en présence de partage de ressources concerne l'approximation réalisée sur la durée d'exécution des tâches. Le programme correspondant à chaque tâche est représenté par sa durée d'exécution, le paramètre temporel C_k indique la pire durée d'exécution (WCET). Certaines instances ont une durée d'exécution inférieure au WCET. Lorsque les tâches ne partagent pas de ressources, ce fait n'a pas d'incidence : tous les systèmes de tâches ordonnançables dans le pire cas (WCET) sont aussi ordonnançables en considérant les durées réelles d'exécution. L'exemple de la figure 2.11 montre que cette propriété n'est plus vraie lorsque les tâches partagent des ressources. Dans cet exemple, les trois tâches utilisent une ressource R tout du long de leur exécution : tout leur programme est en section critique. Dans l'ordonnancement de droite, la durée d'exécution de la tâche τ_1 est inférieure à son WCET. La tâche τ_3 peut alors débuter son exécution plus tôt que dans l'ordonnancement de gauche. Lorsque la tâche τ_2 est activée, elle est bloquée puisque τ_3 détient la ressource. Ne pouvant s'exécuter qu'après la tâche τ_3, elle manque alors son échéance. Cet exemple montre que lors de la validation d'un système de tâches partageant des ressources, on doit tenir compte du fait que le paramètre C_k est un majorant de la durée d'exécution des instances de τ_k.

Pour répondre aux problèmes dus au partage de ressources, plusieurs protocoles de gestion de ressources ont été développés. Dans les trois sections qui suivent, nous détaillons les principaux protocoles utilisés en temps-réel. L'utilisation de majorants des durées de blocage des tâches amène à des résultats pessimistes : les durées réelles de blocage étant généralement bien inférieures aux majorations. L'étude du temps de blocage moyen a été abordée dans [39, 58, 109, 38]. D'autres travaux suivant une approche probabiliste traitent aussi de ce problème, par exemple [31, 47, 77, 78, 79, 83, 84]. Les blocages entre les tâches sont représentés par des variables aléatoires, et plus précisément par des marches aléatoires. Les lois limites correspondantes sont alors caractérisées par des processus de diffusion. Les résultats obtenus sont donc beaucoup plus précis que ceux de [39, 58, 109, 38].

2.6.2 Priority Inheritance Protocol

Le protocole *Priority Inheritance Protocol* (PIP) [97, 98] permet de limiter le phénomène de l'inversion de priorités. Son principe est le suivant : lorsqu'une tâche détient une ressource, elle hérite la priorité des tâches bloquées par une demande d'accès à cette ressource. Reprenons l'exemple de la figure 2.10 en utilisant maintenant le protocole PIP (voir figure 2.12). On remarque que la tâche τ_2 ne peut plus préempter la tâche τ_3. Ainsi, les tâches de priorités intermédiaires (entre τ_1 et τ_3) ne peuvent plus retarder la tâche τ_1. En utilisant le protocole PIP, on peut alors borner le temps de blocage d'une tâche.

Toutefois, ce protocole présente plusieurs inconvénients. Lors de l'attente d'une ressource, une tâche peut être bloquée par plusieurs sections critiques successives, ainsi une tâche peut rester bloquée très longtemps, la borne du temps de blocage est donc généralement très importante. De plus, ce protocole ne permet pas d'éviter les interblocages, et il n'est utilisable qu'avec des

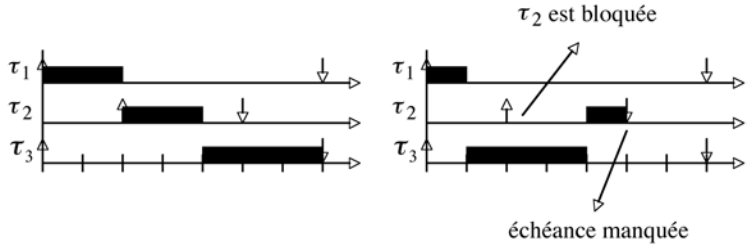


FIG. 2.11 – Influence de l'utilisation du WCET

politiques d'ordonnancement à priorités fixes.

2.6.3 Priority Ceiling Protocol

Le protocole *Priority Ceiling Protocol* (PCP) permet de prévenir les interblocages et de diminuer le temps de blocage maximum des tâches. Il a été initialement conçu pour les politiques d'ordonnancement à priorités fixes [98], et [27] l'a adapté pour fonctionner avec EDF. Le protocole PCP attribue une priorité plafond à chaque ressource, sa valeur est définie par la priorité maximale des tâches utilisant cette ressource. Lorsqu'une tâche demande l'accès à une ressource, le protocole PCP l'autorise à deux conditions :

- la ressource doit être disponible,
- la priorité de la tâche doit être strictement supérieure aux priorités plafonds des ressources en cours d'utilisation.

Lorsque l'accès à une ressource est refusé, la tâche demandant l'accès est bloquée, et toutes les tâches détenant des ressources de priorités plafonds supérieures à la sienne héritent sa priorité. Avec ce protocole, le temps de blocage maximal B_k d'une tâche τ_k est égal à la durée de la plus longue section critique des autres tâches. Plusieurs conditions suffisantes pour assurer l'ordonnançabilité d'un système de tâches partageant des ressources selon le protocole PCP ont été obtenues.

Théorème 2.11 *[98] Un système de tâches à échéances sur requête et partageant des ressources est ordonnançable par RM couplé avec PCP en contexte préemptif et monoprocesseur si la condition suivante est vérifiée :*

$$\forall k \in \{1, \ldots, n\}, \frac{B_k}{T_k} + \sum_{i=1}^{k} \frac{C_i}{T_i} \leq k.(2^{1/k} - 1)$$

Théorème 2.12 *[27] Un système de tâches à échéances sur requête et partageant des ressources est ordonnançable par EDF couplé avec PCP en contexte préemptif et monoprocesseur si la condition suivante est vérifiée :*

$$\sum_{k=1}^{n} \frac{C_i + B_i}{T_i} \leq 1$$

2.6.4 Stack Resource Protocol

L'utilisation de PCP avec des politiques à priorités dynamiques implique une implémentation assez lourde puisque les priorités plafonds des ressources utilisées doivent être recalculées à chaque instant. Le protocole *Stack Resource Protocol* (SRP) [10] est plus adapté que PCP aux politiques à priorités dynamiques. De plus, il est aussi utilisable avec des ressources multi-instances (e.g. du type écrivain/lecteur).

Le protocole SRP attribue à chaque tâche un niveau de préemption qui peut être choisi arbitrairement. Chaque ressource se voit attribuer une valeur plafond qui correspond au plus haut

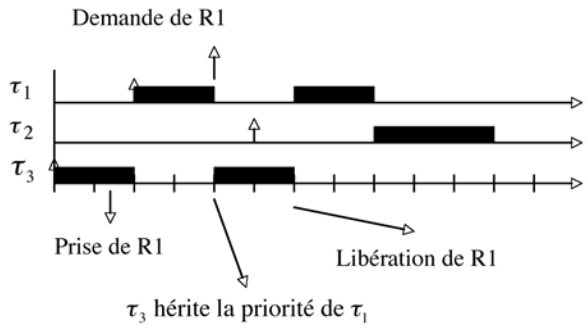


FIG. 2.12 – Exemple d'ordonnancement avec PIP

niveau de préemption des tâches actives requérant plus d'instances de cette ressource qu'il n'y en a de disponibles. La valeur plafond d'une ressource est donc dynamique, alors que les niveaux de préemption des tâches sont statiques. Le plafond système est défini par la plus grande valeur plafond des ressources. Avec le protocole SRP, une tâche τ_i préempte une tâche τ_j (accède prioritairement au processeur) si les trois conditions suivantes sont vérifiées :

- la priorité dynamique de τ_i est supérieure ou égale à celle de τ_j,
- le niveau de préemption de τ_i est supérieur à celui de τ_j,
- le niveau de préemption de τ_i est supérieur au plafond système.

Le protocole SRP présente les mêmes propriétés que PCP : évitement des interblocages, pas d'inversion de priorités, au plus un seul blocage infligé à chaque tâche. Avec ce protocole, une tâche préemptée par une autre ne peut pas à son tour la préempter. Ainsi, une tâche préemptée est placée au sommet d'une pile ; à la terminaison de la tâche en cours d'exécution, le processeur est attribué soit à une nouvelle tâche, soit à la tâche située au sommet de la pile. Cette propriété facilite l'implémentation du protocole SRP.

Lorsque le protocole SRP est utilisé avec EDF, [10] a obtenu une condition suffisante d'ordonnançabilité.

Théorème 2.13 *[10] Un système de tâches partageant des ressources est ordonnançable par EDF couplé avec SRP en contexte préemptif et monoprocesseur si la condition suivante est vérifiée :*

$$\forall k \in \{1, \ldots, n\}, \frac{B_k}{D_k} + \sum_{i=1}^{k} \frac{C_i}{D_i} \leq 1$$

Le protocole SRP a été adapté par [101] pour tenir compte des contraintes de précédence.

2.7 Stratégies de validation

Un système temps-réel est ordonnançable par une politique en-ligne si et seulement si l'ordre d'exécution des tâches choisi par la politique permet d'assurer que chaque tâche respecte son échéance. Pour évaluer cette propriété, plusieurs méthodes ont été proposées. [74] a proposé des conditions (théorèmes 2.2 et 2.8, respectivement pages 20 et 22) de complexité polynomiale par rapport au nombre de tâches qui permettent d'évaluer l'ordonnançabilité d'un système de tâches par RM et EDF. Toutefois, ces conditions s'appliquent seulement pour les systèmes de tâches à échéances sur requête et exécutés en contexte préemptif et monoprocesseur.

De nombreux travaux ont contribué à étendre leur contexte d'application. [50, 56, 67, 8] proposent une approche basée sur le calcul du pire temps de réponse des tâches. Cette méthode permet d'obtenir des conditions (théorèmes 2.3 et 2.6, respectivement pages 20 et 21) de complexité pseudo-polynomiale valables pour n'importe quelle politique à priorités fixes lorsque les tâches sont exécutées en contexte préemptif et monoprocesseur. Lorsque les tâches sont asynchrones - i.e. à aucun moment les tâches ne se réactivent toutes simultanément, ces conditions sont généralement pessimistes dans le sens où elles ne sont pas nécessaires, mais seulement suffisantes.

Lorsqu'aucune condition ne peut assurer l'ordonnançabilité d'un système de tâches par une politique, on recourt généralement à une simulation. Pour cela, on utilise un *intervalle de faisabilité* - i.e. si aucune échéance appartenant à cet intervalle n'est dépassée, alors le système temps-réel est ordonnançable. Cette méthode a été introduite par [70] pour EDF. Pour obtenir un intervalle de faisabilité, on suit généralement la démarche suivante. Tout d'abord, on montre que les séquences d'exécution sont cycliques. Ensuite, on borne la durée d'exécution nécessaire pour atteindre le régime cyclique ; on l'appelle la *durée de montée en charge.*

Théorème 2.14 *[70] Un système de tâches est ordonnançable par EDF en contexte préemptif monoprocesseur si et seulement si les deux conditions suivantes sont vérifiées :*

- *l'ordonnancement produit par simulation dans l'intervalle* $[0, r + 2P]$ *est valide,*
- *l'état des instances actives* $(CE_s(\tau_{k,i}, t))$ *à l'instant* $r + P$ *est le même qu'à l'instant* $r + 2P$.

Lorsque le système n'est pas surchargé ($U \leq 1$), [13] a montré que la deuxième condition est toujours vérifiée lorsque EDF est utilisé pour produire l'ordonnancement. Ce résultat permet donc de valider un système de tâches en utilisant une simulation, toutefois, la complexité résultante est exponentielle puisque la durée nécessaire dépend de la métapériode (P). Toujours pour EDF, [13] a réduit la durée de simulation à une durée pseudo-polynomiale pour les systèmes à départs simultanés, cette borne a encore été améliorée par [93, 111].

Théorème 2.15 *[13] Un système de tâches à départs simultanés vérifiant $U < 1$ est ordonnançable par EDF en contexte préemptif monoprocesseur si et seulement si toutes les échéances sont respectées dans l'intervalle :*

$$\left[0, \frac{U}{1-U} \max_{\tau_k \in \tau}\{T_k - D_k\}\right]$$

A notre connaissance, pour les systèmes de tâches à départs différés, le seul intervalle de faisabilité connu pour EDF est $[0, r+2P]$. Dans [21], cet intervalle de faisabilité est étendu à n'importe quelle politique d'ordonnancement appartenant à DCL.

Théorème 2.16 *[21] Soit π une politique d'ordonnancement appartenant à DCL. Un système de tâches est ordonnançable par π en contexte préemptif monoprocesseur si et seulement si toutes les échéances sont respectées dans l'intervalle :*

$$[0, r+2P]$$

Pour les politiques à priorités fixes, [42] propose un intervalle de faisabilité spécifique. Cet intervalle dépend de l'ordre de priorités des tâches et varie donc en fonction de la politique d'ordonnancement utilisée. Toutefois, la durée de simulation correspondante reste de complexité exponentielle (proportionnelle à la métapériode P).

Théorème 2.17 *[42] Soient $(S_k)_{k\in\{1,\ldots,n\}}$ et $(X_k)_{k\in\{1,\ldots,n\}}$ les suites définies par :*

$$\begin{cases} S_1 = r_1 \\ S_k = \max\{r_k, r_k + \left\lceil \frac{S_{k-1}-r_k}{T_k} \right\rceil T_k\} \end{cases} \quad \begin{cases} X_n = S_n \\ X_k = r_k + \left\lfloor \frac{X_{k+1}-r_k}{T_k} \right\rfloor T_k \end{cases}$$

L'intervalle $[X_1, S_n + P]$ est un intervalle de faisabilité du système de tâches τ ordonnancé en contexte préemptif et monoprocesseur par les politiques à priorités fixes vérifiant :

$$prio(\tau_1) > \cdots > prio(\tau_n)$$

La notion d'intervalle de faisabilité utilisée par [42] est plus puissante que celle que nous avons présentée : il n'est pas nécessaire de tenir compte des instances activées avant l'intervalle de faisabilité. On peut alors considérer que l'instant $t = X_1$ est l'origine de l'ordonnancement.

2.8 Ordonnancement multiprocesseur

Ces dernières années, l'étude des systèmes multiprocesseurs s'est intensifiée en raison du développement de ces architectures et aussi des besoins croissants en terme de puissance des systèmes temps-réel. Deux approches ont été suivies. L'approche partitionnée consiste à répartir statiquement les tâches entre les processeurs, ainsi la théorie de l'ordonnancement monoprocesseur s'applique sur chaque processeur indépendamment des autres. Cependant, la répartition des tâches sur les différents processeurs est un problème NP [40, 20], ainsi la difficulté n'est que déplacée. L'approche globale considère qu'une même tâche peut être allouée à n'importe quel processeur, sous l'hypothèse de migration totale, une tâche peut même changer de processeur en cours d'exécution. Toutefois, les approches globales et partitionnées sont complémentaires pour l'ordonnancement en priorités fixes [71] : il existe des systèmes de tâches ordonnançables en priorités fixes seulement par l'une des deux approches.

Cependant, la théorie de l'ordonnancement monoprocesseur ne s'applique pas à l'approche globale. Nous montrons par un exemple dans le chapitre 3 que l'intervalle de faisabilité donné dans [21] n'est plus valide pour des systèmes de tâches ordonnançés par EDF ou LLF en multiprocesseur. D'autres "anomalies" ont aussi été relevées, par exemple [7, 44, 17]. Elles montrent que la théorie de l'ordonnancement multiprocesseur ne peut pas être une simple extension du cas monoprocesseur. En particulier, il n'existe pas de politique d'ordonnancement en-ligne optimale dans le contexte multiprocesseur [34]. Ce résultat marque la frontière entre l'ordonnancement monoprocesseur et multiprocesseur. De plus, la plupart des problèmes d'ordonnancement en multiprocesseur sont NP-complets, alors qu'en monoprocesseur, il existe des algorithmes polynomiaux dans de nombreux cas.

Certains travaux adaptent au cas multiprocesseur les conditions d'ordonnançabilité établies dans le contexte monoprocesseur, par exemple [90, 11, 76, 5, 12] : les conditions obtenues sont généralement très pessimistes. D'autres travaux consistent à déterminer un intervalle de faisabilité pour les ordonnancements multiprocesseurs. A notre connaissance, seules les politiques à priorités fixes ont été étudiées à ce jour. En particulier, aucun intervalle de faisabilité n'est connu pour des politiques à priorités dynamiques comme EDF ou LLF.

La cyclicité des ordonnancements préemptifs et multiprocesseurs produits par les politiques à priorités fixes a été abordée pour la première fois dans [22]. Les ordonnancements biprocesseurs sont étudiés, et [22] montre qu'ils sont cycliques de période P. La durée d'exécution avant l'entrée dans le régime cyclique est caractérisée par la date t_c du dernier temps creux acyclique - i.e. le dernier instant t où un processeur est oisif en t et actif en $t+P$, s'il n'en existe pas, on pose $t_c = -1$.

Théorème 2.18 *[22] Les ordonnancements produits par les politiques à priorités fixes en contexte préemptif et biprocesseur sont cycliques de période P au plus tard à partir de l'instant $t_c + 1$.*

Parallèlement, le théorème 2.17 établi par [42] a été généralisé au cas multiprocesseur par [24].

Théorème 2.19 *[24] Soient $(S_k)_{k\in\{1,\ldots,n\}}$ et $(X_k)_{k\in\{1,\ldots,n\}}$ les suites définies de la même manière que dans le théorème 2.17. L'intervalle $[X_1, S_n + P]$ est un intervalle de faisabilité du système de tâches τ ordonnancé en contexte préemptif et multiprocesseur par les politiques à priorités fixes vérifiant :*

$$prio(\tau_1) > \cdots > prio(\tau_n)$$

La méthode utilisée dans [42, 24] a été adaptée aux systèmes de tâches à échéances après requête. Un intervalle de faisabilité pour ces systèmes de tâches est proposé dans [25].

Théorème 2.20 *[25] Soient τ un système de tâches à échéances après requête et $(S_k)_{k\in\{1,\ldots,n\}}$ la suite définie par :*

$$\begin{cases} S_1 = r_1 \\ S_k = PPCM(T_1, \ldots, T_k) + \max\{r_k, r_k + \left\lceil \frac{S_{k-1} - r_k}{T_k} \right\rceil T_k\} \end{cases}$$

L'intervalle $[0, S_n + P]$ est un intervalle de faisabilité du système de tâches τ ordonnancé en contexte préemptif et multiprocesseur par les politiques à priorités fixes vérifiant :

$$prio(\tau_1) > \cdots > prio(\tau_n)$$

2.9 Approches hors-ligne

On distingue généralement plusieurs types d'approches hors-ligne. Les méthodes exactes reviennent généralement à réaliser un parcours exhaustif de toutes les solutions possibles. Dans le cadre de l'ordonnancement, le problème à résoudre étant NP-dur, toutes les méthodes exactes ont une complexité exponentielle. Ces méthodes sont généralement basées sur des formalismes spécifiques. Plusieurs travaux ont utilisé les réseaux de Petri, par exemple [18, 45, 72]. D'autres sont

basés sur les automates finis [2, 3, 4, 63], ou encore sur la géométrie discrète [62, 64]. Pour les systèmes de tâches ordonnançables, ces méthodes permettent d'exhiber un ordonnancement valide destiné à être implanté dans un séquenceur. Elles sont généralement intéressantes pour la richesse des interactions entre les tâches que l'on peut prendre en compte. Malheureusement, leur complexité est tellement importante qu'elles ne sont pas utilisables avec des systèmes temps-réel de taille industrielle.

Pour obtenir des solutions en un temps raisonnable, des méthodes approchées ont été appliquées au problème de l'ordonnancement. Elles permettent d'obtenir une solution approchée d'un problème NP en un temps polynomial. Ces techniques sont basées sur le recuit simulé, la méthode tabou, les algorithmes génétiques[41] et les processus de décision markoviens[89]. A notre connaissance, seuls les algorithmes génétiques ont été appliqués au problème de la validation des applications temps-réel, par exemple dans [87]. Les articles [53, 54] présentent une comparaison entre ces méthodes appliquées au problème de l'allocation de fréquence ; [65] propose une application des processus de décision markoviens au problème de la planification de trajectoire en robotique.

Finalement, on peut distinguer une troisième catégorie de méthodes hors-ligne : celles basées sur le calcul hors-ligne de paramètres d'ordonnancement en-ligne. L'exemple le plus représentatif de cette catégorie est l'algorithme d'Audsley pour attribuer des priorités fixes [9]. Le principe de ces méthodes est de déterminer à l'avance certaines caractéristiques du mécanisme d'ordonnancement : certaines décisions sont prises hors-ligne, et d'autres en-ligne. Ces caractéristiques peuvent être la priorité de certaines tâches, ou la date d'activation de certaines instances, etc. Cette approche permet d'augmenter la puissance d'ordonnancement des méthodes en-ligne sans tomber dans la rigidité généralement imposée par les méthodes hors-ligne.

Deuxième partie

Contributions au problème de la cyclicité

Chapitre 3

Préambule

Dans ce chapitre, nous donnons tout d'abord de nouveaux exemples pour mettre en évidence les différences entre l'ordonnancement en contexte monoprocesseur et celui en contexte multiprocesseur. Nous nous plaçons ici dans le cadre des exécutifs préemptifs avec migration totale. Dans ce contexte, des exemples [7, 44, 17] ont déjà permis de constater certaines différences. Ceux que nous présentons ici permettent de faire ressortir les points suivants :

- la borne de la montée en charge - i.e. la durée avant qu'un ordonnancement soit cyclique, n'est pas $r + P$ pour EDF et LLF en multiprocesseur,
- $[0, r + 2P]$ n'est pas un intervalle de faisabilité pour EDF en multiprocesseur.

L'étude de la cyclicité des séquences d'exécution est l'un des thèmes de recherche traités au LISI. Des exemples similaires ont aussi été proposés par Mme Choquet-Geniet. Les constats que nous faisons ici sont en partie issus des discussions que nous avons eues ensemble.

Dans ce chapitre, nous faisons ensuite ressortir l'une des particularités de la classe de politiques d'ordonnancement PFI : il existe des séquences valides produites par ces politiques qui sont acycliques. Durant nos travaux, nous avons étudié cette classe de politiques à plusieurs reprises. Nous avons été amenés à distinguer deux sous-classes que nous présentons à la fin de ce chapitre.

3.1 Borne de la durée de montée en charge

La durée de montée en charge correspond à la durée d'exécution nécessaire pour atteindre la partie cyclique d'un ordonnancement. Il est montré dans [21] que cette durée est bornée par $r + P$ pour les séquences d'exécution monoprocesseur engendrées par une politique d'ordonnancement appartenant à DCL. Nous montrons ici que cette borne n'est plus valide dans le cas multiprocesseur à l'aide de deux exemples de séquences d'exécution, l'une produite par EDF, et l'autre par LLF. Pour EDF, [17] donne déjà un un tel exemple.

Exemple 3.1 *Considérons le système de tâches suivant :*

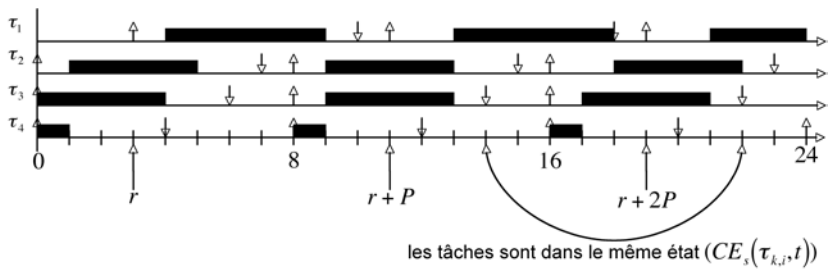


FIG. 3.1 – Ordonnancement biprocesseur du système de tâches de l'exemple 3.1 par EDF

	r_k	C_k	D_k	T_k
τ_1	3	5	7	8
τ_2	0	4	7	8
τ_3	0	4	6	8
τ_4	0	1	4	8

La figure 3.1 présente la séquence d'exécution biprocesseur produite par EDF pour ce système de tâches (lorsque deux instances ont la même priorité, on choisit toujours celle de plus faible indice). Le régime cyclique débute à $t = 14$, il commence donc après $r + P = 11$.

Exemple 3.2 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k
τ_1	0	2	4	8
τ_2	1	5	6	8
τ_3	0	3	7	8
τ_4	5	5	6	8

La figure 3.2 indique la séquence d'exécution biprocesseur produite par LLF pour ce système de tâches (lorsque deux instances ont la même priorité, on choisit toujours celle de plus faible indice). Le régime cyclique débute à $t = 15$, il commence donc après $r + P = 13$.

Dans ces deux exemples, on constate que la durée de montée en charge est supérieure à la borne établie dans le cas monoprocesseur. On remarque aussi que la borne est dépassée, mais de peu. Nous les avons choisis ainsi pour pouvoir les présenter facilement. Considérons le système de tâches $\{\tau_1 = (4, 8, 20, 20), \tau_2 = (3, 9, 20, 20), \tau_3 = (1, 9, 20, 20), \tau_4 = (11, 12, 20, 20)\}$ ordonnancé par EDF en biprocesseur. La séquence d'exécution obtenue est valide et est cyclique à partir de $t = 139$, soit après $r + 6P$. Ainsi, le nombre de métapériodes nécessaire pour atteindre le régime cyclique peut être très grand, relativement à la taille du système de tâches.

3.2 Intervalle de faisabilité

Un intervalle de faisabilité est un intervalle temporel tel que si au moins une échéance est manquée durant toute la vie d'un système temps-réel, alors il existe au moins une échéance manquée dans cet intervalle. Un tel intervalle permet donc de décider l'ordonnançabilité d'un système de tâches en ne considérant qu'une partie finie de la séquence d'exécution, par exemple par une simulation.

L'intervalle $[0, r+2P]$ est un intervalle de faisabilité pour les séquences d'exécution en monoprocesseur produites par des politiques d'ordonnancement appartenant à DCL [21]. Nous exhibons ici une séquence d'exécution biprocesseur produite par EDF où la première échéance manquée arrive après $r + 2P$, ce qui établit que $[0, r + 2P]$ n'est pas un intervalle de faisabilité en multiprocesseur.

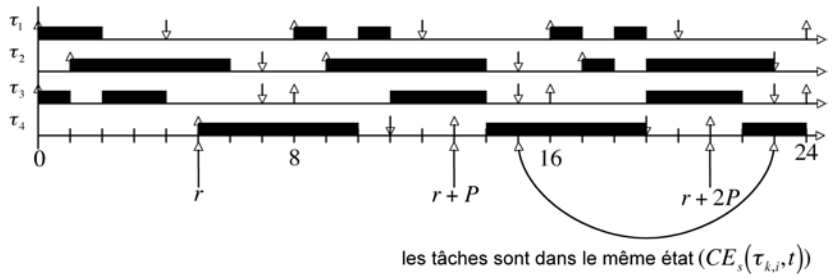


FIG. 3.2 – Ordonnancement biprocesseur du système de tâches de l'exemple 3.2 par LLF

Exemple 3.3 *Reprenons le système de tâches de l'exemple 1 en incrémentant C_4 d'une unité :*

	r_k	C_k	D_k	T_k
τ_1	3	5	7	8
τ_2	0	4	7	8
τ_3	0	4	6	8
τ_4	0	2	4	8

La figure 3.3 fournit la séquence d'exécution biprocesseur produite par EDF pour ce système de tâches (lorsque deux instances ont la même priorité, on choisit toujours celle de plus faible indice). A l'instant $t = 26$, la $3^{\text{è}}$ instance de τ_1 manque son échéance. Or, l'intervalle de faisabilité en monoprocesseur pour ce système de tâches est $[0, 19]$.

3.3 Propriétés propres à PFI

Les séquences d'exécution monoprocesseur produites par des politiques appartenant à DCL sont cycliques de période P [21]. L'exemple suivant montre qu'il existe des séquences valides d'exécution monoprocesseur produites par une politique appartenant à PFI qui sont cycliques de période $2P$ et non de période P.

Exemple 3.4 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k
τ_1	0	1	2	2
τ_2	0	1	2	2

Et la méthode d'attribution des priorités suivantes :

	priorité
$\tau_{1,i}$	$i \bmod 2$
$\tau_{2,i}$	$(i+1) \bmod 2$

où $x \bmod y$ est le reste de la division euclidienne de x par y.

La figure 3.4 fournit la séquence d'exécution que l'on obtient en monoprocesseur. La méta-période est $P = PPCM(2,2) = 2$. Or, la séquence n'est pas périodique de période 2, mais de période $4 = 2P$.

L'exemple suivant montre qu'il existe des séquences d'exécution valides en monoprocesseur produites par une politique appartenant à PFI qui ne sont pas cycliques. Un exemple ayant des propriétés similaires est présenté dans [43], il est construit à partir de EDF et de la suite des nombres premiers. Ici, nous utilisons les décimales de π.

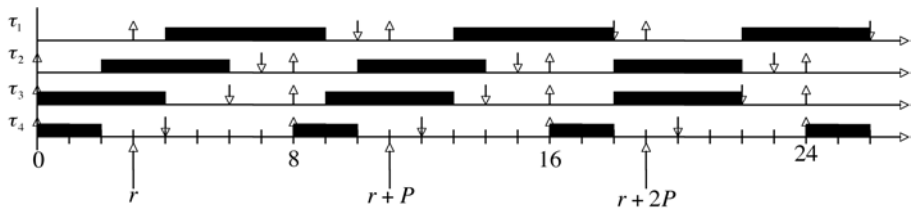


FIG. 3.3 – Ordonnancement biprocesseur du système de tâches de l'exemple 3.3 par EDF

FIG. 3.4 – Exemple de séquence valide cyclique de période 2P

Exemple 3.5 *Considérons la suite $(\pi_k)_{k\in\mathbb{N}}$ des décimales de π, i.e. $\pi_0 = 3$, $\pi_1 = 1$, $\pi_2 = 4$, etc. Considérons maintenant, un système formé de 10 tâches périodiques ayant les paramètres temporels suivants :*

$$\forall \tau_k \in \tau, r_k = 0, C_k = 1, D_k = T_k = 10$$

Quelle que soit l'attribution des priorités, ce système de tâches est ordonnançable en monoprocesseur (dès que la séquence est conservative). Etudions sa cyclicité lorsque les priorités sont attribuées de la manière suivante :

$$\forall \tau_{k,i} \in \overline{\tau}, prio(\tau_{k,i}) = (k + \pi_i) \bmod 10$$

Pour la $i^{\text{è}}$ métapériode, l'ordre dans lequel sont exécutées les tâches dépend de la $i^{\text{è}}$ décimale de π. Donc, les portions de séquences correspondant à deux métapériodes différentes i et j sont identiques si et seulement si la $i^{\text{è}}$ décimale de π est la même que la $j^{\text{è}}$ décimale de π. Or, les décimales de π n'admettent aucun cycle [73], il en est alors de même pour la séquence d'exécution engendrée par cette méthode d'attribution des priorités. Il existe donc des séquences valides acycliques engendrées par une politique d'ordonnancement appartenant à PFI.

Ces deux exemples montrent qu'il existe des séquences d'exécution valides produites par des politiques appartenant à PFI qui ne sont pas cycliques de période P. Il en est donc de même pour les politiques à priorités dynamiques (DYN). On peut aussi en déduire que PFI n'est pas inclus dans DCL, puisque toutes les séquences valides monoprocesseurs de DCL sont cycliques de période P [21]. D'autre part, DCL n'est pas inclus dans PFI puisque LLF n'appartient pas à PFI. On a donc :

$$PFI \not\subset DCL \text{ et } DCL \not\subset PFI$$

Remarquons finalement que la politique d'ordonnancement que nous avons construite dans l'exemple 3.5 est déterministe : il n'y a jamais d'ambiguïté lors de la génération de la séquence, autrement dit, on ne peut lui faire correspondre qu'une unique séquence.

3.4 Sous-classes de PFI : $\overline{\text{PFI}}$ et $\overline{\overline{\text{PFI}}}$

Dans la section 2.1, nous avons présenté plusieurs classes de politiques d'ordonnancement. Lors de nos travaux, nous avons aussi étudié deux sous-classes de PFI :

– $\overline{\text{PFI}}$: les relations de priorité entre les instances sont préservées entre deux métapériodes :

$$prio(\tau_{k,i}) > prio(\tau_{k',i'}) \Rightarrow \forall j \in \mathbb{Z}, prio(\tau_{k,i+j\delta_k}) > prio(\tau_{k',i'+j\delta_{k'}})$$

– $\overline{\overline{\text{PFI}}}$: les priorités de deux instances équivalentes à j métapériodes près sont identiques :

$$\forall \tau_{k,i} \in \overline{\tau}, \forall j \in \mathbb{Z}, prio(\tau_{k,i}) = prio(\tau_{k,i+j\delta_k})$$

Ces deux classes sont des sous-ensembles de PFI, plus précisément, on a :

$$\overline{\overline{\text{PFI}}} \subset \overline{\text{PFI}} \subset \text{PFI}$$

Dans [20], Carpenter et al. montrent que EDF appartient à PFI et que LLF n'appartient pas à PFI. D'autre part, EDF et LLF appartiennent toutes deux à DCL. Etudions maintenant l'appartenance de EDF à $\overline{\text{PFI}}$.

Théorème 3.1 *La politique EDF appartient à $\overline{PFI}$.*

Démonstration :
Considérons l'attribution de priorités suivante :

$$prio(\tau_{k,i}) = \frac{1}{d_{k,i} + 1/(k+1)}$$

Cette attribution donne les mêmes priorités que EDF, elle intègre en plus la règle permettant de départager les instances de même échéance par l'indice de la tâche. Cette méthode d'attribution appartient à PFI. On obtient aussi :

$$\begin{aligned} prio(\tau_{k,i}) > prio(\tau_{k',i'}) \Rightarrow & \quad d_{k,i} + 1/(k+1) + jP < d_{k',i'} + 1/(k'+1) + jP \\ \Rightarrow & \quad d_{k,i+j\delta_k} + 1/(k+1) < d_{k',i'+j\delta_{k'}} + 1/(k'+1) \\ \Rightarrow & \quad prio(\tau_{k,i+j\delta_k}) > prio(\tau_{k',i'+j\delta_{k'}}) \end{aligned}$$

Cette méthode d'attribution appartient donc à $\overline{\text{PFI}}$, et donc EDF $\in \overline{\text{PFI}}$.

CQFD.

Dans le chapitre 8, nous étudions le positionnement de EDF par rapport à $\overline{\overline{\text{PFI}}}$.

Chapitre 4

Cyclicité des ordonnancements de tâches périodiques

4.1 Introduction

La cyclicité des séquences d'exécution est une propriété importante pour calculer des intervalles de faisabilité. Les études précédentes portant sur la cyclicité traitent généralement d'une politique d'ordonnancement particulière ou d'une classe particulière de politiques d'ordonnancement. Elles sont aussi généralement propres à un exécutif temps-réel particulier. Par exemple, [70, 13] montrent que $[0, r + 2P]$ est un intervalle de faisabilité pour EDF en contexte monoprocesseur et préemptif avec migration totale. Ce résultat a été étendu à toute politique appartenant à DCL par [21]. Peu de travaux traitent du cas multiprocesseur. Pour les politiques à priorités fixes, [22, 24, 25] montrent que les séquences d'exécution produites sont cycliques, ils fournissent aussi un intervalle de faisabilité. A notre connaissance, pour les politiques qui ne sont pas à priorités fixes comme EDF et LLF, aucun intervalle de faisabilité n'est connu en multiprocesseur.

On peut distinguer un grand nombre de classes d'ordonnanceurs en fonction des informations et des méthodes utilisées pour attribuer les priorités : par exemple, les paramètres temporels des tâches, l'état dynamique des instances actives (charge absorbée, distance à l'échéance), l'indice des tâches, le numéro des instances, la durée écoulée depuis le début de l'exécution. Pour chacune d'elles, le problème de la cyclicité des séquences d'exécution produites doit être étudié. Toutefois, les exemples précédents montrent que cette étude est vouée à l'échec dans PFI et donc aussi dans DYN. Les deux contextes les plus généraux que l'on puisse alors considérer pour cette étude sont $\overline{\text{PFI}}$ et DCL. Malheureusement, rien ne prouve que l'une de ces deux classes englobe l'autre, et l'on est alors amené à développer plusieurs analyses. A cela s'ajoute la diversité des exécutifs temps-réel, des interactions entre les tâches, etc.

Une autre approche consiste à énoncer des hypothèses portant sur les propriétés des séquences plutôt que sur les mécanismes utilisés pour les engendrer. Si ces hypothèses sont suffisamment générales, elles permettront d'englober les séquences engendrées par un grand nombre de politiques d'ordonnancement. C'est l'approche que nous suivons ici. Nos hypothèses initiales portent uniquement sur le temps de réponse des instances. Notre objectif est de fournir une méthode la plus générale possible pour aborder le problème de la cyclicité des séquences d'exécution. Notre démarche se déroule en deux temps :

1. tout d'abord, nous établissons un certain nombre de résultats concernant les séquences d'exécution qui satisfont nos hypothèses de travail,
2. ensuite, nous étudions les politiques d'ordonnancement qui engendrent des séquences d'exécution satisfaisant nos hypothèses de travail.

Dans la section 4.2, nous présentons nos hypothèses de travail en discutant nos choix et en les justifiant par des exemples. Dans la section 4.3, nous démontrons nos principaux résultats.

Et dans la section 4.4, nous montrons que notre étude s'applique à une large classe de politiques d'ordonnancement englobant par exemple RM, DM et EDF. Finalement, nous présentons certaines perspectives possibles.

4.2 Contexte d'étude

Nous ne traitons que les systèmes composés de tâches périodiques. Nous ne faisons aucune hypothèse sur les échéances des tâches : elles peuvent être plus petites, ou égales, ou plus grandes que les périodes. Comme nous ne considérons que les tâches qui ne sont pas parallélisables, on doit avoir :

$$\forall \tau_k \in \tau, C_k \leq D_k$$

De même, nous ne faisons aucune hypothèse sur l'exécutif temps-réel ciblé, sur les interactions entre les tâches, et sur la politique d'ordonnancement utilisée. Notre contexte d'étude est donc essentiellement défini par deux hypothèses de travail.

Notre première hypothèse signifie que le temps de réponse d'une instance est supérieur ou égal à celui des instances équivalentes des métapériodes précédentes. Dans la suite, nous exprimons cette hypothèse en énonçant simplement que *les temps de réponse ne diminuent pas* :

$$\forall \tau_{k,i} \in \overline{\tau}, TR_s(\tau_{k,i}) \leq TR_s(\tau_{k,i+\delta_k})$$

Notre seconde hypothèse signifie que lorsque le temps de réponse d'une instance est supérieur à celui de l'instance équivalente de la métapériode précédente, alors elle a été retardée par une autre instance dont le temps de réponse a aussi augmenté. Dans la suite, nous exprimons cette hypothèse en énonçant simplement que *les temps de réponse croissent seulement par effet domino*. Nous avons été amenés à étudier plusieurs "versions" de cette hypothèse. La première stipule que lorsque le temps de réponse d'une instance augmente, alors il en existe une seconde dont le temps de réponse a augmenté et qui s'est terminée durant l'exécution de la première. Formellement parlant, cette première version s'exprime ainsi :

$$\begin{array}{lll} \forall \tau_{k,i} \in \bar{\tau}, & TR_s(\tau_{k,i}) < TR_s(\tau_{k,i+\delta_k}) & \\ & \Rightarrow & \\ & \exists \tau_{k',i'} \in \bar{\tau}, & r_{k,i+\delta_k} \leq e_s(\tau_{k',i'}) < e_s(\tau_{k,i+\delta_k}) \wedge \\ & & [TR_s(\tau_{k',i'-\delta_{k'}}) < TR_s(\tau_{k',i'}) \vee r_{k',i'} < r_{k'} + P] \end{array}$$

Cette propriété est généralement vérifiée, mais en multiprocesseur il existe des contre-exemples (exemple 4.1).

Exemple 4.1 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k
τ_1	0	1	5	5
τ_2	0	1	5	5
τ_3	4	4	5	5

La figure 4.1 indique la séquence d'exécution obtenue en biprocesseur lorsque τ_2 est la tâche la moins prioritaire. Le temps de réponse de la deuxième instance de τ_2 augmente par rapport à celui de la première instance. Or, il n'existe aucune instance dont le temps de réponse augmente et qui se

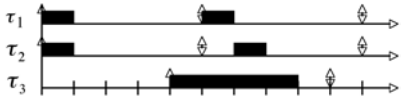


FIG. 4.1 – Contre-exemple de la première version de l'hypothèse de croissance des temps de réponse seulement par effet domino

termine durant la vie de $\tau_{2,1}$ (dans l'intervalle $[5,7[$). Ainsi, la première version de l'hypothèse de croissance de temps de réponse par effet domino n'est pas vérifiée en biprocesseur avec les politiques à priorités fixes.

La construction d'un tel exemple n'est pas possible en monoprocesseur, puisque lorsqu'une instance est retardée par une autre, celle-ci a forcément une priorité plus forte, et elle se termine donc avant. Par contre, nous avons dû reformuler cette hypothèse pour le cas multiprocesseur.

La deuxième version que nous avons développée stipule que lorsque le temps de réponse d'une instance augmente, alors au moment de son activation, il en existe une seconde dont le temps de réponse a augmenté. On obtient alors :

$$\begin{array}{ll} \forall \tau_{k,i} \in \bar{\tau}, & r_{k,i} \geq r + P \wedge TR_s(\tau_{k,i}) > TR_s(\tau_{k,i-\delta_k}) \\ & \Rightarrow \\ & \exists \tau_{k',i'} \in \bar{\tau}, \quad r_{k',i'} < r_{k,i} < e_s(\tau_{k',i'}) \wedge \\ & \qquad\qquad [TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'-\delta_{k'}}) \vee r_{k',i'} < r_{k'} + P] \end{array}$$

Techniquement parlant, la différence entre ces deux versions est minime : dans la première, l'écartement entre les deux instances concernées est exprimé à partir des échéances, alors que dans la seconde, il est exprimé à partir des dates d'activation. L'analyse qui découle de cette hypothèse est insensible à cette différence : elle peut être menée avec l'une ou l'autre. Toutefois, la deuxième version étant plus générale, c'est celle que nous utilisons dans la suite.

Nous traitons maintenant deux exemples de séquences d'exécution pour illustrer nos deux hypothèses : non-décroissance des temps de réponse et croissance des temps de réponse seulement par effet domino (seconde version).

Exemple 4.2 *Reprenons la séquence d'exécution présentée sur la figure 3.1 (exemple 3.1), et examinons les temps de réponse des instances :*

	$\tau_{k,0}$	$\tau_{k,1}$	$\tau_{k,2}$	$\tau_{k,\ldots}$
τ_1	6	7	7	*idem* $\tau_{k,2}$
τ_2	5	5	6	
τ_3	4	5	5	
τ_4	1	1	1	

Pour ce système de tâches, on a $\delta_k = 1$ pour toutes les tâches $\tau_k \in \tau$, il n'y a donc qu'un seul groupe d'instances par tâche. On constate que les temps de réponses sont bien croissants. Cette séquence d'exécution vérifie donc notre première hypothèse.

Examinons maintenant les instances $\tau_{k,i+\delta_k}$ vérifiant $r_{k,i+\delta_k} \geq r+P$ et dont le temps de réponse augmente par rapport à $\tau_{k,i}$, on identifie : $\tau_{1,1}$, $\tau_{2,2}$. Pour chacune d'elles, on obtient :

$\tau_{k,i+\delta_k}$	$\exists \tau_{k',i'} \mid r_{k',i'} < r_{k,i} < e_s(\tau_{k',i'})$	
	$r_{k',i'} < r + P$	$TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'-\delta_{k'}})$
$\tau_{2,2}$		$\tau_{1,1}$
$\tau_{1,1}$	$\tau_{3,1}$	

Ainsi, pour chacune d'elles, il existe une instance permettant de valider l'hypothèse de croissance par effet domino. Cette séquence d'exécution engendrée par EDF vérifie donc les deux hypothèses que nous avons énoncées.

L'exemple suivant montre que nos hypothèses sont même vérifiées par des séquences produites par LLF, et donc propres à DCL.

Exemple 4.3 *Reprenons la séquence d'exécution présentée sur la figure 3.2 (exemple 3.2), et examinons les temps de réponse des instances :*

	$\tau_{k,0}$	$\tau_{k,1}$	$\tau_{k,2}$	$\tau_{k,...}$
τ_1	2	3	3	*idem* $\tau_{k,2}$
τ_2	5	5	6	
τ_3	4	5	5	
τ_4	5	6	6	

Pour ce système de tâches, on a $\delta_k = 1$ pour toutes les tâches $\tau_k \in \tau$, il n'y a donc qu'un seul groupe d'instances par tâche. On constate que les temps de réponse sont bien croissants. Cette séquence d'exécution vérifie donc notre première hypothèse.

Examinons maintenant les instances $\tau_{k,i+\delta_k}$ vérifiant $r_{k,i+\delta_k} \geq r+P$ et dont le temps de réponse augmente par rapport à $\tau_{k,i}$, on identifie : $\tau_{2,2}$, $\tau_{4,1}$. Pour chacune d'elles, on obtient :

$\tau_{k,i+\delta_k}$	$\exists \tau_{k',i'} \mid r_{k',i'} < r_{k,i} < e_s(\tau_{k',i'})$	
	$r_{k',i'} < r+P$	$TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'-\delta_{k'}})$
$\tau_{2,2}$		$\tau_{4,1}$
$\tau_{4,1}$	$\tau_{3,1}$	

Ainsi, pour chacune d'elles, il existe une instance permettant de valider l'hypothèse de croissance par effet domino. Cette séquence d'exécution engendrée par LLF vérifie donc les deux hypothèses que nous avons énoncées.

On peut justifier ces hypothèses par les arguments suivants. La charge injectée lors de chaque métapériode est croissante : pour chaque métapériode postérieure à r, la charge injectée est $U.P$ (lemme 2.1), pour les métapériodes antérieures, elle est strictement inférieure à $U.P$. Or, si la charge injectée est croissante, l'aptitude du système à servir chaque tâche diminue, et donc, les temps de réponse augmentent ou au mieux stagnent. Lorsque le temps de réponse d'une instance est supérieur à celui de l'instance équivalente de la métapériode précédente, cela signifie qu'elle a eu moins rapidement accès à un processeur, et donc qu'au moment de son activation, le système était plus chargé que lors de la métapériode précédente.

4.3 Intervalle de faisabilité

En nous appuyant sur les deux hypothèses énoncées dans la section 4.2, nous proposons ici une nouvelle approche de l'ordonnancement multiprocesseur des systèmes de tâches périodiques. Tout d'abord, nous étudions certaines conditions permettant d'affirmer que le temps de réponse des instances est borné. Ensuite, nous en déduisons que les temps de réponse se stabilisent, et nous donnons une borne de la durée avant stabilisation. Finalement, nous établissons un intervalle de faisabilité.

4.3.1 Majoration des temps de réponse

Dans cette section, nous étudions les séquences vérifiant les deux propriétés suivantes :
- non-décroissance des temps de réponse
- conservativité

Dans toute la suite, nous désignons par S^+ l'ensemble des séquences d'exécution de τ vérifiant ces deux propriétés. Nous montrons pour les séquences de S^+ que si la charge du système est inférieure ou égale au nombre de processeurs, alors les temps de réponse sont bornés. Cette propriété est importante puisque conjuguée avec la non-décroissance des temps de réponse, elle permet d'assurer leur stabilisation (résultat établi en section 4.3.2). Cette propriété a aussi un autre intérêt puisqu'elle établit l'existence d'une borne maximale des dépassements d'échéance dès que le système n'est pas surchargé.

Pour obtenir ce résultat, nous raisonnons par contraposée : si les temps de réponse ne sont pas bornés, alors la charge du système est strictement supérieure au nombre de processeurs. Nous commençons par établir certaines propriétés induites par le fait que les temps de réponse ne soient pas bornés.

Lemme 4.1 *Soit $s \in S^+$ telle que le temps de réponse des instances n'est pas borné. Le nombre d'instances actives simultanément n'est pas borné.*

Démonstration : Soient $s \in S^+$ et $M \in \mathbb{N}$.

Par hypothèse, le temps de réponse n'est pas borné, ainsi pour tout $m \in \mathbb{N}$, il existe une instance $\tau_{k,i}$ telle que $TR_s(\tau_{k,i}) \geq m$.

Il existe donc une instance $\tau_{k,i}$ telle que $TR_s(\tau_{k,i}) \geq M.P$.

Ainsi, pendant l'exécution $\tau_{k,i}$, il y a eu au moins M instances générées :

$$\tau_{k,i+\delta_k}, \tau_{k,i+2.\delta_k}, \ldots, \tau_{k,i+M.\delta_k}$$

A la terminaison de $\tau_{k,i}$, ces M instances sont toujours actives puisque leur temps de réponse ne peut pas être inférieur à celui de $\tau_{k,i}$.

CQFD.

Lemme 4.2 *Soit $s \in S^+$ telle que le temps de réponse des instances ne soit pas borné. La charge restante n'est pas bornée.*

Démonstration : Soient $s \in S^+$ et $M \in \mathbb{N}$.

Par hypothèse, le temps de réponse des instances exécutées selon l'ordre établi par s n'est pas borné, ainsi d'après le lemme 4.1, le nombre d'instances actives simultanément n'est pas borné.

Il existe donc $t \in \mathbb{N}$ tel que le nombre d'instances actives soit au moins égal à M. Ces M instances sont actives, donc leur charge restante ne peut pas être nulle, elle est donc au minimum de 1. Ainsi, la charge pendante dans le système à l'instant t est au moins égale à M.

CQFD.

Lemme 4.3 *Soit $s \in S^+$ telle que les temps de réponse des instances ne soit pas borné. Le système arrive à saturation et reste indéfiniment saturé.*

Démonstration : Soit $s \in S^+$.

Par hypothèse, le temps de réponse des instances exécutées selon l'ordre établi par s n'est pas borné, ainsi d'après le lemme 4.1, le nombre d'instances actives simultanément n'est pas borné.

Ainsi, il existe $t \in \mathbb{N}$ tel que le nombre d'instances actives soit supérieur ou égal à $p.P$.

Par hypothèse, la séquence s est conservative, on a donc $s_t = min\{p, |s_t^\star|\} = p$. Ainsi, le système est saturé à l'instant t.

D'autre part, le nombre d'instances actives à l'instant $t+1$ est au moins $p.(P-1)$. Le système est donc encore saturé à l'instant $t+1$, et il en est de même au moins jusqu'à l'instant $t+P-1$. Le système est donc saturé sur la métapériode commençant à l'instant t.

Considérons maintenant l'ensemble E des instances actives à l'instant t. Pour chaque instance $\tau_{k,i} \in E$, on a $r_{k,i} \leq t$ et $e_s(\tau_{k,i}) > t$. On obtient alors :

$$\forall \tau_{k,i} \in E, r_{k,i+\delta_k} \leq t + P$$

De plus par hypothèse, le temps de réponse des instances ne peut pas diminuer, on obtient aussi :

$$\forall \tau_{k,i} \in E, e_s(\tau_{k,i+\delta_k}) > t + P$$

Ainsi, le nombre d'instances actives en $t+P$ est au moins égal à celui en t, le système est donc aussi saturé de $t+P$ à $t+2P-1$. Il en est aussi de même pour tous les instants $t + \mathbb{N}.P$, et donc le système est indéfiniment saturé à partir de t.

CQFD.

Nous pouvons maintenant démontrer le résultat annoncé.

Théorème 4.1 *Soit $s \in S^+$. Si $U \leq p$, alors le temps de réponse des instances, exécutées selon l'ordre établi par s, est borné.*

Démonstration : Soit $s \in S^+$.

Raisonnons sur la contraposée. Posons pour $0 \leq k < P$:

$$E_k = \{t | t \bmod P = k\} \text{ et } F_k = \{\dot{U}_s(t) | t \bmod P = k\}$$

Par hypothèse, le temps de réponse des instances, exécutées selon l'ordre établi par s, n'est pas borné, ainsi d'après le lemme 4.2, la charge restante n'est pas bornée. D'autre part, il existe un nombre fini d'ensembles F_k $(0 \leq k < P)$, il existe alors au moins un de ces ensembles qui n'est pas borné, notons-le F_K.

Soit $t \in E_K$ et posons $E = \{t' | t' \leq t \text{ et } t' \in E_K\}$.

L'ensemble E est fini, ainsi l'ensemble des valeurs $\dot{U}_s(t')$ pour $t' \in E$ est borné. Or F_K n'est pas borné, il existe donc $t' \in E_K \setminus E$ tel que $\dot{U}_s(t') > \dot{U}_s(t)$.

Ainsi quel que soit $t \in E_K$, il existe $t' \in E_K$ tel que $t' > t$ et $\dot{U}_s(t') > \dot{U}_s(t)$.

Par hypothèse, le temps de réponse des instances, exécutées selon l'ordre établi par s, n'est pas borné, ainsi d'après le lemme 4.3, il existe $t \in \mathbb{N}$ tel que le système est toujours saturé à partir de t. Il existe donc aussi $t \geq r$ et appartenant à E_K tel que le système est toujours saturé à partir de t.

Or, d'après ce qui précède, il existe $t' \in E_K$ tel que $t' > t$ et $\dot{U}_s(t') > \dot{U}_s(t)$.

Notons K le nombre de métapériodes séparant t et t' : $K = \frac{t'-t}{P}$.

D'après le lemme 2.1, la charge injectée entre t et t' est égale à $K.U.P$, puisque $t \geq r$. Or, le système est saturé à partir de t et la séquence s est conservative, donc la charge absorbée entre t et t' est égale à $K.P.p$.

On obtient donc successivement :

$$\dot{U}_s(t') = \dot{U}_s(t) + K.U.P - K.P.p$$

$$K.U.P - K.P.p = \dot{U}_s(t') - \dot{U}_s(t) > 0$$

$$U > p$$

CQFD.

Pour obtenir ce résultat, nous utilisons le fait que la séquence s est conservative dans le lemme 4.3 seulement pour assurer que s'il y a au moins p instances actives à un instant donné, alors il y a exactement p instances exécutées à cet instant-là. Lorsque les instances actives sont attribuées aux processeurs dynamiquement, cette propriété est vérifiée : dès qu'une instance se termine, toutes les instances en attente peuvent être choisies : elles ne sont pas pré-attribuées aux processeurs. Ainsi, dans le cadre de l'approche globale, cette propriété est vérifiée dès que le contexte est non-préemptif, ou dès que la migration des instances est globale. En contexte préemptif avec migration partielle, dès qu'une instance a débuté son exécution sur un processeur, elle ne peut plus en changer, d'autre part, une instance peut-être interrompue au profit d'une autre. Ainsi dans ce contexte, il peut y avoir plusieurs instances actives mais toutes attribuées au même processeur, l'hypothèse de conservativité n'est alors plus vérifiée.

Ainsi, le domaine d'application du théorème 4.1 dépend des interactions entre les tâches et de l'exécutif temps-réel ciblé. Lorsque les tâches ne peuvent pas être bloquées (par exemple, si elles sont indépendantes), et lorsque l'exécutif est non-préemptif ou que la migration est totale, alors le théorème 4.1 est directement applicable. Pour les autres contextes, on doit d'abord étudier leur impact sur l'hypothèse de conservativité avant d'appliquer le théorème 4.1. D'autre part, on peut facilement adapter le théorème 4.1 à l'ordonnancement selon l'approche partitionnée : pour les séquences de S^+, les temps de réponse sont bornés dès que la charge attribuée à chaque processeur est inférieure ou égale à 1.

4.3.2 Stabilisation des temps de réponse

Tout d'abord, énonçons formellement ce que nous entendons par la stabilisation des temps de réponse.

Définition 4.1 *Soit s une séquence d'exécution. Les temps de réponse des instances, exécutées selon l'ordre établi par s, sont stables à partir de $t \in \mathbb{N}$ si et seulement si :*

$$\forall \tau_{k,i} \in \bar{\tau}, r_{k,i} \geq t \Rightarrow TR_s(\tau_{k,i}) = TR_s(\tau_{k,i+\delta_k})$$

Cette propriété permet d'assurer une certaine régularité dans une séquence d'exécution : après un certain temps, les temps de réponse n'évoluent plus. Cette régularité est moins rigide que celle imposée par la cyclicité : peu importe qu'il y ait de petites variations locales dans une séquence d'exécution tant que le comportement global (les temps de réponse) n'est pas affecté.

Cette régularité permet d'assurer l'existence d'un intervalle de faisabilité fini : il suffit d'étudier la séquence jusqu'à ce que les temps de réponse n'évoluent plus pour valider ou invalider une séquence d'exécution. Pour montrer que les temps de réponse se stabilisent, nous nous appuyons sur le fait qu'ils sont bornés et non-décroissants. Ensuite, nous présentons les deux contextes pour lesquels nous assurons que les temps de réponse se stabilisent.

Théorème 4.2 *Soit s une séquence d'exécution telle que les temps de réponses soient bornés et non-décroissants. Il existe un instant à partir duquel les temps de réponse sont stables.*

Démonstration :
Soit M un majorant des temps de réponse des instances, puisqu'une même instance n'est pas parallélisable, on obtient :

$$\forall \tau_{k,i} \in \bar{\tau}, C_k \leq TR_s(\tau_{k,i}) \leq M$$

Or, par hypothèse, les temps de réponse ne peuvent pas diminuer, on a donc aussi :

$$\forall \tau_k \in \tau, \forall i < \delta_k, C_k \leq TR_s(\tau_{k,i}) \leq TR_s(\tau_{k,i+\delta_k}) \leq TR_s(\tau_{k,i+2\delta_k}) \leq \ldots \leq M$$

Les indices j vérifiant $TR_s(\tau_{k,i+j\delta_k}) < TR_s(\tau_{k,i+(j+1)\delta_k})$ sont en nombre fini. Il existe donc $\Delta \in \mathbb{N}$ tel que :

$$\forall \tau_k \in \tau, \forall i < \delta_k, \forall j \geq \Delta, TR_s(\tau_{k,i+j\delta_k}) = TR_s(\tau_{k,i+(j+1)\delta_k})$$

CQFD.

En utilisant la condition qui assure que les temps de réponse sont bornés (théorème 4.1), on obtient le corollaire suivant.

Corollaire 4.1 *Soit s une séquence d'exécution conservative telle que les temps de réponse ne diminuent pas. Si $U \leq p$, alors il existe un instant à partir duquel les temps de réponse sont stables.*

Une séquence d'exécution est valide seulement si toutes les instances respectent leur échéance. Ainsi, toutes les instances $\tau_{k,i}$ doivent vérifier : $TR(\tau_{k,i}) \leq D_k$. On peut donc aussi borner le temps de réponse par cette hypothèse. On obtient alors le corollaire suivant.

Corollaire 4.2 *Soit s une séquence d'exécution valide telle que les temps de réponse ne diminuent pas. Il existe un instant à partir duquel les temps de réponse sont stables.*

Le premier corollaire revêt une importance particulière puisqu'il permet d'assurer la stabilisation des temps de réponse sans utiliser le fait que la séquence soit valide. On peut alors étudier les temps de réponse des instances pour une séquence donnée (valide ou non), et déterminer ensuite les échéances minimales que l'on peut imposer aux tâches, ou plus généralement, énoncer un diagnostic fini portant sur l'intégralité de la séquence d'exécution.

4.3.3 Durée de stabilisation des temps de réponse

Dans cette section, nous étudions les séquences d'exécution vérifiant les trois propriétés suivantes :
- les temps de réponse ne peuvent pas diminuer,
- les temps de réponse ne peuvent croître que par effet domino,
- les temps de réponse sont bornés.

Dans toute la suite, nous désignons par $S^\star$ l'ensemble des séquences d'exécution vérifiant ces trois propriétés. Nous donnons maintenant une borne maximale de la durée de stabilisation des temps de réponse. Notre démarche consiste à déterminer le *rythme de croissance* minimal des temps de réponse. Effectivement, l'effet domino permet de majorer la longueur des intervalles temporels ne contenant aucune activation d'instance dont le temps de réponse augmente. On peut alors donner une borne maximale de la durée de stabilisation puisque les temps de réponse ne diminuent pas et évoluent dans un intervalle borné.

Le théorème suivant donne une condition de détection de la stabilisation des temps de réponse et permet de minorer le rythme de croissance des temps de réponse.

Théorème 4.3 *Soient $s \in S^\star$, M un majorant des temps de réponse et $t \geq r$. Les temps de réponse sont stables à partir de t dès que la condition suivante est vérifiée :*

$$\forall \tau_{k,i} \in \overline{\tau}, t \leq r_{k,i} < t + M \Rightarrow TR_s(\tau_{k,i}) = TR_s(\tau_{k,i+\delta_k})$$

Démonstration :
Raisonnons par l'absurde. Supposons qu'il existe une instance $\tau_{k,i}$ telle que $r_{k,i} \geq t$ et vérifiant :

$$TR_s(\tau_{k,i}) < TR_s(\tau_{k,i+\delta_k})$$

Par hypothèse, on a $t \geq r$, on en déduit $r_{k,i+\delta_k} \geq r + P$, donc d'après l'hypothèse de croissance des temps de réponse seulement par effet domino, il existe une instance $\tau_{k',i'+\delta_{k'}}$ vérifiant :
- $r_{k',i'+\delta_{k'}} < r_{k,i+\delta_k} < e_s(\tau_{k',i'+\delta_{k'}})$
- $TR_s(\tau_{k',i'}) < TR_s(\tau_{k',i'+\delta_{k'}})$ ou $r_{k',i'+\delta_{k'}} < r + P$

Supposons $r_{k',i'+\delta_{k'}} \geq r + P$. L'instance $\tau_{k',i'+\delta'_k}$ vérifie alors les hypothèses satisfaites par $\tau_{k,i+\delta_k}$. On peut donc appliquer à $\tau_{k',i'+\delta'_k}$ le raisonnement que l'on vient de mener avec $\tau_{k,i+\delta_k}$. On en déduit qu'il existe une suite $(u_j)_{0 \leq j \leq l}$ d'instances τ_{α_j,β_j} vérifiant :

1. $u_0 = \tau_{k,i+\delta_k}$ et $r_{\alpha_l,\beta_l} < r + P$
2. $\forall j \in \{1, \ldots, l\}, r_{\alpha_j,\beta_j} < r_{\alpha_{j-1},\beta_{j-1}} < e_s(\tau_{\alpha_j,\beta_j})$
3. $\forall j \in \{0, \ldots, l-1\}, TR_s(\tau_{\alpha_j,\beta_j}) > TR_s(\tau_{\alpha_j,\beta_j-\delta_{\alpha_j}})$

Par hypothèse, on a $r_{k,i} \geq t$, et donc $r_{\alpha_0,\beta_0} \geq t + P$. En utilisant (1), on obtient aussi $r_{\alpha_l,\beta_l} < r + P \leq t + P$. Il existe donc une instance τ_{α_j,β_j} vérifiant :

$$r_{\alpha_{j-1},\beta_{j-1}} \geq t + P \text{ et } r_{\alpha_j,\beta_j} < t + P$$

Par (2), on obtient :

$$r_{\alpha_{j-1},\beta_{j-1}} - r_{\alpha_j,\beta_j} < e_s(\tau_{\alpha_j,\beta_j}) - r_{\alpha_j,\beta_j}$$

Puisque M est un majorant des temps de réponse des instances, on obtient :

$$r_{\alpha_{j-1},\beta_{j-1}} - r_{\alpha_j,\beta_j} < M$$

Or par hypothèse, on a $r_{\alpha_j,\beta_j} < t + P$, on obtient donc :

$$r_{\alpha_{j-1},\beta_{j-1}} < t + P + M$$

Et finalement, on a :

$$t + P \leq r_{\alpha_{j-1},\beta_{j-1}} < t + P + M$$

La propriété (3) étant aussi vérifiée, l'instance $\tau_{\alpha_{j-1},\beta_{j-1}-\delta_{\alpha_{j-1}}}$ contredit la propriété satisfaite par l'instant t.

CQFD.

On peut maintenant déterminer une borne maximale de la durée de stabilisation des temps de réponse.

Théorème 4.4 *Soient $s \in S^\star$ et M un majorant des temps de réponse. Les temps de réponse des instances dans s sont stables au plus tard à partir de :*

$$r + P + (M-1).P.\sum_{\tau_k \in \tau} \frac{M - C_k}{T_k}$$

Si la séquence d'exécution est valide, on a aussi :

$$r + P + (D-1).P.\sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}$$

Démonstration :

Soit $M_k \in \mathbb{N}$ un majorant des temps de réponse des instances de τ_k, et M un majorant des M_k.

Classons les instances d'une tâche $\tau_k \in \tau$ en δ_k groupes de la manière suivante :

$$G_{k,i} = \{\tau_{k,j} | j \bmod \delta_k = i\}, \text{ pour } 0 \leq i < \delta_k$$

Par hypothèse, les temps de réponse des instances appartenant à un même $G_{k,i}$ ne peuvent pas diminuer :

$$\tau_{k,i_1}, \tau_{k,i_2} \in G_{k,i} \Rightarrow [i_1 \leq i_2 \Leftrightarrow TR_s(\tau_{k,i_1}) \leq TR_s(\tau_{k,i_2})]$$

D'autre part, les temps de réponse des instances appartenant à $G_{k,i}$ varient entre C_k et M_k. Ainsi, chaque groupe permet au plus $M_k - C_k$ évolutions du temps de réponse, puisque celui-ci ne peut pas diminuer à l'intérieur d'un même groupe.

Ainsi, une tâche τ_k permet au pire $(M_k - C_k).\delta_k$ évolutions possibles du temps de réponse. Le nombre maximum Δ d'instances dont le temps de réponse augmente pour le système de tâches complet est donc :

$$\Delta = \sum_{\tau_k \in \tau} (M_k - C_k).\delta_k = P.\sum_{\tau_k \in \tau} \frac{M_k - C_k}{T_k}$$

S'il n'existe pas d'instance dont le temps de réponse augmente, alors les temps de réponse sont stables dès $t = 0$ et le théorème est alors vérifié.

Supposons maintenant qu'il en existe une et notons-la $\tau_{k,i}$. On a donc $TR_s(\tau_{k,i}) > TR_s(\tau_{k,i-\delta_k})$.

De la même manière que dans la preuve du théorème 4.3, on montre que d'après l'hypothèse de croissance des temps de réponse seulement par effet domino, il existe une suite $(u_j)_{0 \leq j \leq l}$ d'instances τ_{α_j,β_j} vérifiant :

1. $u_0 = \tau_{k,i}$ et $r_{\alpha_l,\beta_l} < r + P$
2. $\forall j \in \{1,\ldots,l\}, r_{\alpha_j,\beta_j} < r_{\alpha_{j-1},\beta_{j-1}} < e_s(\tau_{\alpha_j,\beta_j})$
3. $\forall j \in \{0,\ldots,l-1\}, TR_s(\tau_{\alpha_j,\beta_j}) > TR_s(\tau_{\alpha_j,\beta_j-\delta_{\alpha_j}})$

On en déduit :

$$r_{\alpha_0,\beta_0} > r_{\alpha_1,\beta_1} > \cdots > r_{\alpha_l,\beta_l}$$

D'autre part, les temps de réponse étant bornés par M, on obtient en utilisant (2) :

$$\forall j \in \{1,\ldots,l\}, r_{\alpha_{j-1},\beta_{j-1}} - r_{\alpha_j,\beta_j} < M$$

De proche en proche, on a finalement :

$$r_{\alpha_0,\beta_0} - r_{\alpha_l,\beta_l} \leq l.(M-1)$$

En utilisant le fait que $r_{\alpha_l,\beta_l} < r + P$, on en déduit :

$$r_{\alpha_0,\beta_0} \leq r + P - 1 + l.(M-1)$$

D'autre part, d'après (3), les temps de réponse des instances de la suite (u_j) augmentent tous par rapport à la métapériode précédente, excepté pour $j = l$. Or d'après ce qui précède, il existe au plus Δ instances dont le temps de réponse augmente. On a donc $l \leq \Delta$. On obtient finalement :

$$r_{k,i} \leq r + P - 1 + \Delta.(M-1)$$

Ainsi, dès que le temps de réponse d'une instance augmente, on peut borner sa date d'activation.

D'après le théorème 4.2, les temps de réponse se stabilisent. On peut alors supposer que $\tau_{k,i}$ est la dernière instance dont le temps de réponse augmente. Ainsi, le temps de réponse de toutes les instances activées après $r_{k,i}$ est stable, on a :

$$\forall \tau_{k',i'} \in \overline{\tau}, r_{k',i'} > r_{k,i} \Rightarrow TR_s(\tau_{k',i'}) = TR_s(\tau_{k',i'+\delta_{k'}})$$

En combinant cette propriété avec la borne obtenue ci-dessus, on obtient que les temps de réponse sont stables au pire à partir de :

$$r + P + (M-1).P.\sum_{\tau_k \in \tau} \frac{M_k - C_k}{T_k}$$

CQFD.

Dans les exemples 3.1 et 3.2, nous avons donné des séquences d'exécution, produites respectivement par EDF et LLF, qui vérifient nos deux hypothèses. La borne que nous venons de déterminer s'applique donc à ces deux exemples. Nous pouvons alors la comparer avec les valeurs exactes. On obtient les valeurs suivantes :

	Durée exacte de stabilisation	Borne de la durée de stabilisation
Exemple avec EDF	17	71
Exemple avec LLF	17	61

Les durées exactes de stabilisation des temps de réponse sont bien inférieures à notre borne. La différence entre la durée exacte et notre borne est principalement due au fait que le temps de réponse des instances appartenant à un même groupe $(\tau_{k,i})_{i \in \delta_k.\mathbb{N}}$ ne varie pas toujours de C_k à D_k. Ainsi, le nombre d'évolutions du temps de réponse pour un tel groupe est strictement inférieur à $D_k - C_k$. Par exemple, dans la séquence d'exécution indiquée sur la figure 3.1, le temps de réponse des instances du groupe $(\tau_{2,i})_{i \in \mathbb{N}}$ varie de 5 à 6, alors que potentiellement, il peut varier de 4 à 7.

Les théorèmes 4.3 et 4.4 sont complémentaires puisque le premier permet de détecter à la volée si la stabilisation des temps de réponse est atteinte, alors que le deuxième borne la durée de stabilisation. Ces deux résultats sont directement applicables aux séquences d'exécution valides puisque le temps de réponse des instances est alors borné par l'échéance des tâches. Pour les séquences d'exécution non-valides, le théorème 4.1 montre l'existence d'une borne des temps de réponse pour les séquences d'exécution conservatives et les systèmes de tâches non-surchargés. Les théorèmes 4.3 et 4.4 sont donc aussi applicables aux séquences d'exécution non-valides. Ce type de résultat est nouveau à notre connaissance puisqu'il établit l'existence d'un intervalle d'étude fini pour les séquences d'exécution non-valides. On peut alors établir un diagnostic tenant compte de toute la séquence d'exécution.

4.3.4 Intervalle de faisabilité

Le théorème 4.4 fournit une borne de la durée de stabilisation du temps de réponse des instances pour les séquences d'exécution valides. Nous en déduisons maintenant un intervalle de faisabilité

applicable à toutes les séquences d'exécution satisfaisant nos deux hypothèses de travail : non-décroissance du temps de réponse et croissance du temps de réponse seulement par effet domino. Notre résultat est relativement général puisque nos hypothèses ne dépendent ni de l'exécutif temps-réel ciblé, ni des interactions entre les tâches, ni d'un mécanisme particulier d'attribution des priorités aux instances.

Théorème 4.5 *L'intervalle suivant est un intervalle de faisabilité des séquences pour lesquelles le temps de réponse des instances ne diminue pas et ne croît que par effet domino :*

$$\left[0, r + P + 2D + (D-1).P.\sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}\right[$$

Dans le cas où les échéances D_k des tâches $\tau_k \in \tau$ vérifient toutes $D_k \leq T_k$, on obtient aussi :

$$[0, r + P + 2D + (D-1).P.(n-U)[$$

Démonstration :
Soit $s \in S^\star$ une séquence d'exécution dans laquelle au moins une échéance est manquée.
Notons $\tau_{k,i}$ l'instance correspondant à la plus petite échéance manquée :

$$r_{k,i} = \min\{r_{k',i'} | e_s(\tau_{k',i'}) > d_{k',i'}\}$$

Si $r_{k,i} < r + P$, on a alors $d_{k,i} < r + P + D$, et le théorème est vérifié.
Supposons maintenant que $r_{k,i} \geq r + P$, ainsi l'instance $\tau_{k,i-\delta_k}$ existe. Etant donné que $\tau_{k,i}$ est la première instance à manquer son échéance, son temps de réponse augmente donc par rapport à celui de $\tau_{k,i-\delta_k}$.

De la même manière que dans la preuve du théorème 4.3, on montre que d'après l'hypothèse de croissance des temps de réponse seulement par effet domino, il existe une suite $(u_j)_{0 \leq j \leq l}$ d'instances τ_{α_j,β_j} vérifiant :

1. $u_0 = \tau_{k,i}$ et $r_{\alpha_l,\beta_l} < r + P$
2. $\forall j \in \{1, \ldots, l\}, r_{\alpha_j,\beta_j} < r_{\alpha_{j-1},\beta_{j-1}} < e_s(\tau_{\alpha_j,\beta_j})$
3. $\forall j \in \{0, \ldots, l-1\}, TR_s(\tau_{\alpha_j,\beta_j}) > TR_s(\tau_{\alpha_j,\beta_j-\delta_{\alpha_j}})$

Réutilisons un raisonnement que nous avons mené dans la preuve du théorème 4.4. Lorsqu'aucune instance ne manque son échéance, le temps de réponse des instances est borné, pour une instance $\tau_{k,i}$ quelconque, il évolue entre C_k et D_k. Par hypothèse, le temps de réponse ne diminue pas, le nombre maximum Δ d'instances augmentant leur temps de réponse sans qu'aucune d'elles ne manque son échéance est alors :

$$\Delta = P.\sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}$$

Excepté pour $j = l$, le temps de réponse des instances de la suite (u_j) augmente. Ainsi, la suite (u_j) contient l instances dont le temps de réponse augmente. Or, sans qu'aucune échéance ne soit manquée, il ne peut y avoir que Δ instances dont le temps de réponse augmente. D'autre part, $\tau_{k,i}$ est la première instance à manquer son échéance. Ainsi, aucune instance $(u_j)_{0<j<l}$ ne manque son échéance, or leurs temps de réponse augmentent tous, on a donc $l \leq \Delta + 1$.

Comme dans la preuve du théorème 4.4, on borne $r_{k,i} - r_{\alpha_l,\beta_l}$ en utilisant la longueur l de la suite. On obtient alors :

$$r_{k,i} \leq r + P + D - 1 + (D-1).P.\sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}$$

Et on a finalement :

$$d_{k,i} < r + P + 2D + (D-1).P.\sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}$$

CQFD.

Dans l'exemple 3.3, nous avons exhibé une séquence d'exécution produite par EDF dans laquelle la première échéance manquée est à $t = 26$ et donc après $r + 2P = 19$. Nous n'avons pas montré explicitement que cette séquence vérifie nos deux hypothèses, cependant les résultats de la section 4.4, nous permettent de l'affirmer. L'intervalle de faisabilité de ce système de tâches est alors $[0, 79[$. Or, la première échéance manquée est à $t = 26$, il en existe donc une qui appartient à l'intervalle de faisabilité.

4.4 Application aux séquences $\overline{\text{PFI}}$

Dans cette section, nous appliquons l'étude menée dans la section 4.3 aux séquences d'exécution produites par des politiques de $\overline{\text{PFI}}$ en contexte multiprocesseur et préemptif avec migration totale. Pour cela, nous montrons que ces séquences d'exécution satisfont nos deux hypothèses de travail : non-décroissance des temps de réponse, et croissance des temps de réponse seulement par effet domino. Nous montrons aussi pour cette catégorie de séquences que la stabilisation des temps de réponse est équivalente à la cyclicité. Dans la suite de cette section, nous appelons par abus de langage séquences $\overline{\text{PFI}}$, toutes les séquences produites par des politiques de $\overline{\text{PFI}}$ en contexte préemptif avec migration totale.

4.4.1 Satisfaction des hypothèses de travail

Pour appliquer les résultats de la section 4.3 aux séquences $\overline{\text{PFI}}$, nous devons montrer qu'elles vérifient les hypothèses de travail. On peut résumer l'idée principale ainsi :

- dans $\overline{\text{PFI}}$, une instance n'obtient l'accès à un processeur que dans deux cas : soit elle vient de s'activer et elle fait partie des p instances les plus prioritaires, soit une instance plus prioritaire vient de se terminer et elle fait alors partie des p instances les plus prioritaires,
- dans $\overline{\text{PFI}}$, une instance ne perd l'accès au processeur que dans deux cas : soit elle vient de se terminer, soit une instance plus prioritaire vient de s'activer et elle ne fait plus partie des p instances les plus prioritaires.

Montrons maintenant que les temps de réponse ne peuvent pas diminuer dans les séquences produites par des politiques d'ordonnancement de $\overline{\text{PFI}}$.

Théorème 4.6 *Soit s une séquence $\overline{PFI}$, alors les temps de réponse ne peuvent pas diminuer.*

Démonstration :

Raisonnons par l'absurde.

Soit $\tau_{k,i} \in \overline{\tau}$ telle que $TR_s(\tau_{k,i}) > TR_s(\tau_{k,i+\delta_k})$.

L'instance $\tau_{k,i+\delta_k}$ a accédé C_k fois à un processeur pendant les $TR_s(\tau_{k,i+\delta_k})$ premiers instants de son exécution. Or, le temps de réponse de $\tau_{k,i}$ est strictement supérieur à celui de $\tau_{k,i+\delta_k}$, ainsi $\tau_{k,i}$ a accédé moins de C_k fois à un processeur pendant les $TR_s(\tau_{k,i+\delta_k})$ premiers instants de son exécution.

Il existe donc un instant $t \in \mathbb{N}$ vérifiant les deux propriétés suivantes :

- à l'instant t, l'instance $\tau_{k,i}$ n'a pas accès à un processeur, donc il existe au moins p instances plus prioritaires qu'elle ; notons E_1 l'ensemble de celles qui sont effectivement exécutées à l'instant t :

$$E_1 = \{\tau_{k',i'} \in \overline{\tau} | prio(\tau_{k',i'}) > prio(\tau_{k,i}) \wedge \tau_{k',i'} \in s_t\}$$

- à l'instant $t+P$, l'instance $\tau_{k,i+\delta_k}$ a accès à un processeur, donc il existe moins de p instances plus prioritaires qu'elle ; notons E_2 leur ensemble :

$$E_2 = \{\tau_{k',i'} \in \overline{\tau} | prio(\tau_{k',i'}) > prio(\tau_{k,i+\delta_k}) \wedge \tau_{k',i'} \in s_{t+P}\}$$

Par définition, on a $|E_1| \geq p > |E_2|$. Il existe donc une instance $\tau_{k',i'} \in E_1$ telle que $\tau_{k',i'+\delta_{k'}} \notin E_2$.

Puisque $\tau_{k',i'} \in E_1$, on a :

$$prio(\tau_{k',i'}) > prio(\tau_{k,i})$$

Par hypothèse ($\overline{\text{PFI}}$), on obtient :

$$prio(\tau_{k',i'+\delta'_k}) > prio(\tau_{k,i+\delta_k})$$

L'instance $\tau_{k',i'+\delta'_k}$ est donc plus prioritaire que l'instance $\tau_{k,i+\delta_k}$.

D'autre part, l'instance $\tau_{k,i+\delta_k}$ s'exécute à l'instant $t+P$, bien que l'instance $\tau_{k',i'+\delta'_k}$ ne s'exécute pas à l'instant $t+P$, puisqu'elle n'appartient pas à E_2. Etant plus prioritaire, l'instance $\tau_{k',i'+\delta'_k}$ ne peut être que terminée à l'instant $t+P$:

$$e_s(\tau_{k',i'+\delta'_k}) \leq t+P$$

De plus, l'instance $\tau_{k',i'}$ est exécutée à l'instant t, son échéance est donc plus tardive :

$$e_s(\tau_{k',i'}) > t$$

On en déduit puisque $r_{k',i'+\delta_{k'}} - r_{k',i'} = P$:

$$TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'+\delta'_k})$$

De plus, l'instance $\tau_{k,i+\delta_k}$ est exécutée à l'instant $t+P$, son échéance est donc plus tardive :

$$e_s(\tau_{k,i+\delta_k}) > t+P$$

On en déduit :

$$e_s(\tau_{k',i'+\delta'_k}) < e_s(\tau_{k,i+\delta_k})$$

Il existe donc une instance $\tau_{k',i'} \in \overline{\tau}$ vérifiant les propriétés suivantes :
- $TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'+\delta'_k})$
- $e_s(\tau_{k',i'+\delta'_k}) < e_s(\tau_{k,i}+\delta_k)$

Le raisonnement que l'on vient de mener repose sur le fait que $TR_s(\tau_{k,i}) > TR_s(\tau_{k,i+\delta_k})$. L'instance $\tau_{k',i'}$ vérifiant aussi cette propriété, on peut le mener à nouveau avec $\tau_{k',i'}$ à la place de $\tau_{k,i}$. En continuant ainsi de proche en proche, on construit une suite infinie d'instances $(u_j)_{j\in\mathbb{N}}$ vérifiant :
- $u_0 = \tau_{k,i}$, $u_1 = \tau_{k',i'}$, etc.
- $\forall j \in \mathbb{N}, e_s(u_j) > e_s(u_{j+1})$

L'existence d'une telle suite amène à considérer des instances dont l'échéance est négative et donc qui se finissent avant d'avoir commencé. Ceci est absurde et on obtient donc une contradiction. Ainsi les temps de réponse ne peuvent pas diminuer.

CQFD.

Montrons maintenant pour les séquences $\overline{\text{PFI}}$ que les temps de réponse peuvent seulement croître par effet domino.

Théorème 4.7 *Soit s une séquence $\overline{PFI}$, alors les temps de réponse peuvent seulement croître par effet domino.*

Démonstration :

Soit $\tau_{k,i} \in \overline{\tau}$ telle que $r_{k,i} \geq r+P$ et $TR_s(\tau_{k,i}) > TR_s(\tau_{k,i-\delta_k})$.

Puisque le temps de réponse de $\tau_{k,i}$ augmente par rapport à celui de $\tau_{k,i-\delta_k}$, il existe un instant t vérifiant les deux propriétés suivantes :
- à l'instant t, l'instance $\tau_{k,i}$ n'a pas accès à un processeur, donc il existe au moins p instances plus prioritaires qu'elle ; notons E_1 l'ensemble de celles qui sont effectivement exécutées :

$$E_1 = \{\tau_{k',i'} \in \overline{\tau} | prio(\tau_{k',i'}) > prio(\tau_{k,i}) \wedge \tau_{k',i'} \in s_t\}$$

– à l'instant $t-P$, l'instance $\tau_{k,i-\delta_k}$ a accès à un processeur, donc il existe moins de p instances plus prioritaires qu'elle ; notons E_2 leur ensemble :

$$E_2 = \{\tau_{k',i'} \in \overline{\tau} | prio(\tau_{k',i'}) > prio(\tau_{k,i+\delta_k}) \wedge \tau_{k',i'} \in s_{t-P}\}$$

Par définition, on obtient $|E_1| \geq p > |E_2|$. Il existe alors une instance $\tau_{k',i'} \in E_1$ telle que $\tau_{k',i'-\delta_{k'}} \notin E_2$. Comme l'instance $\tau_{k',i'}$ est exécutée à l'instant t et que $\tau_{k,i}$ est active à l'instant t, on a :

$$r_{k,i} < e_s(\tau_{k',i'})$$

Supposons que $\tau_{k',i'-\delta_{k'}}$ existe. On a alors $r_{k',i'} \geq r_{k'} + P$.

Or, l'instance $\tau_{k',i'}$ est exécutée à l'instant t et l'instance $\tau_{k,i}$ n'est pas exécutée à l'instant t. On a donc $prio(\tau_{k',i'}) > prio(\tau_{k,i})$. Par hypothèse ($\overline{\text{PFI}}$), on obtient aussi $prio(\tau_{k',i'-\delta_{k'}}) > prio(\tau_{k,i-\delta_k})$.

Or, l'instance $\tau_{k',i'-\delta_{k'}}$ n'est pas exécutée à l'instant $t-P$, alors que l'instance $\tau_{k,i-\delta_k}$ est exécutée à l'instant $t-P$. Puisque l'on a $prio(\tau_{k',i'-\delta_{k'}}) > prio(\tau_{k,i-\delta_k})$, on en déduit que l'instance $\tau_{k',i'-\delta_{k'}}$ est terminée à l'instant $t-P$. Or, l'instance $\tau_{k',i'}$ n'est pas terminée à l'instant t. On en déduit :

$$TR_s(\tau_{k',i'}) > TR_s(\tau_{k',i'-\delta_{k'}})$$

Supposons en plus que $r_{k',i'} \geq r_{k,i}$. Or, le temps de réponse de l'instance $\tau_{k',i'}$ augmente par rapport à celui de $\tau_{k',i'-\delta_{k'}}$. De plus par hypothèse, on a $r_{k',i'} \geq r_{k,i} \geq r + P$. Ainsi, l'instance $\tau_{k',i'}$ vérifie les mêmes hypothèses que l'instance $\tau_{k,i}$. On peut donc lui appliquer le raisonnement que l'on vient de mener avec $\tau_{k,i}$.

On en déduit qu'il existe une suite $(u_j)_{0 \leq j \leq l}$ d'instances τ_{α_j,β_j} vérifiant :

1. $u_0 = \tau_{k,i}$, $u_1 = \tau_{k',i'}$
2. $\forall j \in \{1,\ldots,l\}, r_{\alpha_{j-1},\beta_{j-1}} < e_s(\tau_{\alpha_j,\beta_j})$
3. $\forall j \in \{0,\ldots,l-1\}, TR_s(\tau_{\alpha_j,\beta_j}) > TR_s(\tau_{\alpha_j,\beta_j-\delta_{\alpha_j}})$
4. $r_{\alpha_l,\beta_l} < r_{\alpha_l} + P \vee \left[r_{\alpha_l,\beta_l} < r_{k,i} \wedge TR_s(\tau_{\alpha_l,\beta_l}) > TR_s(\tau_{\alpha_l,\beta_l-\delta_{\alpha_l}})\right]$

Etant donné que $r_{k,i} \geq r + P$, la condition (1) implique :

$$r_{\alpha_l,\beta_l} < r_{k,i}$$

Par construction, on peut supposer que l est le seul indice à vérifier $r_{\alpha_l,\beta_l} < r_{k,i}$. On a donc $r_{\alpha_{l-1},\beta_{l-1}} \geq r_{k,i}$. D'après (2), on obtient $r_{\alpha_{l-1},\beta_{l-1}} < e_s(\tau_{\alpha_l,\beta_l})$, et on en déduit $r_{k,i} < e_s(\tau_{\alpha_l,\beta_l})$. Finalement, on obtient :

$$r_{\alpha_l,\beta_l} < r_{k,i} < e_s(\tau_{\alpha_l,\beta_l})$$

En utilisant la condition (4), on en déduit que l'existence de l'instance u_l montre que l'hypothèse de croissance des temps de réponse seulement par effet domino est vérifiée pour $\tau_{k,i}$.

CQFD.

Ainsi, l'étude menée dans la section 4.3 s'applique aux séquences $\overline{\text{PFI}}$. Remarquons aussi que les séquences $\overline{\text{PFI}}$ sont conservatives, ainsi dès que $U \leq P$, on peut affirmer que les temps de réponse induits par une séquence $\overline{\text{PFI}}$ finissent toujours par se stabiliser.

4.4.2 Cyclicité des séquences $\overline{\text{PFI}}$

Dans cette section, nous montrons que pour les séquences $\overline{\text{PFI}}$, la stabilisation des temps de réponse est équivalente à la cyclicité. La cyclicité implique directement la stabilisation des temps de réponse. Nous montrons ici que la réciproque est vérifiée pour les séquences $\overline{\text{PFI}}$.

Nous représentons les séquences d'exécution comme la suite des ensembles des instances exécutées à chaque instant. Enoncer la cyclicité des séquences d'exécution avec ce formalisme revient à dire que les instances exécutées à l'instant t sont deux à deux équivalentes à une métapériode près à celles exécutées à l'instant $t+P$.

Définition 4.2 *Soient s une séquence d'exécution et $t \geq r$. La séquence s est cyclique à partir de l'instant t si et seulement si :*

$$\forall t' \geq t, s_{t'+P} \equiv_1 s_{t'}$$

Notre démarche consiste à passer progressivement de la propriété de stabilité des temps de réponse à celle de la cyclicité de la séquence d'exécution. La première étape consiste à montrer que les instances actives à un instant t sont deux à deux équivalentes aux instances actives à l'instant $t + P$. Etant donné les propriétés portant sur l'attribution des priorités dans $\overline{\text{PFI}}$, on en déduit ensuite que les instances exécutées à un instant t sont deux à deux équivalentes aux instances exécutées à l'instant $t + P$.

Lemme 4.4 *Soient s une séquence d'exécution telle que les temps de réponse sont stables à partir de $t \geq r$, et M un majorant des temps de réponse. Pour tout $t' \geq t + M$, les instances actives à l'instant t' (resp. $t' + P$) sont équivalentes à une métapériode près à celles actives à l'instant $t' + P$ (resp. t').*

Démonstration :
Soient $t' \geq t + M$, et $\tau_{k,i} \in \overline{\tau}$ une instance active à l'instant t'.
D'après le théorème 4.6, les temps de réponse ne peuvent pas diminuer, on a donc :

$$TR_s(\tau_{k,i}) \leq TR_s(\tau_{k,i+\delta_k})$$

L'instance $\tau_{k,i+\delta_k}$ est donc active à l'instant $t' + P$.
Soient $t' \geq t + M + P$, et $\tau_{k,i} \in \overline{\tau}$ une instance active à l'instant t'.
Puisque $\tau_{k,i}$ est active à l'instant t', on a :

$$e_s(\tau_{k,i}) > t' \geq t + M + P$$

Or, M est un majorant des temps de réponse, on en déduit donc :

$$r_{k,i} > t + P$$

D'autre part, les temps de réponse sont stables à partir de t. On a donc :

$$TR_s(\tau_{k,i}) = TR_s(\tau_{k,i-\delta_k})$$

L'instance $\tau_{k,i-\delta_k}$ est donc active à l'instant $t' - P$.
CQFD.

Théorème 4.8 *Soit s une séquence $\overline{PFI}$. Si les temps de réponse sont stables à partir d'un instant $t \geq r$, alors la séquence est cyclique à partir de $t + M$.*

Démonstration :
Les temps de réponse étant stables à partir de t, d'après le lemme 4.4, pour tout $t' \geq t + M$, les tâches actives à l'instant t' sont équivalentes à une métapériode près aux tâches actives à l'instant $t' + P$. Or par hypothèse ($\overline{\text{PFI}}$), les relations de priorités entre les instances actives à l'instant t' sont les mêmes que celles entre les instances à l'instant $t' + P$. Ainsi, les instances les plus prioritaires à l'instant t' sont équivalentes aux instances les plus prioritaires à l'instant $t' + P$. Les instances exécutées étant les p plus prioritaires, on obtient :

$$\forall t' \geq t + M, s_{t'+P} \equiv_1 s_{t'}$$

CQFD.

4.4.3 Intervalle de faisabilité des séquences $\overline{\text{PFI}}$

Les résultats obtenus dans la section 4.4.1 montrent que l'on peut appliquer aux séquences $\overline{\text{PFI}}$ la méthode établie dans la section 4.3. Cela nous conduit aux résultats suivants.

Corollaire 4.3 *Soit s une séquence engendrée par une politique appartenant à $\overline{PFI}$. Dès que $U \leq p$, alors s est cyclique de période P au pire à partir de :*

$$r + P + M + (M-1).P. \sum_{\tau_k \in \tau} \frac{M - C_k}{T_k}$$

où M est un majorant des temps de réponse des instances exécutées dans l'ordre établi par s.

Corollaire 4.4 *Soit s une séquence engendrée par une politique appartenant à $\overline{PFI}$. Dès que s est valide, alors s est cyclique de période P au pire à partir de :*

$$r + P + D + (D-1).P. \sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}$$

Lorsque toutes les échéances D_k des tâches $\tau_k \in \tau$ vérifient $D_k \leq T_k$, on obtient :

$$r + P + D + (D-1).P.(n - U)$$

Corollaire 4.5 *Soit s une séquence engendrée par une politique appartenant à $\overline{PFI}$. L'intervalle suivant est un intervalle de faisabilité pour s :*

$$\left[0, r + P + 2D + (D-1).P. \sum_{\tau_k \in \tau} \frac{D_k - C_k}{T_k}\right[$$

Lorsque toutes les échéances D_k des tâches $\tau_k \in \tau$ vérifient $D_k \leq T_k$, on a aussi :

$$[0, r + P + 2D + (D-1).P.(n - U)[$$

Les politiques à priorités fixes (PFX) appartiennent à $\overline{\text{PFI}}$, les corollaires ci-dessus sont donc valables pour toutes les politiques à priorités fixes, par exemple RM et DM. Nous avons montré dans le chapitre 3 que EDF appartient à $\overline{\text{PFI}}$, ainsi les corollaires ci-dessus sont donc aussi valables pour les séquences produites par EDF.

4.5 Conclusion

Nous avons mis en place une nouvelle approche de l'ordonnancement multiprocesseur des systèmes temps-réel composés de tâches périodiques. Nous avons établi, sous les hypothèses décrites dans la section 4.2, les résultats suivants pour les systèmes de tâches vérifiant $U \leq p$:

1. pour toute séquence conservative, les temps de réponse se stabilisent toujours, de plus, nous avons borné l'instant où cette stabilisation apparaît en utilisant les paramètres temporels des tâches et un majorant des temps de réponse,
2. pour toute séquence valide, les temps de réponse se stabilisent toujours, de plus, nous avons borné l'instant où cette stabilisation apparaît en utilisant seulement les paramètres des tâches,
3. les séquences d'exécution admettent un intervalle de faisabilité qui ne dépend que des paramètres des tâches.

Nous avons ensuite montré que les séquences d'exécution $\overline{\text{PFI}}$ vérifient nos deux hypothèses de travail. Nous en avons alors déduit les points suivants pour les systèmes de tâches vérifiant $U \leq p$:

1. les séquences d'exécution $\overline{\text{PFI}}$ sont cycliques de période P,

2. la durée de montée en charge des séquences $\overline{\text{PFI}}$ admet une borne qui ne dépend que des paramètres des tâches et du plus grand temps de réponse, pour les séquences valides, elle ne dépend que des paramètres des tâches,
3. les séquences d'exécution $\overline{\text{PFI}}$ admettent un intervalle de faisabilité qui ne dépend que des paramètres des tâches.

Dans la section 4.3.1, nous avons montré que les temps de réponse sont bornés dès que le système n'est pas surchargé ($U \leq p$). Nous avons établi ce résultat pour les séquences d'exécution conservatives qui vérifient nos deux hypothèses de travail. Toutefois, l'hypothèse de conservativité dépend de l'exécutif ciblé et des interactions entre les tâches. Lorsque les tâches sont indépendantes et que l'exécutif ciblé est préemptif avec migration totale, ou non-préemptif, notre résultat est directement applicable. Pour d'autres contextes, notre résultat doit par contre être adapté. D'autre part, nous n'avons pas donné de formule précisant la borne des temps de réponse. Une telle formule permettrait d'obtenir une expression de la borne de la durée de montée en charge des séquences d'exécution non-valides. On obtiendrait alors un intervalle d'étude pour les séquences d'exécution non-valides, et on pourrait alors établir un diagnostic fini traitant toute la séquence d'exécution. Lors de nos recherches, nous avons pressenti que $\max\{T_k | \tau_k \in \tau\}$ constitue une borne des temps de réponse pour toute séquence conservative dès que $U \leq p$; cependant, ceci reste à prouver.

Dans la section 4.4, nous montrons que nos deux hypothèses sont vérifiées par les séquences d'exécution préemptives avec migration totale engendrées par les politiques d'ordonnancement de $\overline{\text{PFI}}$ (par exemple RM, DM et EDF). Nous avons aussi évoqué via un exemple que LLF vérifie nos deux hypothèses de travail, celles-ci sont donc plus générales que le contexte $\overline{\text{PFI}}$. Une voie intéressante de recherche consiste à déterminer la véritable portée des deux hypothèses que nous avons formulées : quel est leur niveau de généralité ? Pour cela, on peut montrer que LLF les vérifie, comme le suggère l'exemple 4.3. Plus généralement, on peut aussi étudier si elles sont vérifiées par les séquences produites par les politiques d'ordonnancement de DCL. On peut aussi étendre la portée de notre analyse en étudiant les séquences d'exécution obtenues lorsque l'exécutif est non-préemptif ou préemptif avec migration partielle. Finalement, on peut aussi prendre en compte les interactions entre les tâches, et notamment, étudier l'impact du blocage d'une tâche par une autre sur nos deux hypothèses de travail.

L'intervalle de faisabilité que nous proposons ne dépend que de nos deux hypothèses de travail. En cela, il a le même niveau de généralité qu'elles. En particulier, il ne dépend pas de la conservativité des séquences d'exécution comme notre résultat portant sur l'existence d'une borne du temps de réponse. A priori, il est donc applicable dans de nombreux contextes :

- exécutif temps-réel préemptif ou non, avec migration ou non, etc,
- politiques d'ordonnancement appartenant à PFX, $\overline{\text{PFI}}$, DCL, etc,
- systèmes de tâches à échéances sur, ou avant, ou même après requête,
- systèmes de tâches indépendantes ou interdépendantes.

En se restreignant à des contextes particuliers, on doit pouvoir obtenir des intervalles de faisabilité beaucoup plus précis. Par exemple, son expression utilise les paramètres D_k et C_k, cependant ils interviennent ici comme un majorant et un minorant des temps de réponse. Les valeurs D_k et C_k correspondent alors à des valeurs extrémales. Pour certains contextes, on doit sans doute pouvoir calculer des majorants et des minorants des temps de réponse qui sont beaucoup plus précis, et en cela améliorer l'intervalle de faisabilité que nous proposons.

Les travaux menés dans ce chapitre ont fait l'objet d'un article soumis à une revue internationale en 2007.

Chapitre 5

Durée de montée en charge des ordonnancements en priorités fixes de tâches périodiques

5.1 Introduction

Pour déterminer un intervalle de faisabilité, on suit généralement la méthode suivante. On commence par prouver que l'exécution d'un système temps-réel est cyclique, ensuite on majore la durée permettant d'atteindre le régime cyclique, et on en déduit finalement l'intervalle recherché. La durée d'exécution nécessaire pour atteindre le régime cyclique a donc une grande importance pour l'étude des intervalles de faisabilité. Généralement, on utilise des majorants de cette durée. Dans le chapitre 4, nous avons proposé une étude de ce problème applicable dans de nombreux contextes. Dans ce chapitre, nous nous focalisons sur l'un des plus utilisés dans le domaine du temps-réel : celui des séquences multiprocesseurs produites par des politiques à priorités fixes en contexte préemptif avec migration totale.

Dans [42] les auteurs proposent une borne de la durée de montée en charge des séquences produites par les politiques à priorités fixes en contexte monoprocesseur et préemptif. Cette borne a été étendue au contexte multiprocesseur dans le cas où les échéances des tâches sont plus petites ou égales à leur période [24], et dans le cas où les échéances des tâches peuvent être plus grandes que leur période [25].

Dans ce chapitre, nous affinons les résultats établis dans [24] en formulant la durée exacte de montée en charge des séquences préemptives multiprocesseurs produites par des politiques à priorités fixes. Toutefois, l'expression de cette durée est de complexité exponentielle. Nous en dérivons alors plusieurs majorations de complexité polynomiale et nous donnons aussi un algorithme permettant d'en calculer la valeur exacte. La complexité au pire du calcul de la valeur exacte est exponentielle, mais les expérimentations que nous avons menées montrent qu'en pratique sa complexité moyenne est polynomiale.

Nous étudions la durée exacte de montée en charge dans la section 5.3. Après, nous déterminons plusieurs majorations de cette durée dans les sections 5.4, 5.5, et 5.6. Finalement, dans la section 5.7, nous donnons un algorithme permettant de calculer la durée exacte de montée en charge. Nous concluons par des tests permettant d'évaluer la qualité des majorants que nous proposons et aussi d'évaluer la complexité de leur calcul.

5.2 Contexte d'étude

Nous nous plaçons dans le cadre de l'approche globale (une même tâche peut migrer entre les processeurs), et sous l'hypothèse de migration totale (une même instance peut changer de

processeurs en cours d'exécution). D'autre part, les tâches sont supposées être indépendantes et préemptibles. Nous supposons aussi que les échéances sont plus petites ou égales aux périodes, on a alors :

$$\forall \tau_k \in \tau, C_k \leq D_k \leq T_k$$

Dans la suite, nous ne considérons que les séquences d'exécution obtenues à l'aide d'une politique d'ordonnancement à priorités fixes, nous désignons par PFX l'ensemble de ces séquences. Ainsi, pour chaque séquence $s \in PFX$, on peut associer à chaque tâche $\tau_k \in \tau$ sa priorité $prio_s(\tau_k)$. D'autre part, nous supposons pour une séquence $s \in PFX$ donnée que les tâches sont ordonnées par priorité décroissante :

$$\forall k \in \{1, \ldots, n-1\}, prio_s(\tau_k) > prio_s(\tau_{k+1})$$

5.3 Etude de la durée de montée en charge

Avant de commencer notre étude, nous devons introduire deux notions. La première décompose une séquence d'exécution pour se focaliser sur les instances d'une même tâche. La deuxième identifie la première instance d'une tâche qui est activée après la fin de montée en charge propre à cette tâche-là.

Définition 5.1 *Soient $s \in PFX$ et $\tau_k \in \tau$. On note s^k la séquence d'exécution obtenue en restreignant s aux seules instances de la tâche τ_k.*

Définition 5.2 *Soient $s \in PFX$ et $\tau_k \in \tau$. La première instance stable $\tau_{k,i}$ de τ_k est la première instance (relativement à i) à satisfaire la propriété suivante :*

$$\forall t \geq r_{k,i}, s_t^k \equiv_1 s_{t+P}^k$$

5.3.1 Date de stabilisation d'une tâche $\tau_k \in \tau$

La connaissance de la première instance stable $\tau_{k,i}$ d'une tâche $\tau_k \in \tau$ permet d'affirmer que la séquence d'exécution restreinte aux seules instances de τ_k est cyclique depuis $r_{k,i}$ (par définition). Cependant, l'instant $r_{k,i}$ n'est pas en général le début du régime cyclique de s^k : celui-ci peut commencer plus tôt. Effectivement, les instants précédant $r_{k,i}$ où aucune instance de τ_k n'est exécutée peuvent faire partie du régime cyclique de s^k.

Le résultat suivant indique que deux instances équivalentes à une métapériode près et dont les temps de réponse sont égaux sont exécutées aux mêmes instants, relativement à leur date d'activation respective. Cette propriété est importante puisqu'elle établit un lien entre l'égalité du temps de réponse de deux instances équivalentes et la cyclicité de leur exécution.

Lemme 5.1 *Soient $s \in PFX$ et $\tau_{k,i} \in \overline{\tau}$ telle que $TR_s(\tau_{k,i}) = TR_s(\tau_{k,i+\delta_k})$. On a :*

$$\forall t \in \mathbb{N}, \tau_{k,i} \in s_t \Leftrightarrow \tau_{k,i+\delta_k} \in s_{t+P}$$

Démonstration :

Considérons un instant $t \in \mathbb{N}$. Notons E_1 l'ensemble des instances actives à l'instant t et plus prioritaires que $\tau_{k,i}$:

$$E_1 = \{\tau_{k',i'} \in s_t^\star | prio_s(\tau_{k',i'}) > prio(\tau_{k,i})\}$$

De même, notons E_2 l'ensemble des instances actives à l'instant $t+P$ et plus prioritaires que $\tau_{k,i+\delta_k}$:

$$E_2 = \{\tau_{k',i'} \in s_{t+P}^\star | prio_s(\tau_{k',i'}) > prio(\tau_{k,i+\delta_k})\}$$

Premièrement, par définition, une instance $\tau_{k',i'} \in \overline{\tau}$ est active aux seuls instants $t' \in \mathbb{N}$ vérifiant :

$$r_{k',i'} \leq t' < r_{k',i'} + TR_s(\tau_{k',i'})$$

D'après le théorème 4.6, les temps de réponse ne peuvent pas diminuer, on a donc :

$$\tau_{k',i'} \in s^{\star}_{t'} \Rightarrow \tau_{k',i'+\delta_{k'}} \in s^{\star}_{t'+P}$$

Deuxièmement, la séquence s est produite par PFX, on a donc :

$$prio_s(\tau_{k,i}) = prio_s(\tau_{k,i+\delta_k}) = prio_s(\tau_k)$$

On en déduit que $|E_1| \leq |E_2|$.

Or par hypothèse, on a $TR_s(\tau_{k,i}) = TR_s(\tau_{k,i+\delta_k})$, les instances $\tau_{k,i}$ et $\tau_{k,i+\delta_k}$ sont donc actives pendant la même durée. Avec ce qui précède, on obtient :

$$\tau_{k,i+\delta_k} \in s_{t+P} \Leftrightarrow |E_2| < p \Rightarrow |E_1| < p \Leftrightarrow \tau_{k,i} \in s_t$$

Ainsi, dès que l'instance $\tau_{k,i+\delta_k}$ est exécutée à un instant $t+P$, alors l'instance $\tau_{k,i}$ est exécutée à l'instant t. Or, ces deux instances sont actives pendant la même durée et elles sont exécutées toutes les deux exactement C_k fois. Elles sont donc exécutées aux mêmes instants (relativement à leur date d'activation respective).

CQFD.

Le résultat suivant indique l'instant exact où le régime cyclique de s^k commence.

Lemme 5.2 *Soient $s \in PFX$ une séquence d'exécution valide et $\tau_{k,i}$ la première instance stable de $\tau_k \in \tau$. L'instant exact t où le régime cyclique de la séquence s^k commence est :*

$$t = \max\{0, r_{k,i} - (T_k - TR_s(\tau_{k,i-1+\delta_k}))\}$$

Démonstration :

Etant donné que la séquence s est valide et que les échéances sont contraintes ($D_k \leq T_k$), il existe à chaque instant au plus une instance active de chaque tâche. En conséquence, les différentes instances d'une même tâche ne peuvent pas interférer directement entre elles : chaque instance se termine avant que la suivante soit activée.

Supposons que $i = 0$.

La séquence s restreinte aux seules instances de τ_k est donc cyclique dès $t = r_k$. Toutefois, aux instants précédant r_k, aucune instance de τ_k n'est exécutée. De plus, puisque les différentes instances d'une même tâche n'interfèrent pas directement entre elles, aucune instance de τ_k n'est exécutée aux instants t vérifiant :

$$r_{k,\delta_k} - (T_k - TR_s(\tau_{k,\delta_k-1})) \leq t < r_{k,\delta_k}$$

Ainsi, la séquence s^k est cyclique dès :

$$t = \max\{r_k - (T_k - TR_s(\tau_{k,\delta_k-1}))\}$$

D'autre part, à l'instant précédent, aucune instance de τ_k n'est exécutée, alors qu'à la métapériode suivante, l'instance τ_{k,δ_k-1} est exécutée. Ainsi la séquence s^k est cyclique exactement à partir de :

$$t = \max\{r_k - (T_k - TR_s(\tau_{k,\delta_k-1}))\}$$

Supposons que $i > 0$.

Par hypothèse, l'instance $\tau_{k,i}$ est la première instance stable de τ_k. La séquence s^k est donc cyclique au pire à partir de $r_{k,i}$. De plus, l'instance $\tau_{k,i-1}$ n'est pas stable. Ainsi, il existe des instants t où $\tau_{k,i-1}$ est exécutée sans que $\tau_{k,i-1+\delta_k}$ soit exécutée en $t+P$. D'après le lemme 5.1, le temps de réponse de $\tau_{k,i-1}$ est alors différent de celui de $\tau_{k,i-1+\delta_k}$. Et d'après le théorème 4.6, il ne peut être que plus petit. On a donc :

$$TR_s(\tau_{k,i-1}) < TR_s(\tau_{k,i-1+\delta_k})$$

A l'instant $e_s(\tau_{k,i-1+\delta_k}) - 1$, l'instance $\tau_{k,i-1+\delta_k}$ est exécutée. D'autre part, à l'instant $e_s(\tau_{k,i-1+\delta_k}) - 1 - P$, l'instance $\tau_{k,i-1}$ n'est pas exécutée puisque son temps de réponse est strictement inférieur. Ainsi, à l'instant $e_s(\tau_{k,i-1+\delta_k}) - 1$, la séquence s^k n'est pas cyclique.

Pour les instants t vérifiant $e_s(\tau_{k,i-1+\delta_k}) \leq t < r_{k,i+\delta_k}$, aucune instance de τ_k ne peut être exécutée puisque les différentes instances d'une même tâche n'interfèrent pas directement entre elles. De même, pour les instants t vérifiant $e_s(\tau_{k,i-1+\delta_k}) - P \leq t < r_{k,i}$, aucune instance de τ_k ne peut être exécutée puisque le temps de réponse de $\tau_{k,i-1+\delta_k}$ est strictement supérieur à celui de $\tau_{k,i-1}$. Ainsi la séquence s^k est cyclique exactement à partir de :

$$t = r_{k,i} - (T_k - TR_s(\tau_{k,i-1+\delta_k}))$$

CQFD.

5.3.2 Cas particulier des p tâches les plus prioritaires

Sur une architecture à p processeurs et lorsque pour chaque tâche $\tau_k \in \tau$ on a $C_k \leq T_k$, les p tâches les plus prioritaires sont dans une situation spéciale : dès qu'une instance de l'une d'elles est activée, elle est exécutée sans interruption puisqu'à chaque instant de son exécution, il ne peut pas y avoir plus de $p-1$ instances actives et plus prioritaires qu'elle. Le lemme suivant traduit ce résultat. Ensuite, nous donnons la durée exacte de montée en charge des systèmes composés d'au plus p tâches.

Lemme 5.3 *Soit $s \in PFX$. On a :*

$$\forall \tau_{k,i} \in \overline{\tau}, 1 \leq k \leq p \Rightarrow TR_s(\tau_{k,i}) = C_k$$

Démonstration :
Lorsque l'instance $\tau_{1,0}$ est activée, il n'existe aucune instance plus prioritaire. La prochaine instance aussi prioritaire qu'elle est $\tau_{1,1}$. Or, celle-ci est activée T_k unités de temps après $\tau_{1,0}$. Etant donné que $C_1 \leq T_1$, l'instance $\tau_{1,0}$ est donc exécutée sans interruption et on a $TR_s(\tau_{1,0}) = C_1$. Cette situation se répète pour chaque instance de τ_1, on a donc :

$$\forall \tau_{1,i} \in \overline{\tau}, TR_s(\tau_{1,i}) = C_1$$

Supposons maintenant que le lemme est vérifié pour les K tâches les plus prioritaires avec $K < p$, et montrons qu'il est vérifié pour la tâche τ_{K+1}.
D'après l'hypothèse de récurrence, on a :

$$\forall \tau_{k,i} \in \overline{\tau}, 1 \leq k \leq K \Rightarrow TR_s(\tau_{k,i}) = C_k$$

Comme on a $C_k \leq D_k \leq T_k$ pour chaque tâche $\tau_k \in \tau$, on en déduit qu'il existe à chaque instant $t \in \mathbb{N}$ au plus K instances actives engendrées par les tâches plus prioritaires que τ_{K+1}. Etant donné que $K < p$, on obtient de la même manière que pour les instances de τ_1 que le temps de réponse des instances de τ_{K+1} est C_{K+1}.
CQFD.

Le lemme 5.3 montre que l'ordre de priorité des p tâches les plus prioritaires n'influe pas sur les temps de réponse des instances de ces tâches-là. Ainsi, les instants où les instances des tâches moins prioritaires peuvent accéder aux processeurs ne dépendent pas de l'ordre de priorité des p tâches les plus prioritaires. En inversant la priorité de deux tâches faisant partie des p plus prioritaires, on ne modifie donc pas la séquence d'exécution. Le nombre de configurations de priorité pour un système de n tâches est $n!$. Le lemme 5.3 montre donc que l'on peut réduire le nombre de configurations utiles à $n!/p!$.

Théorème 5.1 *En contexte préemptif avec migration totale, le nombre de configurations de priorité fixe utiles est $\frac{n!}{p!}$.*

En combinant les lemmes 5.2 et 5.3, on obtient la durée de montée en charge exacte des systèmes composés d'au plus p tâches.

Théorème 5.2 *Soit $s \in PFX$. Si le nombre n de tâches de τ est inférieur ou égal au nombre p de processeurs, alors s est valide et cyclique de période P exactement à partir de :*

$$t = \max\{0, r_1 - (T_1 - C_1), \ldots, r_n - (T_n - C_n)\}$$

Démonstration :
Etant donné que $n \leq p$, d'après le lemme 5.3, on a :

$$\forall \tau_{k,i} \in \overline{\tau}, TR_s(\tau_{k,i}) = C_k$$

Ainsi, la séquence s est valide et les premières instances stables des tâches $\tau_k \in \tau$ sont $\tau_{k,0}$.

D'après le lemme 5.2, chaque séquence s^k est cyclique exactement à partir de $t = r_k - (T_k - TR_s(\tau_{k,\delta_k - 1}))$. Puisque le temps de réponse des instances $\tau_{k,i} \in \overline{\tau}$ est toujours C_k, la séquence d'exécution s est donc cyclique exactement à partir de :

$$t = \max\{0, r_1 - (T_1 - C_1), \ldots, r_n - (T_n - C_n)\}$$

CQFD.

5.3.3 Borne minimale de la durée de montée en charge

Le lemme 5.2 donne la durée exacte de montée en charge de la séquence s^k lorsque l'on connaît la première instance stable de la tâche τ_k. Au plus tôt, cette instance sera $\tau_{k,0}$, on obtient ainsi une borne minimale de la durée de montée en charge.

Théorème 5.3 *Soit $s \in PFX$ une séquence d'exécution valide. La séquence est cyclique de période P au plus tôt à partir de :*

$$t = \max\{0, r_1 - (T_1 - C_1), \ldots, r_n - (T_n - C_n)\}$$

Démonstration :
Considérons $\tau_{k,i}$ la première instance stable de la tâche τ_k. D'après le lemme 5.2, la séquence d'exécution s^k est cyclique exactement à partir de l'instant :

$$t = \max\{0, r_{k,i} - (T_k - TR_s(\tau_{k,i-1+\delta_k}))\}$$

Le temps de réponse des instances de τ_k étant minoré par C_k, on obtient :

$$t \geq \max\{0, r_{k,i} - (T_k - C_k)\} \geq \max\{0, r_k - (T_k - C_k)\}$$

CQFD.

5.3.4 Durée exacte de montée en charge

La durée exacte de montée en charge dépend de la stabilisation de chaque tâche. Cependant, la stabilisation d'une tâche ne dépend pas de la stabilisation des tâches qui sont de plus faibles priorités. On peut donc déterminer incrémentalement la durée de montée en charge.

Théorème 5.4 *Soit $s \in PFX$ une séquence d'exécution valide. La séquence s est cyclique de période P exactement à partir de l'instant $t = u_n$ où la suite $(u_k)_{k \in \{1,\ldots,n\}}$ est définie récursivement comme suit :*

– *pour* $k \in \{1, \ldots, p\}$, *on a :*

$$u_k = \max\{0, r_1 - (T_1 - C_1), \ldots, r_k - (T_k - C_k)\} \tag{5.1}$$

– *pour* $k \in \{p+1, \ldots, n\}$, *on a :*
- *si* $r_k \geq u_{k-1}$, *on a :*

$$u_k = \max\{u_{k-1}, r_k - (T_k - TR_s(\tau_{k,\delta_k - 1}))\} \tag{5.2}$$

- *si* $r_k < u_{k-1}$, *posons* $i = \lceil (u_{k-1} - r_k)/T_k \rceil > 0$, *on a :*
  - *si* $TR_s(\tau_{k,i-1}) \neq TR_s(\tau_{k,i-1+\delta_k})$, *on a :*

$$u_k = \max\{u_{k-1}, r_{k,i-1} + TR_s(\tau_{k,i-1+\delta_k})\} \tag{5.3}$$

  - *si* $TR_s(\tau_{k,i-1}) = TR_s(\tau_{k,i-1+\delta_k})$, *on a :*

$$u_k = u_{k-1} \tag{5.4}$$

Démonstration :
Pour k vérifiant $1 \leq k \leq p$, le théorème 5.2 montre le résultat.
Raisonnons maintenant par récurrence sur le nombre de tâches.
Soit $K \in \{1, \ldots, n\}$, et supposons que la séquence s restreinte aux instances des $K-1$ tâches plus prioritaires, est cyclique de période P exactement à partir d'un instant $\Delta \in \mathbb{N}$. Donc à partir de Δ, les instants où le nombre d'instances actives plus prioritaires que τ_K est strictement inférieur à p sont cycliques de période P.
D'autre part, la séquence s est valide et les échéances sont contraintes ($D_k \leq T_k$). On a donc aussi :

$$\forall \tau_{k,i} \in \tau, TR_s(\tau_{k,i}) \leq T_k$$

Ainsi, à chaque instant t, il existe au plus une instance de chaque tâche qui est active. Les différentes instances d'une même tâche ne peuvent donc pas interférer directement entre elles.
Finalement, à partir de Δ, les instants où une instance $\tau_{K,i}$ peut être exécutée sont les mêmes à une métapériode près que ceux où $\tau_{K,i+\delta_K}$ peut être exécutée. Donc, toutes les instances de τ_K activées après ou en Δ sont stables.
Si on a $r_K \geq \Delta$. Ainsi, toutes les instances de τ_K sont activées après Δ. D'après ce qui précède, la première instance stable de τ_K est donc $\tau_{K,0}$. Le lemme 5.2 montre que la séquence s^K est cyclique exactement à partir de $t = \max\{0, r_K - (T_K - TR_s(\tau_{K,\delta_K - 1}))\}$. La séquence s restreinte aux K instances les plus prioritaires est donc cyclique de période P exactement à partir de :

$$t = \max\{\Delta, r_K - (T_K - TR_s(\tau_{K,\delta_K - 1}))\}$$

Si on a $r_k < \Delta$. Considérons la première instance $\tau_{k,i}$ activée après ou en Δ, on a $i = \lceil (\Delta - r_k)/T_k \rceil > 0$, avec la convention $\forall x \in \mathbb{N}, x = \lfloor x \rfloor = \lceil x \rceil$.
Si $TR_s(\tau_{K,i-1}) \neq TR_s(\tau_{K,i-1+\delta_K})$, alors d'après le lemme 5.1, $\tau_{K,i}$ est la première instance stable de τ_K. Comme $i > 0$, d'après le lemme 5.2, la séquence s^K est cyclique de période P exactement à partir de $t = r_{K,i} - (T_k - TR_s(\tau_{K,i-1+\delta_K}))$. La séquence s restreinte aux K instances les plus prioritaires est donc cyclique de période P exactement à partir de :

$$t = \max\{\Delta, r_{K,i-1} + TR_s(\tau_{K,i-1+\delta_K}))\}$$

Si $TR_s(\tau_{K,i-1}) = TR_s(\tau_{K,i-1+\delta_K})$. Considérons $\tau_{K,j}$ la première instance stable de τ_K. D'après le lemme 5.1, l'instance $\tau_{K,i-1}$ est stable, on a alors $j < i$. D'après le lemme 5.2, la séquence s^K est cyclique de période P exactement à partir de $t = \max\{0, r_{K,j} - (T_K - TR_s(\tau_{K,j-1+\delta_K}))\}$. Si $t = 0$, on a alors $t \leq \Delta$, sinon on obtient :

$$t = r_{K,j-1} + TR_s(\tau_{K,j-1+\delta_K})) \leq r_{K,j} < \Delta$$

La séquence s restreinte aux K tâches les plus prioritaires est donc cyclique de période P exactement à partir de Δ.

CQFD.

Le théorème 5.4 fournit l'instant exact d'entrée dans le régime cyclique. Cependant, le calcul de cet instant nécessite la connaissance du temps de réponse de certaines instances, on ne peut donc pas l'utiliser directement. Dans les sections 5.4, 5.5, et 5.6, nous développons plusieurs majorations de la durée de montée en charge qui admettent des complexités différentes. Dans la section 5.7, nous proposons un algorithme pour calculer le temps de réponse exact d'une instance quelconque qui nous permettra finalement de calculer la durée exacte de montée en charge.

5.4 Majoration du temps de réponse par les échéances relatives

Le théorème 5.4 permet de calculer la durée exacte de montée en charge. Cependant, le temps de réponse de certaines instances doit être connu. Dans cette section, nous donnons une approximation de la formule donnée dans le théorème 5.4 en utilisant le fait que pour une séquence d'exécution valide, le temps de réponse des instances est majoré par leur échéance relative.

Le tableau suivant résume les modifications apportées à la méthode de calcul fournie par le théorème 5.4 :

Cas du théorème 5.4	Modifications effectuées	Relation avec la valeur exacte
(5.1)	$v_k = \max\{0, r_1 - (T_1 - C_1), \ldots, r_k - (T_k - C_k)\}$	$v_k = u_k$
(5.2)	$v_k = \max\{v_{k-1}, r_k - (T_k - D_k)\}$	$v_k \geq u_k$
(5.3) & (5.4)	$v_k = \max\{v_{k-1}, r_{k,i-1} + D_k)\}$	$v_k \geq u_k$

```
DMC_n()
 M = 0
 for k from 1 to n do
   if k ≤ p
   then M = max(M,r_k − (T_k − C_k))
   else
     if r_k ≥ M
     then M = max(M,r_k − (T_k − D_k))
     else M = max(M,r_{k,⌈(M−r_k)/T_k⌉} − (T_k − D_k))
 return M
```

FIG. 5.1 – Approximation linéaire du calcul exact

```
CG06a() [24]
  M = r_1
  for k from 2 to n do
    if r_k ≥ M
    then M = r_k
    else M = r_{k,⌈(M−r_k)/T_k⌉}
  return M
```

FIG. 5.2 – Approximation linéaire du calcul exact fournie par [24]

En utilisant ces majorations, on obtient l'algorithme indiqué sur la figure 5.1. Récursivement, on montre :

$$\forall k \in \{1, \ldots, n\}, v_k \geq u_k$$

Grâce à cette propriété, on obtient le résultat suivant. Etant donné que nous ne le démontrons pas formellement, nous l'énonçons sous la forme d'une proposition.

Proposition 5.1 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\texttt{DMC\_n}() \geq u_n$$

L'algorithme `DMC_n` détermine un majorant de la durée de montée en charge par un calcul dont la complexité est en $O(n)$. [24] donne une autre méthode de même complexité pour résoudre ce problème (voir figure 5.2). Les algorithmes `DMC_n` et `CG06a` diffèrent sur deux points. D'une part, l'algorithme `CG06a` ne prend pas en compte le fait que le régime cyclique d'une tâche puisse commencer avant sa date d'activation (lemme 5.2). D'autre part, il ne prend pas en compte la situation particulière des p tâches les plus prioritaires (lemme 5.3). Récursivement, on montre que l'algorithme `DMC_n` fournit une majoration de la durée de montée en charge qui est toujours inférieure ou égale à celle de `CG06a`.

5.5 Majoration du temps de réponse par les instances inactives

La durée de montée en charge dépend du temps de réponse de certaines instances. Dans la section 5.4, nous avons donné une majoration de cette durée en majorant le temps de réponse des instances par leur échéance relative. Dans cette section, nous majorons plus précisément le temps de réponse des instances : pour cela, nous examinons à un instant donné le nombre d'instances qui sont peut être encore actives, c'est-à-dire celles dont l'échéance n'est pas encore passée.

Considérons une séquence $s \in PFX$ et une instance $\tau_{k,i} \in \overline{\tau}$. Par définition, l'instance $\tau_{k,i}$ est active aux seuls instants $t \in \mathbb{N}$ vérifiant :

$$r_{k,i} \leq t < e_s(\tau_{k,i})$$

Dès que la séquence d'exécution s est valide, on peut majorer $e_s(\tau_{k,i})$ par $d_{k,i}$. Ainsi, dans l'intervalle $[d_{k,i}, r_{k,i+1}[$, l'instance $\tau_{k,i}$ n'est pas active. Inversement, dans l'intervalle $[r_{k,i}, d_{k,i}[$, l'instance $\tau_{k,i}$ est peut-être active : dans la suite, nous disons qu'elle est *potentiellement active.*

A un instant $t \in \mathbb{N}$, on peut déterminer le nombre d'instances qui sont potentiellement actives :

$$|\{\tau_{k,i} \in \overline{\tau} | r_{k,i} \leq t < d_{k,i}\}|$$

```
eval_TR(τ_{k,i})
  nb_acces = 0, t = r_{k,i}
  while ((nb_acces < C_k) and (t < d_{k,i})) do
    many = 0, j = 1
    while ((many < p) and (j < k)) do
      if (t ≥ r_j) then
        if (j ≤ p) and (((t − r_j) mod T_j) < C_j) then many = many + 1
        if (j > p) and (((t − r_j) mod T_j) < D_j) then many = many + 1
      j = j+1
    if (many < p) then nb_acces = nb_acces+1
    t = t+1
  return t - r_{k,i}
```

FIG. 5.3 – Majoration du temps de réponse d'une instance

Considérons maintenant une instance $\tau_{k,i} \in \overline{\tau}$ active à un instant $t \in \mathbb{N}$. Avec ce qui précède, on obtient une condition suffisante pour que l'instance $\tau_{k,i}$ soit exécutée à l'instant t. Effectivement, dès que le nombre d'instances plus prioritaires que $\tau_{k,i}$ et potentiellement actives à l'instant t est strictement inférieur à p, alors l'instance $\tau_{k,i}$ est forcément exécutée à l'instant t.

De plus, d'après le lemme 5.3, on a pour les instances $\tau_{k,i}$ des p tâches les plus prioritaires :

$$\forall t \in \mathbb{N}, \tau_{k,i} \in s_t \Leftrightarrow r_{k,i} \leq t < r_{k,i} + C_k$$

On obtient alors l'algorithme indiqué sur la figure 5.3 pour majorer le temps de réponse d'une instance lorsque la séquence d'exécution est valide. Les remarques faites lors de la présentation de l'algorithme `eval_TR` permettent d'assurer que la valeur retournée est bien un majorant du temps de réponse de l'instance donnée en paramètre. Le résultat suivant est donc vérifié.

Proposition 5.2 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\forall \tau_{k,i} \in \overline{\tau}, TR_s(\tau_{k,i}) \leq \texttt{eval\_TR}(\tau_{k,i}) \leq D_k$$

A l'aide de l'algorithme `eval_TR`, nous pouvons majorer la durée fournie par le théorème 5.4 plus finement que dans la section 5.4. Les modifications que nous apportons sont essentiellement du même ordre que dans la section 5.4. Toutefois, nous distinguons en plus un cas particulier de (5.4). Lorsque $\texttt{eval\_TR}(\tau_{k,i-1+\delta_k}) = C_k$, on a aussi $TR_s(\tau_{k,i-1+\delta_k}) = C_k$. Puisque les temps de réponse ne peuvent pas diminuer (lemme 4.6), on en déduit :

$$TR_s(\tau_{k,i-1+\delta_k}) = TR_s(\tau_{k,i-1}) = C_k$$

On peut aussi être tenté d'utiliser une propriété similaire : $\texttt{eval\_TR}(\tau_{k,i-1}) = D_k$. Cependant, `eval_TR` détermine un majorant du temps de réponse de $\tau_{k,i-1}$, on ne peut donc pas en déduire $TR_s(\tau_{k,i-1}) = D_k$. Ce cas ne permet donc pas d'assurer $TR_s(\tau_{k,i-1+\delta_k}) = TR_s(\tau_{k,i-1})$.

Le tableau suivant résume les modifications que nous effectuons :

Cas du théorème 5.4	Modifications effectuées	Relation avec la valeur exacte
(5.1)	$v_k = \max\{0, r_1 - (T_1 - C_1), \ldots, r_k - (T_k - C_k)\}$	$v_k = u_k$
(5.2)	$v_k = \max\{v_{k-1}, r_k - (T_k - \texttt{eval\_TR}(\tau_{k,\delta_k - 1}))\}$	$v_k \geq u_k$
(5.3)	$v_k = \max\{v_{k-1}, r_{k,i-1} + \texttt{eval\_TR}(\tau_{k,i-1+\delta_k})\}$	$v_k \geq u_k$
(5.4)	$\texttt{eval\_TR}(\tau_{k,i-1+\delta_k}) > C_k$ $\Rightarrow v_k = \max\{v_{k-1}, r_{k,i-1} + \texttt{eval\_TR}(\tau_{k,i-1+\delta_k})\}$	$v_k \geq u_k$
	$\texttt{eval\_TR}(\tau_{k,i-1+\delta_k}) = C_k$ $\Rightarrow v_k = v_{k-1}$	$v_k = u_k$

Nous obtenons alors l'algorithme indiqué sur la figure 5.4 pour approcher le résultat fourni par le théorème 5.4. Récursivement, on montre :

$$\forall k \in \{1, \ldots, n\}, v_k \geq u_k$$

On obtient alors le résultat suivant.

Proposition 5.3 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\texttt{DMC\_Dn2}() \geq u_n$$

Le calcul de $\texttt{eval\_TR}(\tau_{k,i})$ a une complexité majorée par $(k-1).D_k$. Pour $k > p$, l'algorithme `DMC_n2` nécessite exactement un appel à `eval_TR`. La complexité de calcul de `DMC_Dn2` est donc majorée par :

$$p + \sum_{k=p+1}^{n} (k-1).D_k \leq p + D. \sum_{k=p+1}^{n} (k-1) = p + D\frac{n^2 - p^2 - (n-p)}{2}$$

Lorsque $n \leq p$, le théorème 5.2 fournit la durée exacte de montée en charge par un calcul dont la complexité est en $O(n)$. Dans le cas présent, on peut donc supposer $n \geq p$. L'ordre de grandeur de la complexité de `DMC_Dn2` est alors majoré par :

$$O(D.n^2)$$

5.6 Utilisation des instances vérifiant $TR_s(\tau_{k,i}) = C_k$

Le théorème 5.2 montre que le temps de réponse des instances $\tau_{k,i}$ des p tâches les plus prioritaires est égal à C_k. Toutefois, d'autres tâches peuvent aussi vérifier cette propriété. Dans cette section, nous déterminons la plus longue suite $\tau_1, \ldots, \tau_K$ telle que le temps de réponse des instances $\tau_{k,i}$ de ces tâches soit égal à C_k. La connaissance de la valeur de K permet d'améliorer la majoration de la durée de montée en charge donnée dans la section 5.5.

Tout d'abord, examinons l'algorithme `eval_TR`. Pour $\tau_{k,i} \in \overline{\tau}$ telle que $k \leq p+1$, l'algorithme `eval_TR` teste les instances $\tau_{k',i'}$ des tâches dont l'indice est inférieur ou égal à p. Pour déterminer, si une telle instance est potentiellement active à un instant t, il évalue la condition :

$$r_{k',i'} \leq t < r_{k',i'} + C_{k'}$$

Ainsi, il teste en réalité le fait que l'instance $\tau_{k',i'}$ soit active. Le résultat fourni pour le temps de réponse de $\tau_{k,i}$ est donc exact. On obtient alors la proposition suivante.

Proposition 5.4 *Soit $s \in PFX$ une séquence d'exécution valide. Pour $\tau_{k,i} \in \overline{\tau}$ vérifiant $k \leq p+1$, on a :*

$$\texttt{eval\_TR}(\tau_{k,i}) = TR_s(\tau_{k,i})$$

Grâce à cette propriété, nous pouvons déterminer le temps de réponse exact des instances de τ_{p+1}. Considérons un instant $t \in \mathbb{N}$ à partir duquel la séquence s est cyclique. En calculant le temps de réponse des δ_{p+1} premières instances de τ_{p+1} activées après t, on peut déterminer s'il est toujours égal à C_k. Lorsque c'est vérifié, on peut généraliser ce résultat à toutes les instances de τ_{p+1} puisque le temps de réponse des instances ne peut pas diminuer (lemme 4.6) et que la séquence est cyclique au moins à partir de t.

L'approche que nous avons suivie permet de déterminer si le temps de réponse des instances de τ_{p+1} est toujours égal à C_{p+1}. Lorsque c'est le cas, on peut améliorer l'évaluation du temps de réponse fournie par `eval_TR`. Généralisons maintenant notre approche.

Supposons que le temps de réponse des instances $\tau_{k,i}$ des K tâches les plus prioritaires soit égal à C_k. On peut alors modifier l'algorithme `eval_TR` de la façon suivante :

- la condition $j \leq p$ à la ligne 7 est remplacée par $j \leq K$,
- la condition $j > p$ à la ligne 8 est remplacée par $j > K$.

Nous notons $\texttt{eval\_TRK}(K, \tau_{k,i})$ cette variante de l'algorithme $\texttt{eval\_TR}(\tau_{k,i})$. En raisonnant de la même manière avec τ_{K+1} qu'avec τ_{p+1}, on obtient la proposition suivante.

Proposition 5.5 *Soient $K \in \mathbb{N}$ et $s \in PFX$ une séquence d'exécution valide vérifiant la propriété suivante :*

$$\forall \tau_{k,i} \in \overline{\tau}, 1 \leq k \leq K \Rightarrow TR_s(\tau_{k,i}) = C_k$$

```
DMC_Dn2()
  M = 0
  for k from 1 to n do
    if k ≤ p
    then M = max(M,r_k − (T_k − C_k))
    else
      if r_k ≥ M
      then M = max(M,r_k-(T_k-eval_TR(τ_{k,δ_k−1})))
      else
        i = ⌈(M − r_k)/T_k⌉, tr = eval_TR(τ_{k,i−1+δ_k})
        if (tr > C_k) then M = max(M,r_{k,i}-(T_k-tr))
  return M
```

FIG. 5.4 – Approximation pseudo-polynomiale du calcul exact

Pour $\tau_{k,i} \in \overline{\tau}$ vérifiant $k \leq K+1$, on a :

$$\texttt{eval\_TRK}(K, \tau_{k,i}) = TR_s(\tau_{k,i})$$

Grâce à cette propriété et en continuant avec τ_{K+1} le raisonnement que l'on a eu précédemment avec τ_{p+1}, on obtient l'algorithme indiqué sur la figure 5.5 pour déterminer la longueur de la plus longue suite $\tau_1, \ldots, \tau_K$ telle que le temps de réponse des instances $\tau_{k,i}$ de ces tâches soit toujours égal à C_k. En combinant `lg_suite` avec `DMC_Dn2`, on obtient une majoration plus fine de la durée de montée en charge. L'algorithme indiqué sur la figure 5.6 implémente cette approche. La proposition suivante suit directement des propriétés de `lg_suite` et de `DMC_Dn2`.

Proposition 5.6 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\texttt{DMC\_Pn2}() \geq u_n$$

En début d'exécution, `lg_suite` appelle `DMC_n`, la complexité de cet appel est en $O(n)$. Ensuite, `lg_suite` effectue au plus $\sum_{k=p+1}^{n} \delta_k$ appels à `eval_TRK`. La complexité pour évaluer `lg_suite` est donc majorée par :

$$n + \sum_{k=p+1}^{n} (k-1).D_k.\delta_k \leq n + P. \sum_{k=p+1}^{n} (k-1) = n + P.\frac{n^2 - p^2 - (n-p)}{2}$$

```
lg_suite()
  k = p, deb = DMC_n()
  do
    stable = true, i = 0, dec = ⌈(deb − r_{k+1})/T_{k+1}⌉
    while (stable and (i < δ_{k+1})) do
      stable = (eval_TRK(k,k+1,i+dec) = C_k)
      i = i + 1
    if (stable) then k = k+1
  while (stable and (k < n))
  return k
```

FIG. 5.5 – Algorithme de recherche des tâches τ_k pour lesquelles le temps de réponse de chaque instance vaut C_k

```
DMC_Pn2()
  K = lg_suite()
  M = 0
  for k from 1 to n do
    if k ≤ K
    then M = max(M, r_k − (T_k − C_k))
    else
      if r_k ≥ M
      then M = max(M, r_k-(T_k-eval_TRK(K, τ_{k,δ_k−1})))
      else
        i = ⌈(M − r_k)/T_k⌉, tr = eval_TRK(K, τ_{k,i−1+δ_k})
        if (tr > C_k) then M = max(M, r_{k,i}-(T_k-tr))
  return M
```

FIG. 5.6 – Approximation exponentielle du calcul exact

Après l'appel à `lg_suite`, l'algorithme `DMC_Pn2` a la même complexité que `DMC_Dn2` puisque K peut être égal à p. Sa complexité est donc majorée par :

$$p + n + (D + P)\frac{n^2 - p^2 - (n - p)}{2}$$

Lorsque $n \leq p$, Le théorème 5.2 donne la durée exacte de montée en charge avec une complexité de calcul en $O(n)$. Dans le cas présent, on peut donc supposer $n \geq p$. L'ordre de grandeur de la complexité de calcul de `DMC_Pn2` est alors majoré par :

$$O(P.n^2)$$

5.7 Calcul de la durée exacte de montée en charge

Les majorations de la durée de montée en charge que nous avons données précédemment reposent sur une majoration du temps de réponse des instances. Dans cette section, nous utilisons un algorithme pour déterminer le temps de réponse exact d'une instance quelconque. Nous pouvons ensuite appliquer le théorème 5.4 sans le modifier pour déterminer la durée de montée en charge exacte.

Considérons une séquence $s \in PFX$ et une instance $\tau_{k,i} \in \overline{\tau}$. Par définition, l'instance $\tau_{k,i}$ est active aux seuls instants $t \in \mathbb{N}$ vérifiant :

$$r_{k,i} \leq t < e_s(\tau_{k,i}) = r_{k,i} + TR_s(\tau_{k,i})$$

Pour définir l'algorithme `eval_TR`, nous avons majoré le temps de réponse de $\tau_{k,i}$. Considérons maintenant l'algorithme indiqué sur la figure 5.7.

L'algorithme `exact_TR` est récurrent. Cependant, à chaque appel récursif, on a $j < k$. Une suite d'appels récursifs de `exact_TR` se termine donc toujours. De plus, l'algorithme `exact_TR` utilise la définition exacte d'une instance active, le temps de réponse qu'il calcule est donc exact. On obtient alors la proposition suivante.

Proposition 5.7 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\forall \tau_{k,i} \in \overline{\tau}, \texttt{exact\_TR}(\tau_{k,i}) = TR_s(\tau_{k,i})$$

Cette méthode de calcul des temps de réponse étant récursive, il est nécessaire pour l'utiliser de stocker en mémoire les temps de réponse précédemment calculés. Ainsi à chaque appel de `exact_TR`($\tau_{k,i}$), on vérifie si le temps de réponse de $\tau_{k,i}$ n'a pas déjà été calculé, dans le cas

```
exact_TR(τ_{k,i})
  nb_acces = 0, t = r_{k,i}
  while ((nb_acces < C_k) and (t < d_{k,i})) do
    many = 0, j = 1
    while ((many < p) and (j < k)) do
      if (t ≥ r_j) then
        if (j ≤ p) and (((t − r_j) mod T_j) < C_j) then many = many+1
        if (j > p) and (((t − r_j) mod T_j) < exact_TR(τ_{j,⌊(t−r_j)/T_j⌋})) then many = many+1
      j = j+1
    if (many < p) then nb_acces = nb_acces+1
    t = t+1
  return t - r_{k,i}
```

FIG. 5.7 – Calcul du temps de réponse exact d'une instance

contraire, on le calcule et on conserve le résultat dans une mémoire annexe. Le calcul de DMC_n par exemple permet de majorer la taille de la mémoire nécessaire. Cependant, sa complexité peut être exponentielle puisqu'elle dépend de la métapériode. Toutefois, dans la section 5.8, nous montrons qu'en pratique le nombre d'instances dont le temps de réponse est calculé reste polynomial.

Grâce à cette méthode de calcul du temps de réponse, on peut appliquer le théorème 5.4 sans aucune modification. Il n'est même plus nécessaire de rechercher la plus longue suite $\tau_1, \ldots, \tau_K$ telle que le temps de réponse des instances de ces tâches soit égal à C_k. On évite ainsi le calcul de lg_suite. On obtient alors l'algorithme indiqué sur la figure 5.8. Puisque l'algorithme exact_TR calcule le temps de réponse exact des instances, la proposition suivante est vérifiée.

Proposition 5.8 *Soit $s \in PFX$ une séquence d'exécution valide. On a :*

$$\texttt{DMC\_exact}() = u_n$$

La complexité de l'algorithme DMC_exact dépend du nombre I d'instances dont il est nécessaire de calculer le temps de réponse. Quelle que soit l'instance considérée, la complexité d'un appel à exact_TR est majorée par $n.D$. La complexité de l'algorithme DMC_exact est donc majorée par :

$$n + n.D.I$$

L'ordre de grandeur de la complexité de DMC_exact est alors majoré par :

$$O(n.D.I)$$

La variable tr1 utilisée dans l'algorithme DMC_exact correspond au temps de réponse d'instances dont la date d'activation est postérieure à P. Or, le nombre d'instances qui les précèdent est du même ordre de grandeur que $\sum_{k=1}^{n} \delta_k$. Ainsi, l'ordre de grandeur de I est exponentiel. La complexité de DMC_exact est donc majorée par :

$$O(n.D.P)$$

Cependant, les expérimentations que nous avons menées dans la section 5.8 révèlent qu'en pratique l'ordre de grandeur moyen de I est polynomial.

5.8 Expérimentations

5.8.1 Echantillons engendrés

Pour évaluer la qualité des majorations que nous avons obtenues dans les sections précédentes, nous avons réalisé 4 campagnes de tests. Chacune d'elles est consacrée à un nombre de processeurs

```
DMC_exact()
  M = 0
  for k from 1 to n do
    if k ≤ p
    then M = max(M,r_k − (T_k − C_k))
    else
      if r_k ≥ M
      then M = max(M,r_k-(T_k-exact_TR(τ_{k,δ_k−1})))
      else
        i = ⌈(M − r_k)/T_k⌉
        tr1 = exact_TR(τ_{k,i−1+δ_k})
        tr2 = exact_TR(τ_{k,i−1})
        if (tr1 ≠ tr2) then M = max(M,r_{k,i}-(T_k-tr1))
  return M
```

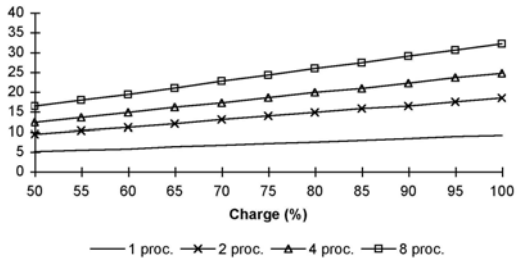
FIG. 5.8 – Algorithme de calcul de la durée exacte de montée en charge

précis, respectivement 1, 2, 4, et 8 processeurs. Pour chaque campagne de tests, nous avons généré aléatoirement plusieurs jeux de 1000 tests en fonction de la charge des systèmes de tâches. Pour la campagne à p processeurs, la charge varie dans l'ensemble $\{p \times 50\%, p \times 55\%, \ldots, p \times 100\%\}$. La figure 5.9 donne les caractéristiques moyennes de chaque jeu de test.

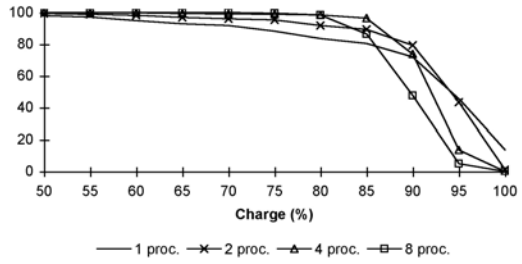
5.8.2 Méthode de simulation utilisée

Pour évaluer les résultats que nous avons obtenus dans les sections précédentes, nous devons choisir une politique d'ordonnancement à priorités fixes : nous avons utilisé DM. Nos résultats sont valides uniquement pour les séquences valides, nous ne pouvons donc utiliser que les systèmes de tâches générés qui sont ordonnançables par DM. La figure 5.9 indique la proportion de systèmes générés qui sont ordonnançables par DM. Pour les échantillons à 2, 4, et 8 processeurs, la proportion de systèmes valides est très faible lorsque la charge est à 100%. Les résultats que nous présentons ensuite étant des moyennes, ceux-ci ne sont donc pas représentatifs pour ces 3 jeux de test, en conséquence, nous ne les présentons pas. Ensuite, nous comparons la durée de montée en charge obtenue par simulation avec les majorations que nous avons établies.

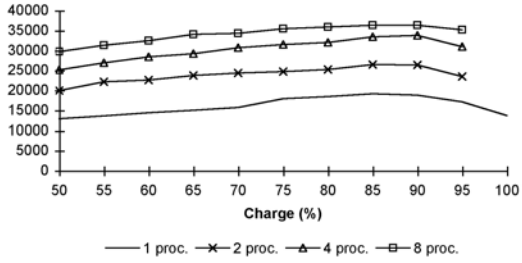
Pour déterminer la durée de montée en charge exacte à partir de la simulation, nous procédons de la manière suivante. Lors de la simulation, les tâches sont représentées à chaque instant par l'état courant de leur instance active. L'état d'une instance $\tau_{k,i}$ à un instant t est décrit par le nombre d'accès processeur qu'elle a déjà réalisés ($CE_s(\tau_{k,i}, t)$) et par la distance à son échéance ($d_{k,i} - t$). Nous simulons l'exécution tout en recherchant un instant $t \in r + P.\mathbb{N}^\star$ tel que l'état de chaque tâche soit identique à celui dans lequel elle était à l'instant $t - P$. Dès que l'on obtient un tel instant, on peut affirmer que la séquence s est périodique depuis l'instant $t - P$ (DM est déterministe). Toutefois, l'instant t ne correspond pas forcément à la durée exacte de montée en charge, car le début du régime cyclique peut se situer avant. Il est donc nécessaire de parcourir la séquence d'exécution de l'instant $t' = t - P - 1$ jusqu'à $t' = 0$ au pire. A chaque étape t' de ce parcours, si $s_{t'} \equiv_1 s_{t'+P}$ alors on passe à $t' - 1$, sinon l'instant exact d'entrée dans le régime cyclique est $t' + 1$.



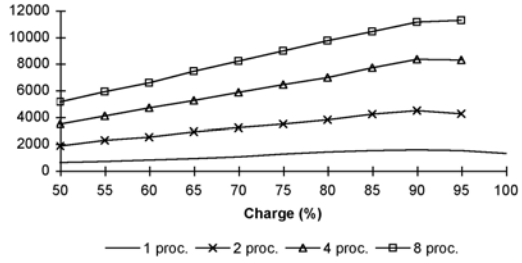
Nombre de tâches



Systèmes ordonnançables par DM



Valeur de la métapériode



Nombre d'instances générées lors de chaque métapériode

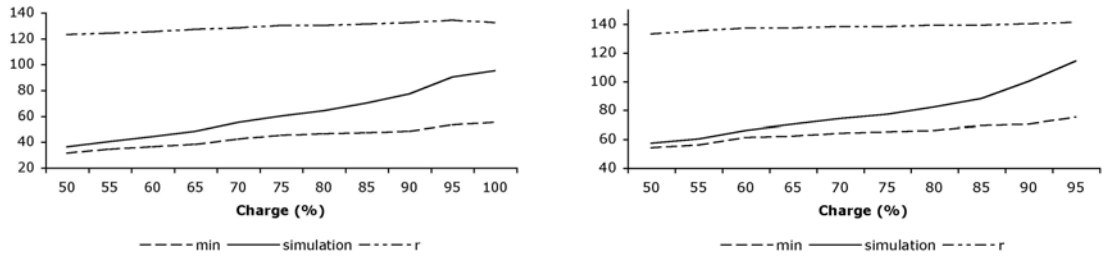
FIG. 5.9 – Caractéristiques des systèmes de tâches générés

5.8.3 Résultats

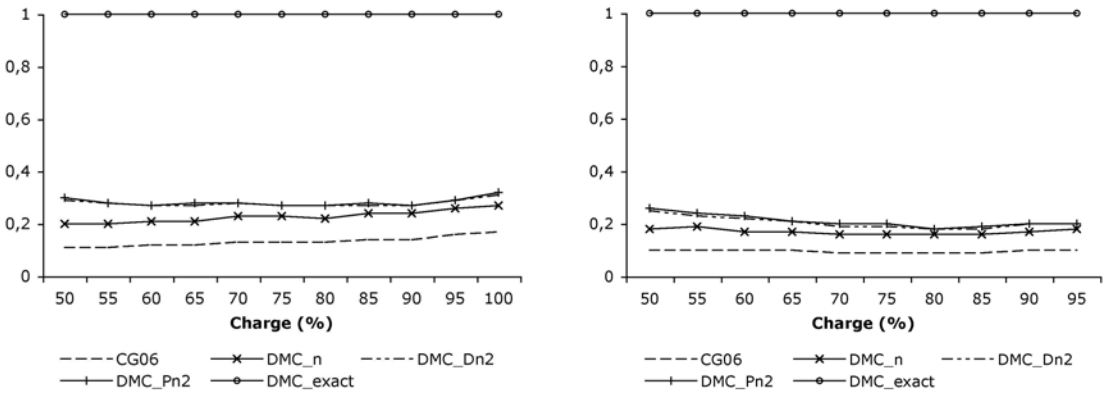
Les figures 5.10 et 5.11 présentent les résultats que nous avons obtenus. Ceux-ci sont relativement homogènes par rapport aux nombres de processeurs.

Tout d'abord, nous avons positionné la durée de montée en charge par rapport à la borne minimale fournie par le théorème 5.3, noté min, et la plus tardive date de première activation r. Quel que soit le nombre de processeurs, on constate que la durée obtenue par simulation est proche de la borne minimale lorsque la charge est faible et tend vers r lorsque la charge augmente. Cependant, r n'est pas un majorant de la durée de montée en charge, contrairement à ce que les résultats que nous avons obtenus pourraient laisser croire. Les résultats que nous présentons sont des moyennes, ainsi bien que r ne soit pas un majorant de la durée de montée en charge, en moyenne il constitue une assez bonne approximation lorsque la charge est élevée.

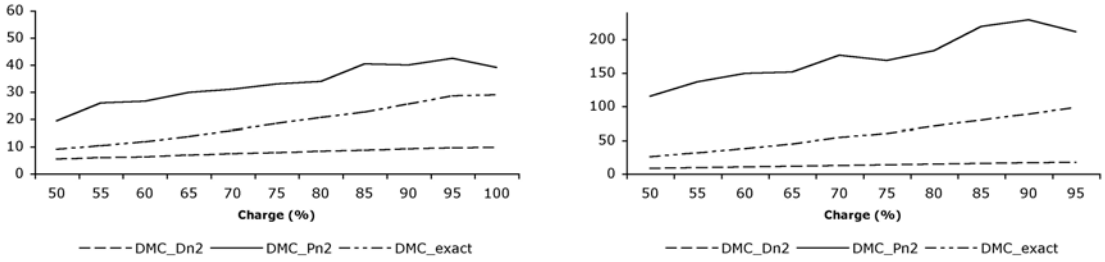
Monoprocesseur Biprocesseur



Durée de montée en charge réelle



Durée de montée en charge calculée



Nombre d'instances dont le temps de réponse est évalué

FIG. 5.10 – Résultats pour les systèmes monoprocesseur et biprocesseur

Ensuite, nous avons comparé les majorations que nous avons obtenues avec la durée de montée en charge obtenue par simulation. Pour cela, nous avons calculé le rapport $\frac{\text{durée fournie par la simulation}}{\text{durée fournie par la méthode x}}$. Ainsi, ce rapport tend vers 1 lorsque l'approximation est bonne, et tend vers 0 dans le cas contraire. Remarquons tout de suite que ce rapport vaut toujours 1 avec la méthode `DMC_exact`, ceci confirme nos résultats puisque les durées obtenues théoriquement et par simulation sont identiques.

Pour les tests en monoprocesseur et biprocesseur, le majorant fourni par `CG06a` est environ 10 fois plus grand que la durée de montée en charge. Le meilleur majorant obtenu est entre 3 et 5 fois plus grand. Les méthodes `DMC_Dn2` et `DMC_Pn2` ne se distinguent presque pas et apportent peu d'amélioration par rapport à `DMC_n`.

Pour les tests avec 4 ou 8 processeurs, les performances des méthodes `CG06a` et `DMC_n` se

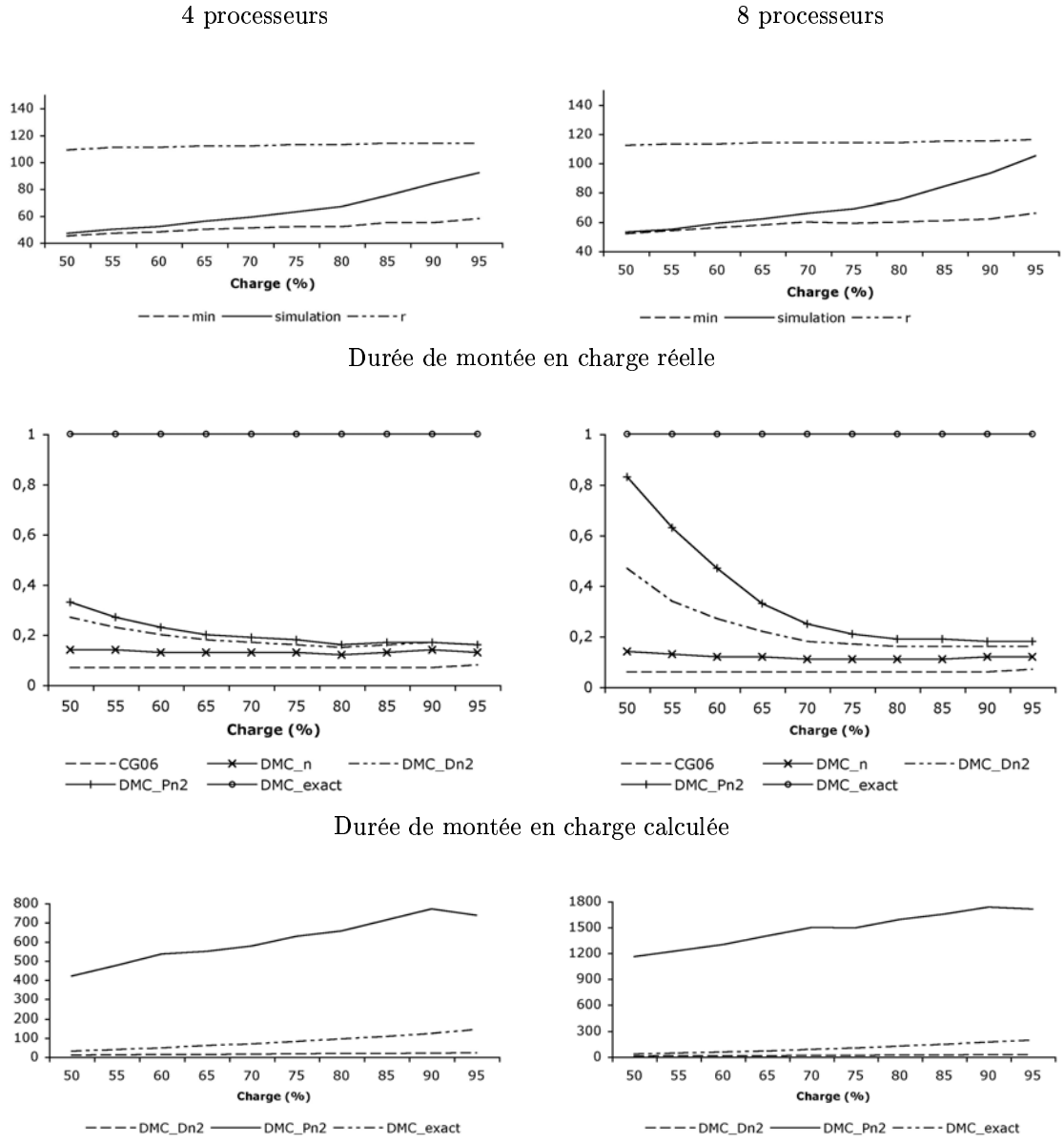


FIG. 5.11 – Résultats pour les systèmes à 4 ou 8 processeurs

dégradent par rapport au cas avec 1 ou 2 processeurs. Par contre, les méthodes `DMC_Dn2` et `DMC_Pn2` se distinguent clairement et augmentent nettement leurs performances lorsque la charge est faible. Dans certains cas, elles fournissent de très bons majorants de la durée de montée en charge.

Ainsi, la méthode `DMC_n` fournit généralement une majoration 2 fois plus petite que celle fournie par `CG06a`. Les méthodes `DMC_Dn2` et `DMC_Pn2` apportent un gain supplémentaire qui s'amplifie lorsque la charge n'est pas trop grande ($<70\%$) et que le nombre de processeurs est assez grand (≥ 4).

Nous avons évalué le nombre I d'instances dont le temps de réponse est calculé en fonction des différentes méthodes. Pour la méthode `DMC_Dn2`, I est inférieur au nombre de tâches n. Par contre, pour `DMC_Pn2`, I est important mais reste bien inférieur au nombre d'instances générées par métapériode (voir figure 5.9). Rappelons que la méthode `DMC_Pn2` peut potentiellement calculer plus de $\sum_{k=1}^{n} \delta_k$ temps de réponse. Finalement, pour la méthode `DMC_exact`, I reste faible et paraît être du même ordre de grandeur que n. En admettant ce point, la méthode `DMC_exact` fournirait alors la durée exacte de montée en charge par un calcul dont la complexité est du même ordre de grandeur que $O(D.n^2)$. En moyenne, sa complexité serait alors du même ordre de grandeur que `DMC_Dn2`. Rappelons que comme pour `DMC_Pn2`, le nombre d'instances dont `DMC_exact` calcule le temps de réponse peut être exponentiel - i.e. du même ordre que $\sum_{k=1}^{n} \delta_k$.

La figure 5.12 présente le temps de calcul nécessaire pour traiter avec chaque méthode l'ensemble de l'échantillon construit pour 8 processeurs. Les méthodes `DMC_Dn2` et `DMC_exact` requièrent des temps de calcul relativement similaires, ce point corrobore le fait que la complexité en moyenne de `DMC_exact` puisse être de l'ordre de grandeur de $O(D.n^2)$. Par contre, on remarque que la méthode `DMC_Pn2` nécessite un temps de calcul bien plus important mais qui reste très inférieur à celui de la simulation.

5.9 Conclusion

Dans ce chapitre, nous avons déterminé la durée exacte de montée en charge. Nous en avons proposé plusieurs majorations basées sur l'évaluation du pire temps de réponse des instances ; d'autres techniques de majoration du temps de réponse peuvent aussi être utilisées.

La suite logique de notre travail consiste à déterminer un intervalle de faisabilité qui soit le plus petit possible. Les propriétés mises en évidence dans [24, 42] doivent alors être croisées avec l'expression de la durée de montée en charge que nous avons donnée.

Plusieurs extensions sont envisageables. Premièrement, les configurations de priorité $\overline{\overline{\text{PFI}}}$ sont équivalentes aux configurations PFX à une transformation près du système de tâches (voir section 8.4.1). Ainsi, on peut appliquer les résultats que nous avons obtenus au contexte $\overline{\overline{\text{PFI}}}$. Toutefois, la durée de montée en charge du système équivalent (PFX) est une majoration de celle du système initial ($\overline{\overline{\text{PFI}}}$). Une étude supplémentaire est donc nécessaire pour étendre correctement à $\overline{\overline{\text{PFI}}}$ les résultats que nous avons obtenus pour PFX. Deuxièmement, nous avons mené notre étude en supposant que l'exécutif temps-réel est préemptif et que la migration totale est autorisée. On peut

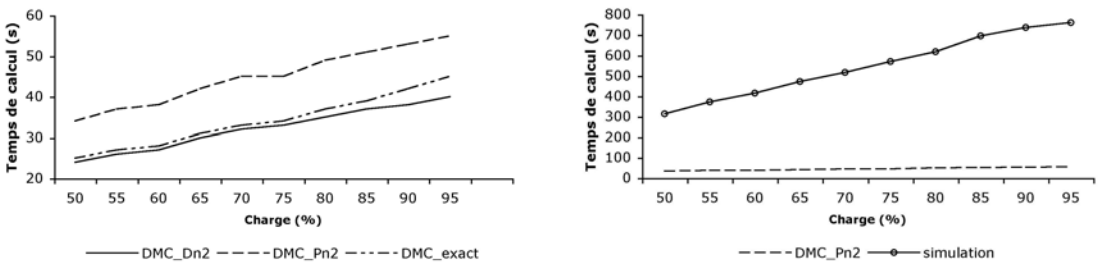


FIG. 5.12 – Temps de calcul de l'échantillon à 8 processeurs, 1000 systèmes de tâches à évaluer pour chaque valeur de la charge

naturellement chercher à étendre nos résultats à d'autres exécutifs temps-réel. Troisièmement, nous avons considéré que les tâches sont indépendantes, on peut aussi étudier l'impact des interactions entre les tâches sur la durée de montée en charge.

Les travaux menés dans ce chapitre ont fait l'objet d'un article soumis à une revue internationale en 2007.

Chapitre 6

Intervalle d'étude pour les méthodes hors-ligne

6.1 Introduction

Dans le chapitre 3, nous avons donné des exemples de séquences d'exécution valides et acycliques appartenant à PFI. Ainsi, il n'existe pas d'intervalle de faisabilité commun à toutes les séquences d'exécution puisque le premier dépassement d'échéances peut être commis arbitrairement loin. D'autre part, les approches hors-ligne exactes consistent généralement à explorer l'ensemble des séquences d'exécution possibles, elles considèrent donc aussi les séquences acycliques. Ainsi, en toute généralité, on ne dispose pas d'intervalle d'étude pour les méthodes hors-ligne.

Dans [21], les auteurs montrent que les séquences d'exécution monoprocesseur engendrées par les politiques appartenant à DCL sont cycliques. De plus, ils fournissent une méthode pour calculer la durée de montée en charge à partir du diagramme de charge. Cette durée est indépendante de la politique DCL utilisée. La méthode hors-ligne proposée par [45] réutilise ces résultats pour définir l'intervalle d'étude. Cependant, les méthodes hors-ligne considèrent aussi les séquences d'exécution acycliques puisqu'elles effectuent une recherche exhaustive. La durée d'étude utilisée par [45] restreint donc la recherche aux séquences dont le régime cyclique commence avant : certaines séquences d'exécution ne seront jamais considérées, la recherche effectuée n'est donc plus exhaustive. Toutefois en monoprocesseur, la classe de politiques DCL est optimale. Donc, s'il existe une séquence valide, alors la méthode de [45] fournit tout de même une solution.

En contexte multiprocesseur, on connaît une borne de la durée de montée en charge pour les séquences engendrées par les politiques appartenant à PFX ou à $\overline{\text{PFI}}$ (voir chapitres 4 et 5). Pour la politique LLF et plus généralement pour les séquences DCL, on ne dispose pas d'une telle borne. Ainsi, la méthode utilisée par [45] en monoprocesseur ne peut pas être reprise pour le contexte multiprocesseur. Dans ce chapitre, nous proposons une nouvelle approche pour déterminer un intervalle d'étude pour les méthodes hors-ligne en multiprocesseur. Elle ne repose pas sur la cyclicité des séquences d'exécution, mais sur les états atteignables par le système de tâches à un instant donné (par état, nous considérons le nombre d'accès processeur réalisés par chaque instance courante).

6.2 Contexte d'étude

Dans cette section, nous posons le formalisme propre à notre étude. Nous considérons les systèmes de tâches périodiques interdépendantes exécutées en contexte multiprocesseur préemptif avec migration totale. Nous supposons que les échéances des tâches sont inférieures ou égales à leur période. De plus, nous ne considérons que les méthodes hors-ligne qui explorent *toutes* les séquences d'exécution. Par exemple, la méthode que nous avons définie dans le chapitre 7 traite

uniquement les séquences à priorités fixes, elle n'entre donc pas dans ce cadre-là. Par contre, la méthode basée sur les réseaux de Petri définie dans [45] et celle que nous présentons dans la partie IV vérifient bien cette hypothèse.

Nous modélisons les séquences d'exécution à l'aide de l'état d'avancement de l'instance courante de chaque tâche.

Définition 6.1 *Soient s une séquence d'exécution, $t \in \mathbb{N}$, et $\tau_k \in \tau$. On note $\epsilon_k(s,t)$ l'état de l'instance courante de τ_k à l'instant t dans la séquence s :*

$$\epsilon_k(s,t) = \begin{cases} 0, \text{ si } r_k < t \\ CE_s(\tau_{k,\lfloor (t-r_k)/T_k \rfloor}, t), \text{ sinon} \end{cases}$$

On note $\epsilon(s,t) = (\epsilon_1(s,t), \ldots, \epsilon_n(s,t))$ l'état des tâches à l'instant t dans la séquence s, et E l'ensemble des états possibles.

Nous modélisons les contraintes induites par les interactions entre les tâches en utilisant des prédicats exprimés à partir du modèle que nous avons choisi. Une contrainte est dite périodique si son expression ne dépend pas de la métapériode courante.

Définition 6.2 *Soit s une séquence d'exécution. Une contrainte d'exécution Q sur s est un prédicat de la forme $Q(t, \epsilon(s,t), s_t)$. La séquence s est valide selon Q jusqu'à l'instant t si et seulement si :*

$$\forall t' < t, Q(t', \epsilon(s,t'), s_{t'})$$

Une contrainte d'exécution Q est cyclique de période P à partir d'un instant $\delta \in \mathbb{N}$ si et seulement si :

$$\forall t \geq \delta, \forall e \in E, \forall x \subset \overline{\tau}, Q(t,e,x) \Leftrightarrow Q(t+P, e, \{\tau_{k,i} | \tau_{k,i-\delta_k} \in x\})$$

Dans toute la suite, on note Q la contrainte d'exécution appliquée au système de tâches τ, et V_t l'ensemble des séquences d'exécution valides selon Q jusqu'à l'instant t. Dans la section 6.3.2, nous précisons les contraintes d'exécution que l'on peut modéliser à partir du prédicat Q.

Nous disons qu'un état est atteignable à l'instant t s'il existe une séquence d'exécution valide jusqu'à l'instant t qui amène le système de tâches dans cet état-là. Cette notion permet de définir une fonction d'atteignabilité : un état est atteignable à partir d'un autre s'il existe une séquence d'exécution valide passant par ces deux états-là. Nous adoptons alors les définitions suivantes.

Définition 6.3 *Pour tout $t \in \mathbb{N}$, on note A_t l'ensemble des états atteignables à l'instant t :*

$$A_t = \{e | \exists s \in V_t \text{ tel que } e = \epsilon(s,t)\}$$

Définition 6.4 *Pour tout instant $t \in \mathbb{N}$, tout $\Delta \in \mathbb{N}$, et tout état $e \in A_t$, on note $att_{t,\Delta}$ la fonction qui associe à e l'ensemble des états atteignables à partir de e à l'instant t en Δ étapes :*

$$att_{t,\Delta}(e) = \{e' | \exists s \in V_{t+\Delta} \text{ tel que } \epsilon(s,t) = e \wedge \epsilon(s, t+\Delta) = e'\}$$

Lorsque $\Delta = 1$, on note plus simplement att_t pour $att_{t,\Delta}$.

6.3 Intervalle d'étude

A partir du modèle défini dans la section précédente, nous montrons que l'on peut mener une analyse finie même dans le cas où les tâches sont à départs différés et bien que certaines séquences d'exécution valides puissent être acycliques. Dans la section suivante, nous appliquons ce résultat à des méthodes de validation hors-ligne connues.

6.3.1 Etude dans le contexte des contraintes périodiques

Dans cette section, nous étudions les ensembles $(A_t)_{t\in\mathbb{N}}$ obtenus lorsqu'il existe un instant $\delta \in \mathbb{N}$ satisfaisant les deux hypothèses suivantes :

- les contraintes sont cycliques de période P à partir de δ
- $A_{\delta+P} \subset A_\delta$

En présence de ces deux propriétés, on suit l'approche suivante pour obtenir une durée de validation finie pour l'étude de toutes les séquences valides. La périodicité des contraintes entraîne la périodicité de la fonction d'atteignabilité (lemme 6.1). Conjuguée avec l'inclusion $A_{\delta+P} \subset A_\delta$, cette propriété induit la "décroissance" des ensembles A_t (théorème 6.1) :

$$A_\delta \supset A_{\delta+P} \supset A_{\delta+2P} \supset \cdots$$

Le nombre d'états possibles étant fini, on montre alors (théorème 6.2) qu'il existe $\eta \in \delta + P.\mathbb{N}$ vérifiant :

$$A_\delta \supset A_{\delta+P} \supset A_{\delta+2P} \supset \cdots \supset A_\eta = A_{\eta+P} = \cdots$$

La suite des ensembles $(A_t)_{t\in\mathbb{N}}$ admet donc une limite. Si c'est l'ensemble vide, alors il n'existe aucune séquence d'exécution de τ qui soit valide ; dans le cas contraire, les ensembles $A_\eta, \ldots, A_{\eta+P-1}$ décrivent les états qui appartiennent au régime cyclique du système de tâches.

Plusieurs méthodes ont été développées pour déterminer les ensembles A_t, par exemple [26, 45, 62, 64]. Le raisonnement mené ci-dessus montre que l'on peut stopper l'évaluation dès que l'on obtient deux ensembles distants de P unités de temps et contenant exactement les mêmes éléments. Cependant, pour déterminer si un ensemble A_t contient les mêmes éléments que l'ensemble A_{t+P}, on est amené à utiliser un algorithme dont la complexité de calcul est majorée par $|A_t| \times |A_{t+P}|$. Evaluer cette condition à chaque instant engendre donc une complexité importante. Toutefois, le théorème 6.2 montre que l'ensemble A_{t+P} est inclus dans l'ensemble A_t, on peut donc tester $A_t = A_{t+P}$ en évaluant simplement $|A_t| = |A_{t+P}|$: la complexité nécessaire descend alors à $\max\{|A_t|, |A_{t+P}|\}$. Ce critère de détection est l'objet du théorème 6.3.

Tout d'abord, nous montrons que dès que la contrainte d'exécution Q considérée est périodique, alors la fonction d'atteignabilité att_t l'est aussi.

Lemme 6.1 *Soient $\delta \in \mathbb{N}$, $t \geq \delta$ et $e \in A_t \cap A_{t+P}$. Si la contrainte Q est périodique à partir de δ, alors on a pour tout état $e' \in E$ et tout $\Delta \in \mathbb{N}$:*

$$e' \in att_{t,\Delta}(e) \Leftrightarrow e' \in att_{t+P,\Delta}(e)$$

Démonstration :

Soient $\Delta \in \mathbb{N}$ et $e' \in att_{t,\Delta}(e)$.

Puisque $e \in A_t$ et que $e' \in att_{t,\Delta}(e)$, il existe une séquence $s \in V_{t+\Delta}$ vérifiant $\epsilon(s,t) = e$ et $\epsilon(s, t+\Delta) = e'$.

D'autre part, e appartient aussi à A_{t+P}. Donc, il existe une séquence $s' \in V_{t+P}$ vérifiant $\epsilon(s', t+P) = e$.

Définissons maintenant la séquence s'' de la manière suivante :

$$\forall t' \in \mathbb{N}, s''_{t'} = \begin{cases} s'_{t'}, \text{ si } t' < t+P \\ s_{t'-P}, \text{ sinon} \end{cases}$$

Par construction, on a $s'' \in V_{t+P}$, ainsi que $\epsilon(s'', t+P) = e$ et $\epsilon(s'', t+P+\Delta) = e'$. Il reste maintenant à montrer que $s'' \in V_{t+P+\Delta}$.

On a $s \in V_{t+\Delta}$, et donc :

$$\forall t' \in \mathbb{N}, t \leq t' < t+\Delta \Rightarrow Q(t', \epsilon(s,t'), s_{t'})$$

Par construction, la portion de la séquence s comprise dans l'intervalle $[t, t+\Delta]$ est identique à la portion de s'' comprise dans l'intervalle $[t+P, t+P+\Delta]$. Comme la contrainte Q est périodique, on obtient :

$$\forall t' \in \mathbb{N}, t+P \leq t' < t+P+\Delta \Rightarrow Q(t', \epsilon(s'',t'), s''_{t'})$$

On a donc $s'' \in V_{t+P+\Delta}$, et finalement $e' \in att_{t+P,\Delta}(e)$.

Supposons maintenant que $e' \in att_{t+P,\Delta}(e)$. En raisonnant de la même manière que précédemment, on montre que $e' \in att_{t,\Delta}(e)$.

CQFD.

Nous montrons maintenant que les ensembles A_t sont décroissants dès que la contrainte Q considérée est périodique et qu'il existe $t \in \mathbb{N}$ vérifiant $A_{t+P} \subset A_t$.

Théorème 6.1 *Soit $\delta \in \mathbb{N}$. Si la contrainte Q est périodique à partir de δ et si $A_{\delta+P} \subset A_\delta$, alors on a :*

$$\forall k \in \mathbb{N}, A_{\delta+P+k.P} \subset A_{\delta+k.P}$$

Démonstration :

Par hypothèse, on a :

$$\forall e \in A_{\delta+P}, e \in A_\delta \cap A_{\delta+P}$$

D'après le lemme 6.1, on obtient alors :

$$\forall e \in A_{\delta+P}, \forall e' \in E, e' \in att_{\delta+P,P}(e) \Leftrightarrow e' \in att_{\delta,P}(e)$$

On obtient donc :

$$att_{\delta+P,P}(A_{\delta+P}) = att_{\delta,P}(A_{\delta+P})$$

Or par hypothèse, on a $A_{\delta+P} \subset A_\delta$, on en déduit donc :

$$A_{\delta+2P} = att_{\delta+P,P}(A_{\delta+P}) \subset att_{\delta,P}(A_\delta) = A_{\delta+P}$$

En réitérant ce raisonnement, on obtient le résultat.

CQFD.

Nous déduisons du résultat précédent que la suite $(A_t)_{t \in \delta+P.\mathbb{N}}$ admet une limite.

Théorème 6.2 *Soit $\delta \in \mathbb{N}$. Si la contrainte Q est périodique à partir de δ et si $A_{\delta+P} \subset A_\delta$, alors il existe $\eta \in \delta + P.\mathbb{N}$ vérifiant :*

$$A_\eta = A_{\eta+P} = A_{\eta+2P} = \cdots$$

Démonstration :

D'après les théorèmes 6.4 et 6.2, on a :

$$A_\delta \supset A_{\delta+P} \supset A_{\delta+2P} \supset \cdots$$

Ainsi, le nombre d'éléments contenus dans les ensembles $A_\delta, A_{\delta+P}, \ldots$ est décroissant. De plus, le nombre d'éléments contenus dans ces ensembles est toujours fini. Il existe donc $\eta \in \delta + P.\mathbb{N}$ vérifiant :

$$A_\eta = A_{\eta+P}$$

En utilisant le lemme 6.1, on montre alors :

$$att_{\eta,P}(A_\eta) = att_{\eta+P,P}(A_{\eta+P})$$

Et on en déduit :

$$A_{\eta+P} = A_{\eta+2P}$$

De proche en proche, on obtient le résultat.

CQFD.

Finalement, nous donnons un critère pour déterminer l'instant auquel la suite $(A_t)_{t \in \delta+P.\mathbb{N}}$ atteint sa limite.

Théorème 6.3 *Soient $\delta \in \mathbb{N}$. Si la contrainte Q est périodique à partir de δ et si $A_{\delta+P} \subset A_\delta$, alors pour tout $\eta \in \delta + P.\mathbb{N}$ vérifiant $|A_\eta| = |A_{\eta+P}|$. On a :*

$$A_\eta = A_{\eta+P} = A_{\eta+2P} = \cdots$$

Démonstration :
D'après les théorèmes 6.4 et 6.2, on a :

$$A_\eta \supset A_{\eta+P} \supset A_{\eta+2P} \supset \cdots$$

Etant donné que $|A_\eta| = |A_{\eta+P}|$, on obtient aussi :

$$A_\eta = A_{\eta+P}$$

En utilisant le lemme 6.1, on montre alors :

$$A_{\eta+P} = A_{\eta+2P}$$

De proche en proche, on obtient alors le résultat.
CQFD.

6.3.2 Intégration des contraintes d'exécution

Les contraintes d'exécution permettent de représenter les interactions entre les tâches et aussi celles dues au partage des processeurs. En utilisant les notations que nous avons définies sur les séquences d'exécution dans la section 2.2, on peut exprimer facilement les contraintes classiques (précédence, exclusion mutuelle, etc). Toutefois, les notions sous-jacentes à ces notations supposent une connaissance globale de la séquence d'exécution. Or, le formalisme que nous utilisons ici ne fournit qu'une représentation partielle de la séquence : on doit donc vérifier les contraintes s'appliquant à un instant donné seulement à l'aide de la connaissance de l'état des instances courantes et de la combinaison d'instances choisie pour être exécutées à cet instant-là. Les contraintes d'exécution doivent donc être formulées seulement à partir de la notion d'état que nous avons définie dans la section 6.2.

Echéances temporelles. Dans un système temps-réel, chaque tâche est soumise à un délai critique : toutes les instances engendrées doivent terminer leur exécution sans dépasser leur échéance. Cette contrainte s'exprime de la manière suivante :

$$\forall \tau_{k,i} \in \overline{\tau}, TR_s(\tau_{k,i}) \leq D_k$$

Pour intégrer cette contrainte, on vérifie pour un instant donné que toutes les instances dont l'échéance se situe à cet instant-là ont bien terminé leur exécution. On obtient alors le prédicat suivant :

$$Q_{re}(t,e,x) \Leftrightarrow \left[\forall \tau_k \in \tau, t \geq r_k \wedge (t-r_k) \bmod T_k = D_k - 1 \Rightarrow e_k + |x \cap \{\tau_{k,\lfloor (t-r_k)/T_k \rfloor}\}| = C_k\right]$$

On peut aussi utiliser un prédicat plus exigeant pour représenter le respect des échéances basé sur la laxité des tâches. On impose alors que la laxité des tâches soit toujours positive ou nulle. Ainsi, on peut détecter les dépassements d'échéance plus tôt. On formule cette seconde version par le prédicat suivant :

$$Q_{re'}(t,e,x) \Leftrightarrow \left[\begin{array}{l} \forall \tau_k \in \tau, t \geq r_k \wedge r_{k,i} \leq t < d_{k,i} \Rightarrow d_{k,i} - t - 1 \geq C_k - e_k - |x \cap \{\tau_{k,i}\}| \\ \text{où } i = \lfloor (t-r_k)/T_k \rfloor \end{array}\right]$$

On vérifie aisément que ces deux contraintes sont périodiques.

Proposition 6.1 *Les contraintes Q_{re} et $Q_{re'}$ sont cycliques de période P à partir de l'instant r.*

Partage des processeurs. Nous considérons les architectures multiprocesseurs et nous supposons que les tâches sont préemptibles et que l'hypothèse de migration globale est vérifiée. Dans ce contexte, la seule contrainte imposée par l'architecture consiste à vérifier que le nombre de tâches exécutées simultanément n'excède jamais le nombre de processeurs :

$$\forall t \in \mathbb{N}, |s_t| \leq p$$

Naturellement, le respect du nombre de processeurs s'exprime alors ainsi :

$$Q_{np}(t, e, x) \Leftrightarrow |x| \leq p$$

Cette contrainte est clairement périodique.

Proposition 6.2 *La contrainte Q_{np} est cyclique de période P à partir de l'instant r.*

Relation de précédence. Nous avons vu dans le chapitre 2 que les communications et les synchronisations entre les tâches sont représentées par des relations de précédence. Nous ne considérons ici que les contraintes de précédence simple : deux tâches liées par une relation de précédence doivent avoir la même période. Ainsi, lorsque τ_k précède $\tau_{k'}$, chaque instance $\tau_{k',i}$ ne peut être exécutée qu'une fois l'instance $\tau_{k,i}$ terminée. On formule le respect des contraintes de précédence ainsi :

$$\forall \tau_{k,i}, \tau_{k',i} \in \overline{\tau}, \tau_k \succ \tau_{k'} \Rightarrow CE_s(\tau_{k',i}, e_s(\tau_{k,i})) = 0$$

Considérons deux tâches liées par une relation de précédence $\tau_k \succ \tau_{k'}$. Dès qu'une instance $\tau_{k',i}$ s'exécute à un instant t donné, l'instance $\tau_{k,i}$ doit être terminée. Pour cela, on doit forcément avoir $t \geq r_{k,\lfloor (t-r_{k'})/T_{k'} \rfloor}$. Posons $i = \lfloor (t - r_{k'})/T_{k'} \rfloor$. Deux cas sont à prendre en considération en plus de la condition $t \geq r_{k,i}$:

- soit $e_k = C_k$: l'instance $\tau_{k,i}$ est terminée,
- soit $t \geq r_{k,i+1}$: l'instance $\tau_{k,i+1}$ est activée, puisque la séquence est valide et que $D_k \leq T_k$, alors l'instance $\tau_{k,i}$ est bien finie.

On peut donc formuler la contrainte des relations de précédence ainsi :

$$Q_{rp}(t, e, x) \Leftrightarrow \left[\begin{array}{l} \forall \tau_k, \tau_{k'} \in \tau, \tau_k \succ \tau_{k'} \wedge e_{k'} > 0 \Rightarrow t \geq r_{k,i} \wedge (e_k = C_k \vee t \geq r_{k,i+1}) \\ \text{où } i = \lfloor (t - r_{k'})/T_{k'} \rfloor \end{array} \right]$$

Le comportement de ce prédicat est indépendant de la métapériode courante, le résultat suivant est donc vérifié.

Proposition 6.3 *La contrainte Q_{rp} est cyclique de période P à partir de l'instant r.*

Exclusion mutuelle. Pour modéliser ce type de contraintes, on utilise généralement des sémaphores. Pour ne pas alourdir le formalisme, nous adoptons les deux restrictions suivantes :

- le système de tâches est doté d'au plus un sémaphore
- chaque instance réalise au plus un accès au sémaphore

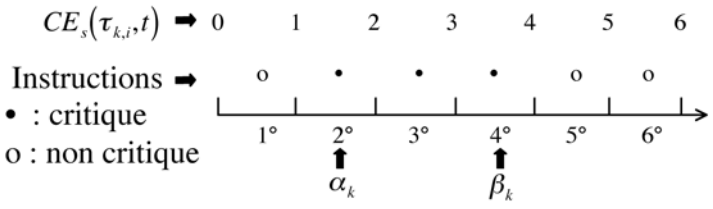


FIG. 6.1 – Accès au sémaphore pour la tâche $\tau_k \in \tau$

L'étude correspondant à ces deux restrictions s'étend mutatis-mutandis au cas général.

Considérons une tâche $\tau_k \in \tau$, nous modélisons l'accès qu'elle peut réaliser au sémaphore à l'aide des deux informations suivantes (voir figure 6.1) :

- α_k : la première instruction appartenant à la section critique,
- β_k : la dernière instruction appartenant à la section critique.

Lorsqu'une tâche $\tau_k \in \tau$ n'accède jamais au sémaphore, nous posons par convention $\alpha_k = \beta_k = -1$.

Dans ce cadre, une instance accède au sémaphore à un instant t si et seulement si :

$$\alpha_k \leq CE_s(\tau_{k,i}, t) < \beta_k \wedge \alpha_k < CE_s(\tau_{k,i}, t+1) \leq \beta_k$$

Dans toute la suite, on note $sem_s(\tau_{k,i}, t)$ le prédicat défini par cette condition. L'exclusion mutuelle est satisfaite si et seulement si à chaque instant le nombre d'instances en section critique est inférieur ou égal à 1 :

$$\forall t \in \mathbb{N}, |\{\tau_{k,i} \in \overline{\tau} | sem_s(\tau_{k,i}, t)\}| \leq 1$$

Pour intégrer la contrainte d'exclusion mutuelle des accès aux sémaphores, on utilise l'expression donnée ci-dessus en apportant quelques modifications :

$$Q_{em}(t, e, x) \Leftrightarrow \left[\begin{array}{l} |\{\tau_k \in \tau | \alpha_k \leq e_k < \beta_k \wedge \alpha_k < e_k + |x \cap \{\tau_{k,i}\}| \leq \beta_k\}| \leq 1 \\ \text{où } i = \lfloor (t - r_k)/T_k \rfloor \end{array} \right]$$

Ce prédicat ne dépendant pas de la métapériode considérée, le résultat suivant est donc vérifié.

Proposition 6.4 *La contrainte Q_{em} est cyclique de période P à partir de l'instant r.*

6.3.3 Décroissance de la suite $(A_t)_{t \in r+P.\mathbb{N}}$

L'étude que nous avons menée dans la section 6.3.1 repose sur deux hypothèses. Dans la section précédente, nous avons montré qu'un certain nombre de contraintes classiques sont exprimables sous la forme de contraintes d'exécution (prédicat) périodiques. Dans cette section, nous montrons que les systèmes de tâches soumis à ces seules contraintes vérifient $A_{r+P} \subset A_r$. Ainsi, l'analyse menée dans la section 6.3.1 s'applique à ces systèmes de tâches.

Théorème 6.4 *Si le système de tâches τ est soumis aux seules contraintes $Q_{re}, Q_{np}, Q_{rp}, Q_{em}$, alors on a :*

$$A_{r+P} \subset A_r$$

Démonstration :

Soit $e \in A_{r+P}$. Par définition, il existe une séquence $s \in V_{r+P}$ vérifiant $\epsilon(s, r+P) = e$.

Définissons maintenant une séquence d'exécution s' de la manière suivante :

$$\forall t \in \mathbb{N}, s'_t = \{\tau_{k,i-\delta_k} | \tau_{k,i} \in s_{t+P} \wedge i \geq \delta_k\}$$

A chaque instant t, la séquence s' ainsi définie correspond à la séquence s, décalée de P unités de temps vers la gauche, et restreinte aux instances $\tau_{k,i}$ dont la date d'activation est postérieure ou égale à $r_k + P$.

Par construction, on a $\epsilon(s', r) = e$. Ainsi, la séquence s' permet d'atteindre l'état e en r étapes. Montrons maintenant que la séquence s' appartient à V_r.

Par définition, la séquence s est valide, on a donc $\forall t \in \mathbb{N}, |s_t| \leq p$. On obtient alors :

$$\forall t \in \mathbb{N}, |s'_t| \leq |s_{t+P}| \leq p$$

Donc, la séquence s' respecte le nombre de processeurs disponibles.

Considérons une instance $\tau_{k,i} \in \overline{\tau}$ dont l'échéance appartient à l'intervalle $[0, r[$. Par construction, pour $t \geq r_k$, on a $\epsilon_k(s', t) = \epsilon_k(s, t+P)$. Or, pour la séquence s, toutes les échéances des tâches appartenant à l'intervalle $[0, r+P[$ sont respectées. Ainsi, l'instance $\tau_{k,i}$ respecte son échéance dans la séquence s'.

Par construction, chaque instance exécutée dans la séquence s' correspond à une instance exécutée dans la séquence s exactement P unités de temps après. On en déduit :

$$\forall t \in \mathbb{N}, |\{\tau_{k,i} | sem_{s'}(\tau_{k,i}, t)\}| \leq |\{\tau_{k,i} | sem_s(\tau_{k,i}, t+P)\}| \leq 1$$

La séquence s' respecte donc l'exclusion mutuelle des accès aux sémaphores.

Considérons maintenant deux instances liées par une relation de précédence $\tau_{k,i} \succ \tau_{k',i}$. Ces deux instances correspondent respectivement dans la séquence s' aux instances $\tau_{k,i+\delta_k}$ et $\tau_{k',i+\delta_{k'}}$ de la séquence s. Or, la séquence s est valide jusqu'à l'instant $r+P$, ainsi pour tout $t \leq \max\{r+P, e_s(\tau_{k,i+\delta_k})\}$, on a :

$$CE_s(\tau_{k',i+\delta_{k'}}, t) = 0$$

Par construction, on en déduit pour tout $t \leq \max\{r, e_{s'}(\tau_{k,i})\}$:

$$CE_{s'}(\tau_{k',i}, t) = 0$$

La séquence s' respecte donc les relations de précédence.

Ainsi, la séquence s' est valide jusqu'à l'instant r, on a donc $e \in A_r$.

CQFD.

6.4 Expérimentations

Les résultats obtenus dans la section précédente sont valables quelle que soit la méthode hors-ligne considérée. Par exemple, on les retrouve dans le graphe de marquage généré par le réseau de Petri proposé par [45], dans le volume engendré par le modèle géométrique proposé par [62, 64], ou encore dans le langage engendré par l'automate fini proposé dans [63]. Dans la partie IV, nous proposons une nouvelle méthode hors-ligne basée sur les chaînes de Markov. On retrouve une fois de plus les mêmes propriétés dans les distributions de probabilités obtenues. Dans la suite, nous vérifions expérimentalement nos résultats en utilisant les implémentations réalisées pour notre modèle basé sur les chaînes de Markov (voir partie IV).

6.4.1 Décroissance de la suite $(A_t)_{t \in r+P.\mathbb{N}}$

Considérons le système de tâches suivant (pour cible monoprocesseur) :

	r_k	C_k	D_k	T_k
τ_1	2	5	10	15
τ_2	0	3	8	15
τ_3	3	1	5	5
τ_4	5	4	10	15

Nous avons appliqué une méthode hors-ligne à ce système en évaluant le nombre d'états atteignables à chaque instant. Nous avons traité séparément les deux variantes que nous avons proposées pour intégrer la contrainte portant sur les échéances des tâches.

En utilisant le prédicat Q_{re}, le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints														
$t = 5..19$	26	60	102	14	34	44	51	7	4	9	2	4	6	7	18
$t = 20..34$	26	60	102	14	34	44	51	$\cdots$							

Pour le prédicat $Q_{re'}$, le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints														
$t = 5..19$	26	44	48	10	16	15	12	5	2	3	2	4	6	7	18
$t = 20..34$	26	44	48	10	16	15	12	$\cdots$							

Quel que soit le prédicat utilisé pour représenter la contrainte portant sur les échéances des tâches, on remarque que le nombre d'états atteints ne peut pas croître entre deux métapériodes successives (ici, il est constant). D'autre part, la suite $(A_t)_{t\in r+P.\mathbb{N}}$ converge dès l'instant $t = r$. L'utilisation du prédicat $Q_{re'}$ permet de diminuer significativement le nombre d'états à traiter.

6.4.2 Stabilisation de la suite $(A_t)_{t\in r+P.\mathbb{N}}$

Dans la section précédente, nous avons présenté un exemple où la suite $(A_t)_{t\in r+P.\mathbb{N}}$ converge dès l'instant $t = r$. Toutefois, comme le montre l'exemple suivant, l'instant r ne constitue pas une borne à la durée de convergence de $(A_t)_{t\in r+P.\mathbb{N}}$.

Considérons le système de tâches suivant (pour cible monoprocesseur) :

	r_k	C_k	D_k	T_k
τ_1	1	3	7	10
τ_2	4	3	4	5

En utilisant le prédicat $Q_{re'}$, le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints									
$t = 4..13$	4	8	6	4	1	1	2	2	3	2
$t = 14..23$	3	7	5	3	1	1	2	$\cdots$		

Pour ce système de tâches, la stabilisation de $(A_t)_{t\in r+P.\mathbb{N}}$ se produit à $t = 8 > r = 4$. Ainsi, r n'est pas une borne de la durée de stabilisation de la suite $(A_t)_{t\in r+P.\mathbb{N}}$.

Nous n'avons pas réussi à produire un exemple où cette stabilisation apparaît après $r + P$, et cela même lorsque EDF produit une séquence d'exécution dont la durée de montée en charge excède $r + P$. Reprenons l'exemple 3.1 présenté page 35. La séquence produite par EDF en biprocesseur pour ce système de tâches est cyclique dès $t = 14 > r + P = 11$. Etudions maintenant le nombre d'états atteignables à chaque instant en utilisant le prédicat $Q_{re'}$:

	Nombre d'états atteints							
$t = 3..10$	23	20	17	6	3	3	14	17
$t = 11..18$	23	20	17	$\cdots$				

La suite $(A_t)_{t\in r+P.\mathbb{N}}$ converge donc dès $t = 3 = r$, alors que la séquence produite par EDF (et donc prise en compte par les méthodes hors-ligne) est cyclique seulement à partir de $t = 14$. Cet exemple montre donc que les méthodes hors-ligne peuvent converger alors que certaines séquences sont encore en phase de montée en charge. Ce constat confère un avantage certain aux méthodes hors-ligne vis-à-vis de la durée de simulation par rapport aux politiques en-ligne comme EDF et LLF. Par exemple, à la page 36, nous avons présenté un système de tâches dont le régime cyclique de la séquence d'exécution produite par EDF en biprocesseur commence en $t = 139 > r+6P$. Nous avons appliqué notre méthode hors-ligne à ce système de tâches : elle converge dès $t = 11 = r$!

6.4.3 Intervalle de faisabilité

Le fait que nous n'ayons pas pu obtenir un système de tâches dont la durée de stabilisation excède $r + P$ nous a amené à conjecturer que $[0, r + P[$ puisse être un intervalle d'étude. Toutefois, l'exemple suivant montre que certains systèmes de tâches peuvent être ordonnançables jusqu'à un instant postérieur à $r + P$ bien qu'il n'existe aucune séquence infiniment valide.

Considérons le système de tâches suivant (pour cible monoprocesseur) :

	r_k	C_k	D_k	T_k
τ_1	4	3	5	5
τ_2	1	5	5	15

En utilisant le prédicat Q_{re}, le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints													
$t = 4..18$	4	9	1	2	3	1	2	3	4	4	1	2	3	7	11
$t = 19..33$	3	7	1	0	0										

Il existe donc des séquences valides jusqu'à l'instant $t = 21 > r + P = 19$, mais ensuite aucune séquence d'exécution ne permet de respecter toutes les contraintes.

Ces différents exemples nous amènent à conjecturer les deux points suivants :

- pour tout système de tâches ordonnançable, les méthodes hors-ligne convergent au plus tard en $r + P - 1$,
- $[0, r + 2P[$ est un intervalle de faisabilité multiprocesseur pour les méthodes hors-ligne.

6.4.4 Séquences d'exécution conservatives

Dans cette section, nous considérons les méthodes hors-ligne qui ne prennent en compte que les séquences conservatives.

Considérons le système de tâches suivant (pour cible biprocesseur) :

	r_k	C_k	D_k	T_k
τ_1	3	5	7	8
τ_2	0	4	7	8
τ_3	0	4	6	8
τ_4	0	1	4	8

En utilisant le prédicat Q_{re} et en considérant toutes les séquences d'exécution (conservative et non-conservative), le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints							
$t = 3..10$	23	20	17	6	3	3	14	17
$t = 11..18$	23	20	17	$\cdots$				

En utilisant le prédicat Q_{re} et en considérant cette fois seulement les séquences d'exécution conservatives, le nombre d'états atteints à chaque instant est indiqué dans le tableau suivant :

	Nombre d'états atteints							
$t = 3..10$	3	5	3	2	2	2	9	8
$t = 11..18$	8	11	8	4	3	3	12	13
$t = 19..26$	14	16	13	5	3	3	$\cdots$	

Cet exemple montre que le fait de ne considérer que les séquences d'exécution conservatives repousse la stabilisation de la suite $(A_t)_{t \in r+P.\mathbb{N}}$. De plus, on remarque que la propriété de décroissance de la suite $(A_t)_{t \in r+P.\mathbb{N}}$ n'est pas vérifiée : à l'instant $t = 3$, il y a 3 états atteignables, et à la métapériode suivante en $t = 11$, il y a 8 états atteignables. Ainsi, le fait que le nombre d'états converge n'implique pas forcément que les ensembles A_t correspondant soient égaux. Notre analyse ne s'applique donc pas directement aux méthodes hors-ligne considérant seulement les séquences conservatives, une étude supplémentaire est alors nécessaire.

6.5 Conclusion

Pour les méthodes hors-ligne traitant les séquences conservatives *et* non-conservatives, nous avons proposé une méthode permettant d'obtenir une durée de simulation finie : l'ensemble des états atteignables est décroissant et finit par se stabiliser. Ainsi, en comparant le nombre d'états atteignables à deux instants distants d'exactement une métapériode, on peut décider la terminaison des méthodes hors-ligne.

Nous avons observé expérimentalement la durée d'étude nécessaire pour les méthodes hors-ligne. Nous avons montré qu'elle peut être inférieure à $r + 2P$ même lorsque des séquences d'exécution comme celles de EDF ne deviennent cycliques qu'après $r + P$, et nécessitent donc un intervalle

d'étude plus grand que $[0, r + 2P]$. De plus, nous n'avons pas réussi à construire un système de tâches nécessitant une durée d'étude pour les méthodes hors-ligne qui soit supérieure à $r + 2P$. Toutefois, nous n'avons pas pu établir formellement une telle borne. Ce point-là semble être une voie intéressante de recherche puisqu'il rendrait les méthodes hors-ligne plus rapidement simulables pour les systèmes multiprocesseurs que les politiques en-ligne.

La prise en compte des seules séquences conservatives semble remettre en question un certain nombre de nos résultats. L'étude des méthodes hors-ligne considérant seulement ces séquences nécessite une analyse supplémentaire. Finalement, nous avons considéré ici les architectures multiprocesseurs et préemptives avec migration globale. D'autres types d'exécutifs peuvent aussi être étudiés.

Troisième partie

Contributions à la production de solutions d'ordonnancement

Chapitre 7

Production des configurations de priorités fixes valides

7.1 Introduction

De nombreux exécutifs temps-réel sont basés sur un nombre limité de niveaux de priorité [105]. Généralement, à chaque tâche est attribué un niveau de priorité distinct de celui des autres tâches. Les ordonnanceurs utilisés par ces exécutifs appartiennent à la classe des politiques à priorités fixes. Pour les systèmes de tâches indépendantes à départs simultanés ordonnancés en contexte monoprocesseur préemptif, il existe des politiques à priorités fixes optimales dans la classe PFX, par exemple DM [71]. Malheureusement, dans tout contexte plus général, il n'existe pas de politique à priorités fixes optimale dans PFX. L'attribution des priorités constitue donc l'un des enjeux majeurs lors de la conception de ces systèmes temps-réel.

Pour le contexte monoprocesseur préemptif, [9] propose un algorithme optimal dans PFX de complexité pseudo-polynomiale pour attribuer les priorités aux tâches. Leur méthode repose sur la propriété suivante : les temps de réponse de la tâche de plus faible priorité ne dépendent pas de l'ordre de priorité des autres tâches. L'algorithme consiste alors à rechercher une tâche qui puisse avoir la priorité la plus faible :

1. on sélectionne une tâche τ_k pour être celle de plus faible priorité,
2. on attribue des priorités quelconques aux autres tâches, pourvu qu'elles soient supérieures à celle de τ_k,
3. on simule l'exécution en observant les temps de réponse de τ_k,
4. si toutes les instances de τ_k respectent leur échéance :
   - alors on attribue la plus faible priorité à τ_k et on répète l'algorithme avec $\tau \setminus \{\tau_k\}$,
   - sinon on recherche une autre tâche pour être celle de plus faible priorité.

Cette méthode est optimale pour attribuer des priorités fixes aux tâches. A chaque étape, on doit tester au plus n tâches comme candidate à la plus faible priorité. La complexité de cette méthode est donc en $O(n^2.S)$, où S est la complexité du processus permettant de déterminer si toutes les instances d'une tâche respectent leur échéance. Généralement, aucune condition d'ordonnançabilité polynomiale ne peut être utilisée, S est donc de complexité exponentielle.

En contexte multiprocesseur, la propriété à la base de cet algorithme n'est plus vérifiée [6]. Il ne peut donc pas être étendu au contexte multiprocesseur sans perdre l'optimalité de l'attribution des priorités. Dans ce chapitre, nous développons une approche permettant d'attribuer des priorités fixes pour les contextes multiprocesseurs. Tout d'abord, nous caractérisons les séquences d'exécutions productibles par des politiques à priorités fixes. A partir d'une séquence d'exécution valide quelconque, nous proposons ensuite une méthode pour déterminer s'il existe une attribution de priorités fixes qui permette d'engendrer cette séquence. Nous produisons aussi l'ensemble de ces

attributions lorsqu'il en existe. Ce résultat permet de transformer n'importe quelle séquence d'exécution valide (par exemple obtenue par une politique en-ligne ou par une méthode hors-ligne) en une séquence d'exécution identique mais produite par une politique à priorités fixes. Les exécutifs temps-réel reposant généralement sur des priorités fixes, notre méthode permet de leur intégrer des séquences d'exécution différentes de celles fournies par les politiques à priorités fixes classiques qui, rappelons-le, ne sont généralement pas optimales.

Finalement, nous complétons notre approche par une méthode permettant de déterminer l'ensemble des configurations de priorité fixe qui permettent d'ordonnancer un système de tâches donné, sans énumérer toutes les $n!$ configurations possibles. Notre méthode est optimale pour attribuer des priorités fixes en contexte multiprocesseur préemptif avec migration totale. La complexité de notre algorithme est proportionnelle au nombre de configurations valides. Nos expérimentations confirment ce point puisque le temps de calcul utilisé par notre algorithme diminue lorsque la charge du système de tâches augmente.

Dans la section 7.3, nous étudions les relations de priorité induites par une séquence d'exécution. Ensuite, nous donnons une méthode pour déterminer les configurations de priorité fixe engendrant une séquence d'exécution donnée. Finalement, nous proposons un algorithme pour déterminer les configurations de priorité fixe ordonnançant un système de tâches donné. Dans la section 7.5, nous appliquons notre méthode au cas des systèmes de tâches indépendantes exécutées en multiprocesseur, et dans la section 7.6, nous évaluons expérimentalement la complexité de calcul de notre méthode, ainsi que la puissance d'ordonnancement qu'elle permet d'atteindre.

7.2 Contexte d'étude

Nous étudions les séquences d'ordonnancement multiprocesseurs préemptifs avec migration totale engendrées par des politiques à priorités fixes. Nous considérons les systèmes de tâches indépendantes, à échéances avant ou sur requête, et à départs différés. Ainsi, dans toute séquence valide, il existe à chaque instant au plus une instance active de chaque tâche. Cette propriété permet de représenter les séquences d'exécutions valides en indiquant, à la place des instances actives, les tâches dont l'instance courante est active. Pour chaque instant t, l'instance courante de chaque tâche $\tau_k \in \tau$ est définie par :

- inexistante, si $t < r_k$,
- $\tau_{k,\lfloor (t-r_k)/T_k \rfloor}$, si $t \geq r$.

La définition des séquences d'exécution que nous avons donnée dans la section 2.2 repose sur les instances. Dans ce chapitre, et en vertu de la propriété ci-dessus, nous représentons les séquences d'exécution à l'aide des tâches dont l'instance courante est active.

Dans toute la suite, nous désignons par S_τ l'ensemble des séquences d'exécution de τ. Pour une séquence $s \in S_\tau$ donnée, on désigne par $C_{k,i,t}(s)$ le nombre d'instants où la $i^{\text{è}}$ instance de τ_k a eu accès à un processeur :

$$C_{k,i,t}(s) = \sum_{t'=r_{k,i}}^{min\{t, r_{k,i}+T_k-1\}} |s_t \cap \{\tau_k\}|$$

Nous supposons que toutes les instances de $\tau_k \in \tau$ requièrent exactement C_k accès à un processeur pour son exécution. Or, une tâche est active à un instant donné si et seulement si son instance courante n'a pas encore terminé son exécution. Une tâche $\tau_k \in \tau$ est donc active à l'instant t si et seulement si :

$$t \geq r_k \wedge C_{k,\lfloor (t-r_k)/T_k \rfloor, t}(s) < C_k$$

Les politiques à priorités fixes attribuent la même priorité aux différentes instances d'une même tâche. De plus, cette priorité ne varie pas avec le temps. Ainsi, pour étudier l'ensemble des séquences d'exécution engendrables par des politiques à priorités fixes, on peut se ramener à l'ensemble des $n!$ permutations possibles des tâches composant le système τ. Nous adoptons la définition suivante.

Définition 7.1 *Une configuration de priorité fixe pour τ est une bijection de τ dans $\{1, \ldots, n\}$.*

Une configuration de priorité fixe permet d'engendrer une unique séquence d'exécution. La définition suivante établit le lien entre les tâches actives à un instant t et celles qui seront exécutées à cet instant.

Définition 7.2 *Soit A une configuration de priorité fixe pour τ. On note $s(A,p)$ la séquence d'exécution engendrée par A sur p processeurs :*

$$s(A,p)_t = \{\tau_k \in s(A,p)^\star_t || E_k \cap s(A,p)^\star_t| < p\}$$

où $E_k = \{\tau_{k'} \in \tau | A(\tau_{k'}) < A(\tau_k)\}$.

Remarquons qu'une séquence $s(A,p)$ produite par un algorithme à priorités fixes A est toujours conservative. Dans la suite, nous disons aussi qu'elle est p-conservative, ou encore qu'elle est (p,t)-conservative pour indiquer qu'elle l'est sur l'intervalle $[0,t[$.

7.3 Ordonnancement en priorités fixes

Dans cette section, nous établissons les correspondances entre les configurations de priorité fixe, les séquences engendrables par une configuration de priorité fixe et les relations de priorité.

Tout d'abord, rappelons quelques notions sur les relations binaires. Pour toute relation binaire R, on désigne par $\overline{R}$ sa clôture transitive. Une relation binaire R sur un ensemble E est totale si et seulement si pour tout $a, b \in E$, $a \neq b$, on a soit $(a,b) \in R$, soit $(b,a) \in R$. Une relation binaire R sur un ensemble E contient un cycle si et seulement s'il existe $a, b \in \tau$ tels que $(a,b) \in \overline{R}$ et $(b,a) \in \overline{R}$.

7.3.1 Relation de priorité induite

Lorsqu'une séquence d'exécution est produite par un algorithme à priorités, les tâches choisies pour être exécutées renseignent sur les relations entre les priorités des tâches. Lorsqu'un algorithme à priorités élit la tâche τ_k alors que plusieurs tâches sont actives, la tâche τ_k est alors la plus prioritaire. Par exemple, pour l'instant 0 de la séquence de la figure 7.1, on obtient :

- les tâches actives sont τ_1, τ_2, τ_3
- la tâche choisie par l'ordonnanceur est τ_1
- en conséquence, on doit avoir : $prio(\tau_1) > prio(\tau_2)$ et $prio(\tau_1) > prio(\tau_3)$

A chaque instant de décision, la priorité des tâches exécutées est supérieure à celle des tâches actives qui ne sont pas exécutées. La définition suivante formalise cette propriété et l'étend au cas où plusieurs tâches peuvent être élues simultanément par l'ordonnanceur (cas multiprocesseur).

Définition 7.3 *Soient $A \subset \tau$ l'ensemble des tâches actives et $B \subset A$ l'ensemble des tâches exécutées. On appelle $RP_A(B)$ les relations de priorité induites par l'exécution simultanée des tâches de B :*

$$RP_A(B) = \{(\tau_i, \tau_j) | \tau_i \in B \text{ et } \tau_j \in A \setminus B\}$$

Ainsi, chaque instant $t \in \mathbb{N}$ induit des relations de priorité entre les tâches. En supposant que la priorité des tâches n'évolue pas (c'est l'hypothèse des priorités fixes), les relations de priorité induites à un instant donné sont valables à chaque instant de l'exécution. Ainsi, en unissant les relations induites à chaque instant $t \in \mathbb{N}$, on obtient une description des relations entre les priorités

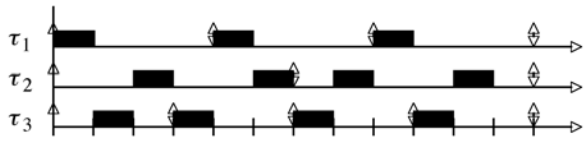


FIG. 7.1 – Exemple de séquence d'exécution

des tâches correspondant à une séquence d'exécution. La table 7.2 donne les relations induites par la séquence d'exécution indiquée sur la figure 7.1.

On adopte la définition suivante des relations de priorité induites par une séquence d'exécution.

Définition 7.4 *Soit $s \in S_\tau$. On appelle $RP_t(s)$ les relations de priorité induites par s à l'instant $t \in \mathbb{N}$:*

$$RP_t(s) = RP_{s_t^\star}(s_t)$$

On appelle $RP(s)$ les relations de priorité induites par s :

$$RP(s) = \overline{\bigcup_{t\in\mathbb{N}} RP_t(s)}$$

La définition suivante caractérise les relations de priorité induites par une configuration de priorité fixe.

Définition 7.5 *Soit A une configuration de priorité fixe. On appelle $RP(A)$ les relations de priorité induites par A :*

$$RP(A) = \{(\tau_i, \tau_j) \in \tau^2 | A(\tau_i) > A(\tau_j)\}$$

La relation $RP(A)$ ne contient jamais de cycle, quelle que soit la configuration de priorité fixe A.

7.3.2 Relation de priorité fixe

Dans cette section, nous déterminons à partir des relations de priorité une condition nécessaire et suffisante pour qu'une séquence d'exécution soit engendrable par une configuration de priorité fixe.

Lemme 7.1 *Soit A une configuration de priorité fixe. On a $RP(s(A,p)) \subset RP(A)$.*

Démonstration :
Soient $t \in \mathbb{N}$ et $(\tau_i, \tau_j) \in RP_t(s(A,p))$. Par définition de $RP_t(s)$, on a $\tau_i \in s(A,p)_t$ et $\tau_j \in s(A,p)_t^\star \setminus s(A,p)_t$. On déduit de la définition de $s(A,p)$ que $A(\tau_i) > A(\tau_j)$. La définition de $s(A,p)$ entraîne $(\tau_i, \tau_j) \in RP(A)$.
CQFD.

Théorème 7.1 *Soit $s \in S_\tau$. S'il existe une configuration A de priorité fixe telle que $s(A,p) = s$, alors s est p-conservative et $RP(s)$ ne contient aucun cycle.*

t	$RP_{s_t^\star}(s_t)$
0	(τ_1, τ_2) et (τ_1, τ_3)
1	(τ_3, τ_2)
2	
3	(τ_3, τ_2)
4	(τ_1, τ_2)
5	
6	(τ_3, τ_2)
7	
8	(τ_1, τ_2)
9	(τ_3, τ_2)
10	
11	
$\mathbb{N}$	(τ_1, τ_2) et (τ_1, τ_3) et (τ_3, τ_2)

FIG. 7.2 – Relations de priorité induites par la séquence de la figure 7.1

Démonstration :

Par hypothèse, il existe une configuration A de priorité fixe vérifiant $s(A,p) = s$. Pour une telle configuration de priorité A, d'après le lemme 7.1, on a $RP(s) = RP(s(A,p)) \subset RP(A)$. Or par définition, $RP(A)$ ne contient aucun cycle, il en est donc de même pour $RP(s)$. De plus par définition, $s(A,p)$ est p-conservative, et il en est donc de même pour s.

CQFD.

Lemme 7.2 *Soient A une configuration de priorité fixe, et $s \in S_\tau$ une séquence p-conservative vérifiant $RP(s) \subset RP(A)$. Pour tout $t \in \mathbb{N}$, on a :*

$$s_t^\star = s(A,p)_t^\star \Rightarrow s_t = s(A,p)_t$$

Démonstration :

Soit $t \in \mathbb{N}$ tel que $s_t^\star = s(A,p)_t^\star$. Si $s(A,p)_t = s(A,p)_t^\star$, la séquence s étant p-conservative, on obtient alors $s_t = s_t^\star$, et donc $s_t = s(A,p)_t$.

Supposons maintenant que $s(A,p)_t \neq s(A,p)_t^\star$. Il existe alors τ_i, τ_j tels que $\tau_i \in s(A,p)_t$ et $\tau_j \in s(A,p)_t^\star \setminus s(A,p)_t$. Pour tout τ_i, τ_j vérifiant cette propriété, par définition de $s(A,p)$ on a $A(\tau_i) > A(\tau_j)$. Par définition de $RP(A)$, on obtient aussi $(\tau_i, \tau_j) \in RP(A)$. Or, $RP(A)$ ne contient pas de cycle et par hypothèse $RP(s) \subset RP(A)$, ainsi (τ_j, τ_i) ne peut pas appartenir à $RP(s)$. En conséquence, on ne peut pas avoir $\tau_j \in s_t$ et $\tau_i \in s_t^\star \setminus s_t$. Les cas possibles sont donc les suivants :

1. $\tau_i, \tau_j \in s_t$
2. $\tau_i, \tau_j \in s_t^\star \setminus s_t$
3. $\tau_i \in s_t$ et $\tau_j \in s_t^\star$

Considérons tout d'abord le cas (1). Par hypothèse, s et $s(A,p)$ sont p-conservatives et $s_t^\star = s(A,p)_t^\star$, on a donc $|s_t| = |s(A,p)_t|$. Or, dans le cas (1), on a $\tau_i, \tau_j \in s_t$, et $\tau_i \in s(A,p)_t$ et $\tau_j \in s(A,p)_t^\star \setminus s(A,p)_t$. Il existe donc une tâche τ_k vérifiant $\tau_k \in s(A,p)_t$ et $\tau_k \notin s_t$. On obtient donc $(\tau_k, \tau_j) \in RP(A)$ et $(\tau_j, \tau_k) \in RP(s)$. Or par hypothèse, $RP(s) \subset RP(A)$ et $RP(A)$ ne contient pas de cycle, on obtient donc une contradiction.

Considérons maintenant le cas (2). Symétriquement au cas (1), on obtient qu'il existe une tâche τ_k vérifiant $\tau_k \in s_t$ et $\tau_k \notin s(A,p)_t$. On en déduit que $(\tau_k, \tau_i) \in RP(s)$ et $(\tau_i, \tau_k) \in RP(A)$. De même que dans le cas (1), on obtient une contradiction.

Ainsi, les cas (1) et (2) sont impossibles, et le cas (3) amène à $s_t = s(A,p)_t$.

CQFD.

Théorème 7.2 *Soit $s \in S_\tau$. Si s est p-conservative et si $RP(s)$ ne contient aucun cycle, alors il existe une (unique, si $RP(s)$ est totale) configuration A de priorité fixe telle que $s(A,p) = s$.*

Démonstration :

Par hypothèse, $RP(s)$ ne contient aucun cycle. On peut donc compléter $RP(s)$ en une relation R totale et sans cycle. Par définition de R, il existe une configuration de priorité fixe A telle que $RP(A) = R$.

Montrons maintenant par récurrence sur t la propriété suivante :

$$\forall k \in \{1, \ldots, n\}, \forall i, t \in \mathbb{N}, C_{k,i,t}(s) = C_{k,i,t}(s(A,p))$$

Pour $t = 0$, cette propriété est effectivement vérifiée. Supposons-la maintenant vraie pour t, et montrons qu'elle l'est pour $t+1$. Puisqu'elle est vraie au rang t, on a par définition $s_{t+1}^\star = s(A,p)_{t+1}^\star$. D'après le lemme 7.2, on a aussi $s_{t+1} = s(A,p)_{t+1}$. Par définition, on obtient donc :

$$\forall k \in \{1, \ldots, n\}, \forall i \in \mathbb{N}, C_{k,i,t+1}(s) = C_{k,i,t+1}(s(A,p))$$

Ainsi cette propriété est vraie pour tout $t \in \mathbb{N}$, on obtient donc aussi pour tout $t \in \mathbb{N}$, $s_t^\star = s(A,p)_t^\star$, et en conséquence $s_t = s(A,p)_t$.

Pour toute configuration de priorité A engendrant s, on a $RP(s) \subset RP(A)$. Si $RP(s)$ est totale, on a alors $RP(s) = RP(A)$, et A est donc unique. CQFD.

Nous pouvons finalement définir la notion de relations de priorité fixe.

Définition 7.6 *Une relation binaire transitive et acyclique sur τ est une relation de priorité fixe entre les tâches de τ. On note RP_τ l'ensemble de ces relations.*

7.3.3 Configurations PFX engendrant une séquence donnée

Dans cette section, nous proposons une méthode pour déterminer les configurations de priorité fixe engendrant une séquence d'exécution donnée. Considérons une séquence d'exécution s conservative. En construisant la relation $RP(s)$, le théorème 7.2 nous permet de conclure :

- si $RP(s)$ contient un cycle, alors il n'existe aucune configuration de priorité fixe qui engendre s,
- si $RP(s)$ est acyclique, alors il existe au moins une configuration de priorité fixe qui engendre s.

La relation $RP(s)$ permet aussi de déterminer l'ensemble des configurations de priorité fixe engendrant la séquence s, lorsqu'il en existe. Considérons le résultat suivant.

Théorème 7.3 *Soient A une configuration de priorité fixe et s une séquence d'exécution p-conservative. La séquence produite par A est identique à s si et seulement si :*

$$RP(s) \subset RP(A)$$

Démonstration :
Le lemme 7.2 montre :

$$RP(s) \subset RP(A) \Rightarrow s = s(A, p)$$

D'autre part, on a $RP(s(A,p)) \subset RP(A)$, on obtient alors :

$$s = s(A, p) \Rightarrow RP(s) = RP(s(A, p)) \Rightarrow RP(s) \subset RP(A)$$

CQFD.

Pour déterminer l'ensemble des configurations de priorité fixe engendrant une séquence conservative s donnée, il suffit donc d'énumérer toutes les configurations A vérifiant $RP(s) \subset RP(A)$. Pour cela, nous proposons la méthode suivante. Observons tout d'abord cette propriété : les relations de priorité correspondent à des relations d'ordre entre les priorités des tâches. Ainsi, pour obtenir les configurations de priorité correspondant à une relation de priorité, il faut attribuer aux priorités des valeurs numériques qui respectent l'ordre indiqué par la relation.

Supposons tout d'abord que la relation RP(s) est totale. Ainsi, il existe une tâche τ_i vérifiant :

$$\forall \tau_k \in \tau, k \neq i \Rightarrow (\tau_i, \tau_k) \in RP(s)$$

La tâche τ_i est donc la plus prioritaire selon l'ordre établi par $RP(s)$, sa priorité est donc n. Considérons maintenant la relation R obtenue en restreignant $RP(s)$ à $\tau \setminus \{\tau_i\}$. La relation R est aussi totale, et il existe donc une tâche τ_j vérifiant :

$$\forall \tau_k \in \tau \setminus \{\tau_i\}, k \neq j \Rightarrow (\tau_j, \tau_k) \in R$$

La priorité attribuée à la tâche τ_j est donc $n-1$. De proche en proche, on attribue ainsi une priorité à chacune des tâches.

Supposons maintenant que la relation $RP(s)$ soit partielle. Alors, il existe au moins un couple de tâches (τ_i, τ_j) vérifiant $(\tau_i, \tau_j) \notin RP(s)$ et $(\tau_j, \tau_i) \notin RP(s)$. Les priorités de τ_i et de τ_j ne sont donc pas départagées ; ceci signifie que lors de la génération de la séquence s, il n'est jamais

nécessaire de comparer la priorité de τ_i avec celle de τ_j. La relation $RP(s)$ correspond alors à plusieurs configurations de priorité. Pour les obtenir, on construit deux relations intermédiaires :

$$R_1 = \overline{R \cup \{(\tau_i, \tau_j)\}} \wedge R_2 = \overline{R \cup \{(\tau_j, \tau_i)\}}$$

Si les relations R_1 et R_2 sont totales, alors on construit les configurations de priorité correspondantes en utilisant la méthode que nous avons donnée ci-dessus. Dans le cas contraire, on continue à compléter les deux relations jusqu'à ce qu'elles soient totales.

La méthode que nous proposons permet de transformer une séquence d'exécution quelconque en une séquence identique et produite par une configuration de priorité fixe. Elle permet alors d'intégrer dans les exécutifs temps-réel à priorités fixes des séquences d'exécution obtenues par n'importe quelle stratégie : génération par une politique en-ligne, utilisation d'une méthode hors-ligne exacte ou approchée, etc.

7.4 Recherche des configurations PFX valides

7.4.1 (t, p)-consistance

Intuitivement, on peut dire qu'une relation est consistante si et seulement si à chaque instant elle désigne sans ambiguïté les tâches à exécuter. Par exemple, la relation de priorité induite par un ordre total sur les priorités des tâches est consistante.

Pour illustrer la notion de (t, p)-consistance, prenons l'exemple de la relation $R = \{(\tau_1, \tau_2), (\tau_1, \tau_3)\}$ et du système de tâches suivant :

	r_i	C_i	D_i	T_i
τ_1	0	1	4	4
τ_2	0	2	6	6
τ_3	0	1	3	3

La figure 7.3 présente le début de séquence engendrée par R pour ce système de tâches.

A l'instant $t = 0$, les trois tâches sont actives. La relation R indique que τ_1 est plus prioritaire que τ_2 et que τ_3. Ainsi, en respectant les relations de priorité fournies par R, seule la tâche τ_1 peut être exécutée. Il n'y a donc pas d'ambiguïté possible. Donc, R est $(1, 1)$-consistante. La notion de consistance est paramétrée par (t, p) : t indique la longueur de la séquence pour laquelle la relation

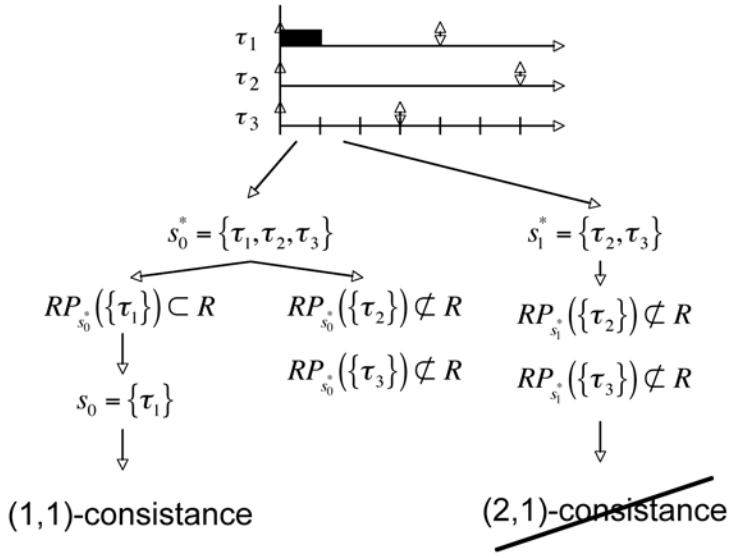


FIG. 7.3 – Illustration de la (t, p)-consistance

doit être consistante, et p indique le nombre de processeurs disponibles, ce dernier paramètre influe puisque l'on impose à la séquence d'être conservative.

A l'instant $t = 1$, la tâche τ_1 n'est plus active. Or, la relation R ne permet pas de départager τ_2 et τ_3. Ainsi, tout en respectant les relations de priorité induites par R, τ_1 ou τ_2 peut être exécutée. Il y a donc une ambiguïté et nous disons que R n'est pas $(2,1)$-consistante.

Définition 7.7 *Soient $R \in RP_\tau$, un instant $t \in \mathbb{N}$. La relation R est (t,p)-consistante si et seulement s'il existe une séquence $s \in S_\tau$ qui soit p-conservative et telle que $\cup_{t'=0}^{t-1} RP_{t'}(s) \subset R$. Une telle séquence s est dite compatible avec R.*

On note $RP_{\tau,t,p}$ l'ensemble des relations de priorité sur τ qui sont (t,p)-consistantes.

Lemme 7.3 *Soient un instant $t \in \mathbb{N}$ et $R \in RP_{\tau,t,p}$. Pour tout $s, s' \in S_\tau$ compatibles avec R et tout $t' < t$, on a $s_{t'} = s'_{t'}$.*

Démonstration :

Par définition, R est (t,p)-consistante, elle ne contient donc aucun cycle. Il existe donc une configuration de priorité fixe A telles que $R \subset RP(A)$. Par hypothèse, on a $\cup_{t'=0}^{t-1} RP_{t'}(s) \subset R \subset RP(A)$. Cela correspond à un cas particulier du lemme 7.2, car ses hypothèses requièrent $RP(s) \subset RP(A)$. Dans le cas présent, ses conclusions sont donc valides jusqu'à l'instant $t-1$. On obtient donc $\forall t' < t, s^\star_{t'} = s(A,p)^\star_{t'} \Rightarrow s_{t'} = s(A,p)_{t'}$. En utilisant cette propriété de la même manière que dans la preuve du théorème 7.2, on montre que $\forall t' < t, s_{t'} = s(A,p)_{t'}$. En raisonnant de la même manière pour s', on obtient alors $\forall t' < t, s_{t'} = s'_{t'}$.

CQFD.

Ainsi, pour chaque relation (t,p)-consistante, il existe une unique séquence de longueur t qui est compatible avec celle-ci. Dans la suite, on la note $s(R,p,t)$. Pour un problème de cohérence, nous devons aussi définir les valeurs de $s(R,p,t)$ pour $t' > t$, pour cela, on adopte la convention suivante :

$$\forall t' \geq t, s(R,p,t)_{t'} = \emptyset$$

La propriété suivante exprime la monotonie de la (t,p)-consistance.

Lemme 7.4 *Soient $t \in \mathbb{N}$, $R \in RP_{\tau,t,p}$, et $R' \in RP_\tau$. Si $R \subset R'$, alors R' est (t,p)-consistante et $s(R,p,t) = s(R',p,t)$.*

Démonstration :

Par définition, $s(R,p,t)$ est compatible avec R. Or, on a par hypothèse $R \subset R'$. On en déduit que $s(R,p,t)$ est aussi compatible avec R'. Donc, R' est bien (t,p)-consistante. Par définition, $s(R',p,t)$ est compatible avec R'. Ainsi, $s(R,p,t)$ et $s(R',p,t)$ sont compatibles avec R'. D'après le lemme 7.3, on obtient $\forall t' < t, s(R,p,t)_{t'} = s(R',p,t)_{t'}$. Par définition de $s(.,p,t)$, on a aussi $\forall t' \geq t, s(R,p,t)_{t'} = s(R',p,t)_{t'} = \emptyset$.

CQFD.

7.4.2 Base des relations consistantes

La représentation directe de l'ensemble des configurations de priorité fixe engendre une forte complexité (majoré par $|\tau|!$). En utilisant les relations de priorité, on peut diminuer cette complexité. Pour cela, on introduit la notion de *base* de $RP_{\tau,t,p}$.

Définition 7.8 *Un sous-ensemble E de $RP_{\tau,t,p}$ est une base de $RP_{\tau,t,p}$ si et seulement si pour toute séquence $s \in S_\tau$ engendrable par une relation de priorité de $RP_{\tau,t,p}$, il existe une unique $R \in E$ vérifiant $\forall t' < t, s(R,p,t)_{t'} = s_{t'}$*

Le cardinal des bases de $RP_{\tau,t,p}$ est égal au nombre de séquences engendrables jusqu'à l'instant t par une configuration de priorité fixe. Sa complexité ne dépend donc pas du nombre de configurations de priorité fixe ($|\tau|!$), mais du nombre de configurations de priorité fixe engendrant des séquences différentes.

Une base de $RP_{\tau,t,p}$ permet de caractériser les séquences recherchées, cependant elle ne permet pas toujours de retrouver facilement l'ensemble des configurations de priorité fixe. Prenons par exemple une base composée uniquement par des configurations de priorité fixe (relations totales), et supposons qu'il existe deux configurations différentes engendrant la même séquence. Ainsi, seule l'une des deux est représentée dans la base et le seul moyen de savoir si l'autre est valide consiste à comparer les séquences d'exécution respectives. Cependant, une base particulière existe pour laquelle on peut résoudre ce problème simplement en comparant les relations de priorité.

Définition 7.9 *Soient $t \in \mathbb{N}$ et $R \in RP_{\tau,t,p}$. La relation R est minimale si et seulement si $R = RP(s(R,p,t))$.*

On note $B_{\tau,t,p}$ l'ensemble des relations minimales de $RP_{\tau,t,p}$.

Comme l'indique le théorème suivant, l'ensemble $B_{\tau,t,p}$ forme une base de $RP_{\tau,t,p}$. Dans la suite, on l'appelle la base minimale.

Théorème 7.4 *Soient $R \in RP_\tau$, un instant $t \in \mathbb{N}$. Les deux assertions suivantes sont équivalentes :*

- *R est (t,p)-consistante ($R \in RP_{\tau,t,p}$)*
- *$\exists! R' \in B_{\tau,t,p}$ telle que $R' \subset R$*

L'ensemble $B_{\tau,t,p}$ forme une base de $RP_{\tau,t,p}$.

Démonstration :

Supposons $\exists! R' \in B_{\tau,t,p}$ telle que $R' \subset R$. Par définition, R' est (t,p)-consistante et $R' \subset R$. D'après le lemme 7.4, R est (t,p)-consistante.

Supposons que R est (t,p)-consistante. La relation $RP(s(R,p,t))$ vérifie $RP(s(R,p,t)) \subset R$. Elle est de plus (t,p)-consistante et minimale, on a donc $RP(s(R,p,t)) \in B_{\tau,t,p}$. Soit $R' \in B_{\tau,t,p}$ telle que $R' \subset R$. D'après le lemme 7.4, on a $s(R',p,t) = s(R,p,t)$. De plus, par définition, on a $R' = RP(s(R',p,t))$. On obtient donc $R' = RP(s(R,p,t))$.

CQFD.

Ainsi, la base $B_{\tau,t,p}$ permet de déterminer si une relation $R \in RP_\tau$ est (t,p)-consistante, simplement en déterminant si l'une de ses relations inclut R. Cette base est la seule à vérifier cette propriété.

7.4.3 Q-validité

Les relations consistantes permettent de représenter les configurations de priorité fixe et les séquences qu'elles engendrent. Cependant, les séquences d'exécution doivent aussi satisfaire d'autres contraintes : par exemple, échéances des tâches, exclusion mutuelle des accès aux ressources partagées.

Définition 7.10 *Une contrainte sur τ est un prédicat Q dépendant du temps sur les séquences d'exécution de τ. Une séquence $s \in S_\tau$ est (Q,t)-valide si et seulement si $\forall t' < t, Q_t(s)$. Une séquence (Q,t)-valide pour tout $t \in \mathbb{N}$ est dite Q-valide.*

Définition 7.11 *Soient Q une contrainte sur τ et un instant $t \in \mathbb{N}$. On appelle $B_{\tau,t,p,Q}$ l'ensemble obtenu en restreignant la base minimale de $RP_{\tau,t,p}$ aux relations générant une séquence (Q,t)-valide.*

Les relations de $B_{\tau,t,p,Q}$ permettent de décider si une relation de priorité quelconque (et donc une configuration de priorité fixe quelconque) engendre une séquence d'exécution valide pour Q jusqu'à l'instant t. Cela se fait sans comparer les séquences d'exécution respectives mais simplement en comparant les relations.

Théorème 7.5 *Soient Q une contrainte sur τ, un instant $t \in \mathbb{N}$, pour toute relation $R \in RP_{\tau,t,p}$, les deux assertions suivantes sont équivalentes :*

- *R engendre une séquence (Q,t)-valide*
- *$\exists! R' \in B_{\tau,t,p,Q}$ vérifiant $R' \subset R$*

L'ensemble $B_{\tau,t,p,Q}$ forme une base des relations de $RP_{\tau,t,p}$ engendrant une séquence (Q,t)-valide.

Démonstration :

Soit $R \in RP_{\tau,t,p}$ une relation générant une séquence (Q,t)-valide. D'après le théorème 7.4, il existe une unique $R' \in B_{\tau,t,p}$ telle que $R' \subset R$. D'après le lemme 7.4, R et R' génèrent la même séquence. Par hypothèse, $s(R,p,t)$ est (Q,t)-valide, il en est donc de même pour $s(R',p,t)$. Or, $B_{\tau,t,p,Q}$ est la restriction $B_{\tau,t,p}$ aux relations générant une séquence (Q,t)-valide. Ainsi $R' \in B_{\tau,t,p,Q}$ est la seule relation de $B_{\tau,t,p,Q}$ à vérifier $R' \subset R$.

Soit $R' \in B_{\tau,t,p,Q}$ vérifiant $R' \subset R$. D'après le lemme 7.4, R et R' génère la même séquence. Or $R' \in B_{\tau,t,p,Q}$, ainsi $s(R',p,t)$ est (Q,t)-valide, il en est donc de même pour $s(R,p,t)$.

CQFD.

Une configuration de priorité fixe correspond à une relation de priorité totale ($RP(A) \in RP_{\tau,t,p}$). Le théorème précédent s'applique donc aux configurations de priorité fixe.

7.4.4 Construction de la base minimale

Dans cette section, nous présentons une méthode pour calculer $B_{\tau,t,p,Q}$. Elle procède récursivement sur t, et consiste à construire pour chaque relation R de $B_{\tau,t,p,Q}$ les relations correspondant aux séquences obtenues en exécutant à l'instant t toutes les combinaisons de tâches possibles. Les combinaisons de tâches exécutables pour une relation donnée doivent vérifier plusieurs contraintes : la séquence s obtenue doit être valide pour Q, être p-conservative, et $RP(s)$ ne doit pas contenir de cycle.

Définition 7.12 *Soient $s \in S_\tau$ une séquence (t,p)-conservative et telle que $\overline{\cup_{t'=0}^{t-1} RP_{t'}(s)}$ soit sans cycle, et $a \subset s_t^\star$ l'ensemble des tâches exécutées à l'instant t. On appelle $(s.a)$ la séquence définie par :*

$$\begin{cases} \forall t' < t, (s.a)_{t'} = s_{t'} \\ (s.a)_t = a \\ \forall t' > t, (s.a)_{t'} = \emptyset \end{cases}$$

Lemme 7.5 *Soient $t \in \mathbb{N}$, $s \in S_\tau$ une séquence (t,p)-conservative et telle que $\overline{\cup_{t'=0}^{t-1} RP_{t'}(s)}$ soit sans cycle, et $a \subset s_{t+1}^\star$ l'ensemble des tâches exécutées. La séquence $(s.a)$ est $(t+1,p)$-conservative et $\overline{\cup_{t'=0}^{t} RP_{t'}(s.a)}$ est acyclique si et seulement si les deux conditions suivantes sont vérifiées :*

- *$|a| = min\{p, |s_t^\star|\}$*
- *$\forall \tau_i \in a, \forall \tau_j \in s_t^\star \setminus a, (\tau_j, \tau_i) \notin \overline{\cup_{t'=0}^{t-1} RP_{t'}(s)}$*

On note $CCL_{\tau,s,t,p}$ l'ensemble des actions a vérifiant les conditions données ci-dessus.

Démonstration :

Remarquons tout d'abord que $(s.a)_t^\star = s_t^\star$ et que $\overline{\cup_{t'=0}^{t-1} RP_{t'}(s)} = \overline{\cup_{t'=0}^{t-1} RP_{t'}(s.a)}$.

Supposons que les deux conditions de l'énoncé du théorème soient vérifiées. Par hypothèse, s est (t,p)-conservative et $|a| = min\{p, |s_t^\star|\}$. Ainsi, $s.a$ est bien $(t+1,p)$-conservative. D'après la deuxième condition, les couples (τ_j, τ_i) correspondant à un couple (τ_i, τ_j) de $RP_{(s.a)_t^\star}(a)$ n'appartiennent pas à $\overline{\cup_{t'=0}^{t-1} RP_{t'}(s)}$. De plus par hypothèse, $\overline{\cup_{t'=0}^{t-1} RP_{t'}(s)}$ ne contient aucun cycle, ainsi, il ne peut pas y avoir de cycle dans $\overline{\cup_{t'=0}^{t} RP_{t'}(s.a)}$.

Supposons que $(s.a)$ est $(t+1,p)$-conservative et que $\overline{\cup_{t'=0}^{t} RP_{t'}(s.a)}$ soit sans cycle. Par définition de la $(t+1,p)$-conservativité, on a bien $|a| = min\{p, |s_t^\star|\}$. Ainsi, la première des deux conditions est bien vérifiée. Par définition, on a $RP_{(s.a)_t^\star}(a) = \{(\tau_i, \tau_j) | \tau_i \in a \wedge \tau_j \in (s.a)_t^\star \setminus a\}$. Or,

par hypothèse, on a aussi $\overline{\cup_{t'=0}^{t} RP_{t'}(s.a)}$ sans cycle. Ainsi, pour tout $(\tau_i, \tau_j) \in RP_{(s.a)_t^\star}(a)$, on a aussi $(\tau_j, \tau_i) \notin \overline{\cup_{t'=0}^{t-1} RP_{t'}(s.a)}$. Donc, la deuxième condition est également vérifiée.

CQFD.

Théorème 7.6 *Soient Q une contrainte sur τ. On a :*

- $B_{\tau,0,p,Q} = \{\emptyset\}$
- $\forall t \in \mathbb{N}, B_{\tau,t+1,p,Q} = \bigcup_{R \in B_{\tau,t,p,Q}} \bigcup_{a \in ACT} \overline{R \cup RP_{s(R,t,p)_t^\star}(a)}$

où $ACT = \{a \in CCL_{\tau,s(R,t,p),t,p} | Q_{t+1}(s.a)\}$

Démonstration :

La relation $\emptyset$ est $(0,p) - consistante$, elle est aussi minimale, on a alors $B_{\tau,0,p,Q} = \{\emptyset\}$.

Soit $t \in \mathbb{N}$, et posons $E = \cup_{R \in B_{\tau,t,p,Q}} \cup_{a \in ACT} \overline{R \cup RP_{s(R,t,p)_t^\star}(a)}$.

Soient $R \in B_{\tau,t,p,Q}$, $a \in ACT$ et $R' = \overline{R \cup RP_{s(R,t,p)_t^\star}(a)}$. Par définition, on a $s(R', p, t+1) = (s(R,p,t).a)$. Or, par hypothèse et d'après le lemme 7.5, $RP(s(R,p,t).a)$ ne contient pas de cycle, $s(R,p,t).a$ est $(p, t+1)$-conservative et $(Q, t+1)$-valide, il en est donc de même pour $s(R', p, t+1)$. On a aussi $RP(s(R,p,t).a) = R'$, ainsi R' est minimale. On a donc $R' \in B_{\tau,t+1,p,Q}$.

Soit $R \in B_{\tau,t+1,p,Q}$. Posons $R' = RP(s(R,p,t))$. Par construction, on a $R' \in B_{\tau,t,p,Q}$. Par hypothèse, $s(R,p,t+1)$ est $(p,t+1)$-conservative et $(Q,t+1)$-valide. D'après le lemme 7.5, $s(R,p,t+1)_{t+1} \in ACT$, et donc $RP(s(R,p,t+1)) \in E$. Par hypothèse, R est minimal, on a donc $R = RP(s(R,p,t+1))$. On obtient donc $R \in E$.

CQFD.

Le théorème précédent permet de construire la base minimale des relations de priorité fixe engendrant des séquences valides selon Q. Grâce à elle, on peut déterminer simplement si une configuration de priorité fixe est valide. On peut aussi construire l'ensemble des configurations de priorité fixe valides en appliqant la méthode donnée dans la section 7.3.3 à chaque relation appartenant à la base.

7.5 Tâches indépendantes

Dans cette section, nous appliquons notre méthode au cas des tâches indépendantes ordonnancées en priorités fixes en contexte multiprocesseur et préemptif avec migration totale.

7.5.1 Contrainte de validité

Dans le cas des tâches indépendantes, la seule propriété que doivent vérifier les configurations de priorité fixe est le respect des échéances des tâches.

Définition 7.13 *Soit $s \in S_\tau$, l'instance courante d'une tâche $\tau_k \in \tau$ a une laxité nulle à l'instant $t \in \mathbb{N}$ dans la séquence s si et seulement si la condition suivante est vérifiée :*

$$D_{k,\lfloor (t-r_k)/T_k \rfloor, t} - t = C_k - C_{k,\lfloor (t-r_k)/T_k \rfloor, t}(s)$$

On note $s_t^\#$ l'ensemble des tâches de laxité nulle à l'instant t dans la séquence s.

Lemme 7.6 *Soit $s \in S_\tau$. Les deux assertions suivantes sont équivalentes :*

- *toutes les échéances sont respectées*
- $\forall t \in \mathbb{N}, s_t^\# \subset s_t$

Démonstration :

Supposons que $\forall t \in \mathbb{N}, s_t^\# \subset s_t$ soit vérifiée. Ainsi, aucune tâche n'atteint une laxité négative. Donc, toutes les échéances sont respectées.

Supposons que $\forall t \in \mathbb{N}, s_t^{\#} \subset s_t$ ne soit pas vérifiée. Il existe alors une tâche de laxité nulle qui n'est pas exécutée, sa laxité devient donc négative. Ainsi, il y a au moins une échéance qui n'est pas respectée.

CQFD.

On note V_t la contrainte sur τ définie par $V_t(s) \Leftrightarrow (s_t^{\#} \subset s_t)$.

7.5.2 Stabilisation de $B_{\tau,t,p,V}$

Le théorème 7.6 fournit une méthode récursive pour calculer la base V-minimale. L'objet de cette section est de déterminer la condition d'arrêt de cette récurrence.

Les séquences d'exécution monoprocesseurs produites par des configurations de priorité fixe sont cycliques de période P au pire à partir de $r + P$ [21]. L'intervalle d'étude nécessaire pour obtenir les configurations de priorité fixe valides est donc $[0, r + 2P]$.

Lorsque les tâches sont à départs simultanés, alors les séquences d'exécution multiprocesseurs produites par des configurations de priorité fixe sont cycliques de période P dès $t = 0$ [24]. L'intervalle d'étude nécessaire pour obtenir les configurations de priorité fixe valides est donc $[0, P]$.

Pour ces deux premiers cas, on dispose d'un intervalle d'étude qui ne dépend pas de la configuration de priorité fixe utilisée. Ainsi, la base $B^V_{M,p}(\tau)$ décrit les configurations de priorité fixe infiniment valides, respectivement avec :

1. $M = r + 2P$, pour les séquences monoprocesseurs.
2. $M = P$, pour les systèmes de tâches à départs simultanés.

Pour les systèmes de tâches à départs différés et ordonnancés en multiprocesseur par une configuration de priorité fixe, [22, 24] montrent que les séquences d'exécution produites sont cycliques. De plus, [24] borne aussi la durée de montée en charge. Dans les chapitres 4 et 5, nous avons affiné ces résultats. Toutefois, le calcul de la durée de montée en charge dépend de l'ordre de priorité entre les tâches. Or, notre méthode considère simultanément plusieurs configurations de priorité fixe. Elle requiert donc un majorant de la durée de montée en charge indépendant de l'ordre des priorités. Cependant, le calcul de la durée de montée en charge de chaque configuration de priorité possibles ($n!$) au total est peu envisageable. Toutefois, les majorants de la durée de montée en charge que nous avons proposés dans le chapitre 5 permettent de montrer facilement que $r + \sum_{k=p+1}^{n} T_k$ est une borne supérieure valable quelle que soit la configuration de priorité fixe utilisée.

Pour l'implémentation de notre algorithme, nous avons voulu minimiser au maximum la durée de simulation à effectuer. Pour cela, on utilise une méthode de détection "à la volée" de l'entrée dans le régime cyclique. Considérons un instant $t \geq r + P$. Si l'état des tâches est le même en t et en $t - P$ alors la séquence d'exécution est cyclique à partir de $t - P$, et l'intervalle d'étude est $[0, t]$. L'état d'une tâche τ_k à l'instant t dans une séquence d'exécution s est décrit par la quantité $C_{k,\lfloor (t-r_k)/T_k \rfloor}(s, t)$. A partir de ce résultat, nous pouvons déterminer "à la volée" l'instant où les séquences d'exécution sont toutes cycliques et donc l'instant à partir duquel la base $B^V_{t,p}(\tau)$ est stable.

7.5.3 Algorithme de calcul de $B_{\tau,t,p,V}$

L'algorithme que nous proposons construit récursivement $B_{\tau,t,p,V}$ et implémente le critère d'arrêt correspondant au cas général. A chaque instant $t \in \mathbb{N}$, la structure de données utilisée correspond à une liste de triplets composés ainsi :

- R : une relation de priorité
- $I_k = C_{k,\lfloor (t-r_k)/T_k \rfloor, t}(s(R, p, t))$: la situation courante de chaque tâche τ_k
- $(M_k)_{\tau_k \in \tau}$: la situation des tâches au début de la métapériode courante

Remarquons que la connaissance de I et de l'instant courant t permet de déterminer l'état de l'ensemble du système : on peut en déduire par exemple la laxité de chaque tâche ($lax_k(I, t)$), les tâches actives ($act(I, t)$). Par convention, lorsque la tâche τ_k n'est pas active, on considère que sa laxité est infinie ($\tau_k \notin act(I, t) \Rightarrow lax_k(I, t) = +\infty$).

La figure 7.4 indique l'algorithme construisant les relations de $B_{\tau,t+1,p,V}$ résultant d'une relation de $B_{\tau,t,p,V}$. Pour obtenir la base minimale stabilisée, nous suivons la méthode suivante. A chaque étape, l'algorithme réinitialise les champs I_k des tâches τ_k qui se réactivent à cet instant-là. Ensuite, il recherche les séquences cycliques. Enfin, il construit $B_{\tau,t+1,p,V}$ à partir de $B_{\tau,t,p,V}$. Remarquons que dès que l'algorithme détecte une séquence cyclique, il retire la relation correspondante de B et la place dans res. Lorsque l'algorithme termine, res contient la base minimale stabilisée. La figure 7.5 indique l'algorithme construisant la base minimale stabilisée.

Le nombre d'éléments contenus dans $B_{\tau,t,p,V}$ correspond au nombre de séquences valides et engendrables par une configuration de priorité fixe, il ne dépend donc pas du nombre de configurations de priorité fixe possibles ($|\tau|!$).

Pour une relation $R' \in B_{\tau,t+1,p,V}$, il existe une unique relation $R \in B_{\tau,t,p,V}$ et une unique action $a \subset act(I)$ telles que $R' = \overline{R \cup RP_{act(I)}(a)}$. Ainsi, la base B construite par l'algorithme ne contient pas de doublon : chaque relation n'y apparaît qu'une seule fois. Cette propriété est très

```
rec(t, R, I, M, B', a, k) :
 s := Σ_{k=1}^{|τ|} a_k
 si k > |τ| alors
  si s = min{p, |act(I,t)|} alors
   valid := VRAI
   pour tout τ_i ∈ τ tel que a_i = 1,
   pour tout τ_j ∈ act(I,t) tel que a_j = 0
    si (τ_j, τ_i) ∈ R alors valid := FAUX
   si valid = VRAI alors
    B' := B' ∪ (R ∪ RP_{act(I,t)}(a)‾, I + a, M)
 sinon
  si s < p et τ_k ∈ act(I,t) alors
   rec(t, R, I, M, B', (a_1, .., a_{k-1}, 1, a_{k+1}, .., a_n), k + 1)
  si lax_k(I,t) > 0 alors
   rec(t, R, I, M, B', (a_1, .., a_{k-1}, 0, a_{k+1}, .., a_n), k + 1)
```

FIG. 7.4 – Algorithme de calcul des relations de $B_{\tau,t+1,p,V}$ résultant d'une relation $R \in B_{\tau,t,p,V}$

```
t := 0
B := {(∅, (0, ..., 0), (−1, ..., −1))}
res := ∅
tant que B ≠ ∅ faire
 pour tout k tel que t ∈ r_k + T.ℕ alors
  pour tout (R, I, M) ∈ B faire I_k := 0
 si t ∈ r + P.ℕ alors
  pour tout (R, I, M) ∈ B faire
   si I = M alors
    res := res ∪ R
    B := B \ {(R, I, M)}
   si M = (−1, ..., −1) alors M := I
 B' := ∅
 pour tout (R, I, M) ∈ B faire
  rec(t, R, I, M, B', (0, ..., 0), 1)
 B := B'
 t := t + 1
```

FIG. 7.5 – Algorithme de calcul de $B_{\tau,t,p,V}$

importante : si elle n'était pas vérifiée, il serait nécessaire, à chaque ajout d'une relation dans la base, de vérifier si elle n'y est pas déjà ; ceci entraînerait une complexité très importante.

L'algorithme utilise deux opérations sur les relations de priorité : la recherche d'un couple (τ_i, τ_j) dans la clôture transitive d'une relation R, et l'ajout d'un couple (τ_i, τ_j) à la clôture transitive d'une relation R, sachant que celui-ci n'entraîne pas de cycle. Afin de réaliser ces deux opérations efficacement, il est nécessaire de développer une structure de donnée adaptée.

On peut optimiser l'algorithme à partir du constat suivant. Tant que les p tâches les plus prioritaires (pour une relation donnée) restent actives, l'algorithme effectue des calculs redondants : à chaque instant, il recherche toutes les actions possibles et détermine les relations engendrées. Or, il n'y a qu'une seule action possible : continuer à exécuter les p tâches les plus prioritaires, et alors la relation de priorité n'évolue pas. Comme les instants d'activation des tâches sont connus et que la durée d'exécution des tâches est aussi connue, on peut prévoir les instants auxquels la combinaison de tâches en cours d'exécution risque de changer. Aux yeux de l'algorithme, seuls ces instants-là sont importants, on peut donc réaliser des "sauts temporels" au lieu de traiter chaque instant.

7.6 Expérimentations

7.6.1 Un exemple d'utilisation de la méthode PFX

Etudions la validation du système de tâches suivant sur une architecture biprocesseur :

	r_k	C_k	D_k	T_k
τ_1	15	7	11	38
τ_2	47	1	8	38
τ_3	4	4	43	45
τ_4	17	8	13	19
τ_5	43	3	3	6
τ_6	22	8	11	19
τ_7	30	6	25	25

Ce système de tâches n'est pas ordonnançable par RM et DM. De plus, même les politiques à priorités dynamiques EDF et LLF échouent à l'ordonnancer. Appliquons maintenant notre méthode à ce système de tâches.

La base minimale, stabilisée dès $t = 17147$ et produite par l'algorithme de la section 7.5.3, contient une seule relation de priorité (notée R dans la suite) :

```
La tâche 1 est plus prioritaire que la tâche 2.
La tâche 1 est plus prioritaire que la tâche 3.
La tâche 1 est plus prioritaire que la tâche 4.
La tâche 1 est plus prioritaire que la tâche 6.
La tâche 1 est plus prioritaire que la tâche 7.

La tâche 2 est plus prioritaire que la tâche 3.

La tâche 4 est plus prioritaire que la tâche 2.
La tâche 4 est plus prioritaire que la tâche 3.
La tâche 4 est plus prioritaire que la tâche 6.
La tâche 4 est plus prioritaire que la tâche 7.

La tâche 5 est plus prioritaire que la tâche 2.
La tâche 5 est plus prioritaire que la tâche 3.
La tâche 5 est plus prioritaire que la tâche 4.
La tâche 5 est plus prioritaire que la tâche 6.
La tâche 5 est plus prioritaire que la tâche 7.

La tâche 6 est plus prioritaire que la tâche 2.
```

```
La tâche 6 est plus prioritaire que la tâche 3.
La tâche 6 est plus prioritaire que la tâche 7.

La tâche 7 est plus prioritaire que la tâche 2.
La tâche 7 est plus prioritaire que la tâche 3.
```

La base n'étant pas vide, ce système de tâches est donc ordonnançable en priorités fixes. Remarquons que la relation R n'est pas totale puisque les tâches τ_1 et τ_5 ne sont pas départagées. En appliquant la méthode de la section 7.3.3, on obtient deux configurations de priorité compatibles avec R.

Task	$R_1 = \overline{R \cup \{(\tau_1, \tau_5)\}}$	$R_2 = \overline{R \cup \{(\tau_5, \tau_1)\}}$
τ_1	7	6
τ_2	2	2
τ_3	1	1
τ_4	5	5
τ_5	6	7
τ_6	4	4
τ_7	3	3

Cet exemple montre clairement que notre méthode permet d'obtenir des solutions d'ordonnancement nouvelles par rapport aux politiques habituelles. Il montre aussi que des algorithmes à priorités dynamiques comme EDF et LLF sont complémentaires avec les algorithmes à priorités fixes. D'autre part, le calcul de la base minimale et des deux configurations de priorité fixe valides prend moins d'un dixième de seconde sur un Apple-G5 à 2,5GHz.

Dans le chapitre 5, nous avons montré (théorème 5.1) que l'ordre de priorité entre les p tâches les plus prioritaires n'a pas d'influence sur la séquence produite. On retrouve ici cette propriété puisque la relation contenue dans la base ne départage pas les deux tâches les plus prioritaires (τ_1 et τ_5).

7.6.2 Performances moyennes

Dans cette section, nous comparons notre méthode, appelée PFX, avec les politiques d'ordonnancement RM, DM, EDF, LLF, et aussi avec l'algorithme RM-US$[m/(3m-2)]$ défini dans [5].

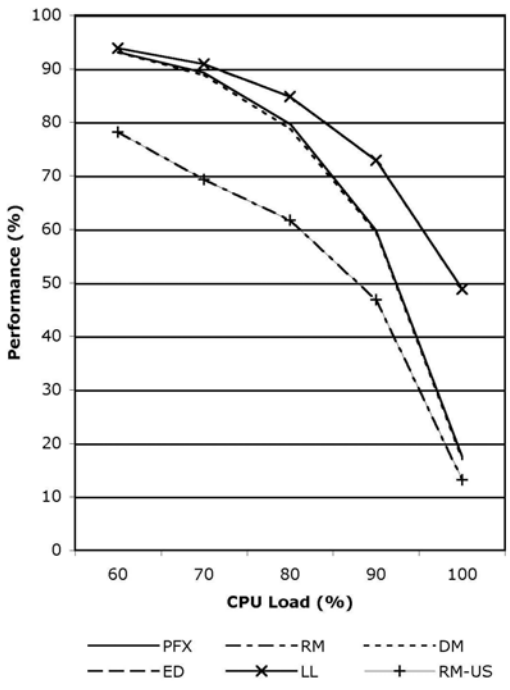
Nous avons généré aléatoirement plusieurs échantillons de systèmes de tâches. Chaque échantillon contient 1000 systèmes de tâches définis pour un nombre de processeurs donné ($p \in \{1, 2, 3, 4\}$) et une charge CPU donnée ($U \in \{p - 0,5, \ldots, p - 0,1, p\}$). Les paramètres des tâches sont attribués aléatoirement en utilisant la méthode suivante :

1. les périodes (T_k) sont choisies uniformément entre 1 et 100,
2. les durées d'exécution (C_k) sont choisies uniformément entre 1 et 40,
3. les échéances (D_k) sont choisies uniformément entre C_k et T_k,
4. les dates de première activation (r_k) sont choisies uniformément entre 0 et T_k.

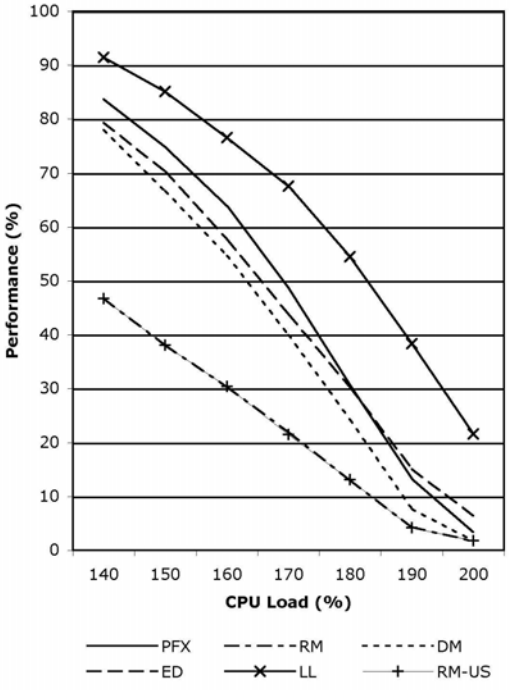
Chaque système de tâches est généré itérativement en ajoutant une nouvelle tâche tant que la charge CPU désirée n'est pas atteinte. En moyenne, chaque système contient une dizaine de tâches.

Chaque système de tâches est simulé à l'aide de chaque méthode d'ordonnancement. Ainsi, nous obtenons pour chaque échantillon et chaque méthode d'ordonnancement le nombre de systèmes de tâches qu'elle permet d'ordonnancer. Ce nombre décrit la puissance d'ordonnancement de chaque méthode. Nos résultats présentent la puissance d'ordonnancement des méthodes RM, DM, EDF, LLF, RM-US$[m/(3m-2)]$ et PFX (voir figure 7.6). Nous nous intéressons particulièrement à la courbe PFX qui correspond à la méthode que nous présentons ici et qui est représentative de la puissance d'ordonnancement des exécutifs temps-réel à priorités fixes.

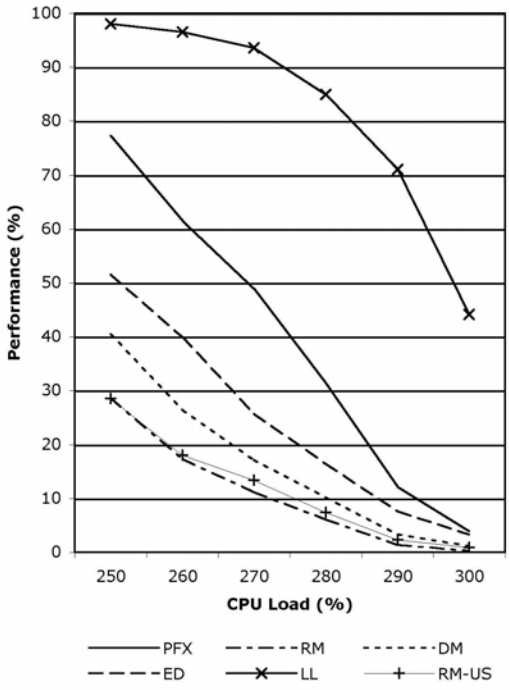
Tous les systèmes de tâches ne sont pas ordonnançables. Toutefois, EDF et LLF sont optimaux en monoprocesseur, ainsi leur performance indique dans ce contexte-là le taux de systèmes ordonnançables. Les performances de DM et de PFX sont très proches bien que DM ne soit optimal en



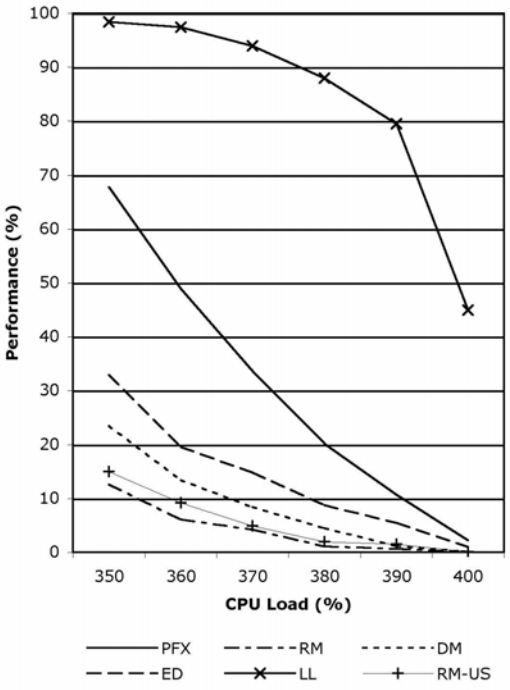
1 processeur



2 processeurs



3 processeurs



4 processeurs

FIG. 7.6 – Puissance d'ordonnancement de chaque méthode

monoprocesseur que lorsque les tâches sont à départs simultanés. La politique DM, bien que n'étant pas toujours optimale en monoprocesseur, est donc une très bonne politique d'ordonnancement en priorités fixes. Ainsi, l'utilisation d'autres politiques à priorités fixes en monoprocesseur ne semble pas prometteuse.

Inversement, aucune politique en-ligne n'est optimale en multiprocesseur [34]. Dans ce contexte, les résultats indiqués sur la figure 7.6 montrent que LLF domine toutes les autres méthodes. Cependant, les ordonnancements produits par LLF contiennent un grand nombre de changements de contexte. Généralement, le coût CPU induit n'est pas acceptable. En fait, dans nos expérimentations, LLF sert principalement de référence pour approximer le taux de systèmes de tâches ordonnançables.

Naturellement, PFX domine les trois autres politiques à priorités fixes (RM, DM et RM-US$[m/(3m-2)]$). Ainsi, aucune des trois ne permet d'exploiter pleinement la puissance de l'ordonnancement en priorités fixes. De plus, nos résultats montrent aussi que PFX domine EDF. Ce résultat est surprenant : il est plus intéressant d'ordonnancer en priorités fixes avec une attribution des priorités adaptée à chaque système que d'utiliser une politique d'ordonnancement à priorités dynamiques comme EDF qui est pourtant largement reconnue pour ses qualités dans ce domaine. La puissance d'ordonnancement atteinte paraît très encourageante pour les méthodes "hors-ligne" de calcul de priorités fixes comme PFX.

La figure 7.7 présente le temps de calcul moyen utilisé par PFX sur un Apple-G5 à 2,5GHz pour un système de tâches. Le temps de calcul varie selon l'inverse de la charge des systèmes de tâches. C'est une conséquence directe de l'utilisation des bases pour représenter l'ensemble des solutions à un instant t : la complexité est proportionnelle au nombre de séquences valides de longueur t, au lieu du nombre de configurations de priorité fixe possibles ($n!$). Donc, plus les systèmes de tâches sont contraints, moins il y a de solutions et donc, plus le calcul est court. Cette propriété est très intéressante et permet d'envisager l'utilisation de PFX avec des systèmes de taille industrielle.

7.7 Conclusion

Nous avons présenté une nouvelle méthode pour obtenir des configurations de priorité valides. Tout d'abord, nous avons déterminé les conditions nécessaires pour qu'une séquence d'exécution donnée soit productible par une configuration de priorité fixe. Ce résultat permet d'intégrer les séquences d'exécution produites par n'importe quelle approche, lorsque c'est possible. Ensuite, nous avons proposé une méthode pour déterminer *toutes* les configurations de priorité fixe en contexte multiprocesseur et préemptif avec migration totale qui ordonnancent correctement un système de tâches donné. Les expérimentations ont montré plusieurs points :

1. la puissance d'ordonnancement en priorités fixes dépasse celle de EDF,
2. aucune politique en-ligne à priorités fixes testée ne permet d'exploiter pleinement la puissance

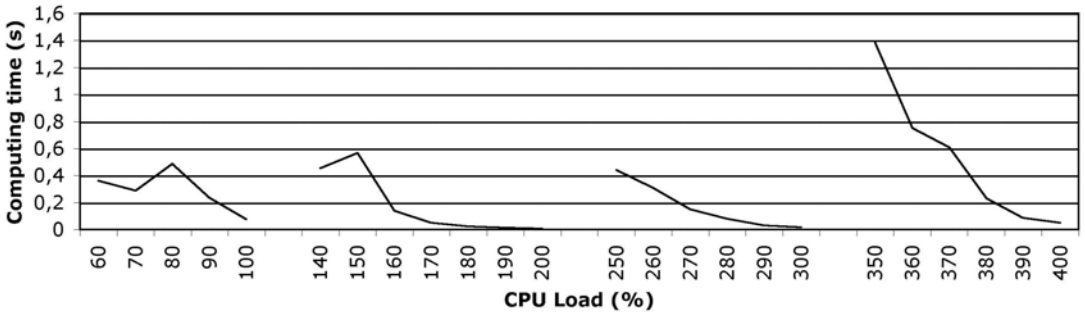


FIG. 7.7 – Temps de calcul moyen de PFX

d'ordonnancement en priorités fixes, le recours à des méthodes comme PFX paraît donc nécessaire,

3. PFX permet d'ordonnancer certains systèmes de tâches qui ne sont ordonnançables ni par RM, ni par DM, ni par EDF et ni par LLF, notre méthode permet donc d'obtenir de nouvelles solutions pour ordonnancer un système de tâches.

L'algorithme que nous avons proposé peut être optimisé sur plusieurs points (voir section 7.5.3). De plus, le traitement de chaque relation de la base étant indépendant des autres relations, on peut facilement le paralléliser.

D'autre part, notre méthode est très modulable. Au lancement de l'algorithme, la base est initialisée avec la relation vide ; cette valeur correspond à une recherche exhaustive des configurations de priorité fixe valides. En choisissant une autre relation initiale, on peut guider l'algorithme vers un sous-ensemble des solutions. Ceci rend possible deux nouvelles utilisations de l'algorithme :

1. le concepteur du système temps-réel peut introduire des contraintes sur les priorités des tâches en imposant par exemple qu'une tâche soit plus prioritaire qu'une autre,
2. plus généralement, on peut effectuer une résolution incrémentale en appliquant l'algorithme à des sous-ensembles du système de tâches et en combinant ensuite les solutions obtenues ; différentes mises en œuvre peuvent ici être envisagées.

La possibilité de faire une résolution incrémentale peut être très utile pour appliquer notre méthode à de "gros" systèmes de tâches.

Notre implémentation de PFX est basée sur une recherche en largeur des solutions : elle est donc orientée vers une recherche exhaustive des solutions. En utilisant une recherche en profondeur, l'algorithme se focalisera sur la recherche d'une solution, cette implémentation permet donc d'obtenir des résultats plus rapidement. Cette approche permet aussi d'introduire de l'interactivité dans le déroulement de l'algorithme : par exemple, à chaque obtention d'une solution, on peut décider de stopper ou de continuer la recherche, on peut aussi guider l'algorithme vers une solution particulière en choisissant pour lui la prochaine relation à utiliser et donc implicitement les prochaines solutions qu'il va explorer.

Notre méthode a été définie pour traiter les systèmes de tâches périodiques et indépendantes. Toutefois, la validité des séquences est définie par un prédicat Q, elle est donc "générique". On peut donc introduire facilement de nouvelles contraintes de validation comme les relations de précédence, la non-préemption, l'exclusion mutuelle ou le partage de ressources. Toutefois, lorsque les tâches sont interdépendantes, on ne peut plus représenter l'état d'une instance par les seuls états "active" et "exécutée", il faut aussi considérer qu'une tâche peut être bloquée. Les relations de priorité engendrées à un instant donné ne dépendent pas des tâches bloquées puisque celles-ci ne sont pas éligibles pour être exécutées. Elles dépendent donc seulement des tâches "prêtes" et "exécutées" : les tâches "exécutées" étant plus prioritaires que les tâches "prêtes". Une fois ces considérations prises en compte, notre méthode pourra être hybridée avec des protocoles de gestion de ressources. On pourra alors déterminer les configurations de priorité fixe qui ordonnancent un système en utilisant par exemple PIP ou PCP.

Les travaux menés dans ce chapitre ont fait l'objet d'un article soumis à une revue internationale en 2007.

Chapitre 8

Configurations de priorités fixes par instances

8.1 Introduction

Dans le chapitre 7, nous avons étudié la puissance des ordonnancements à priorités fixes en contexte préemptif avec migration totale. Ceux-ci se révèlent très efficaces lorsque la charge du système de tâches n'est pas trop élevée ($< 80\%$). Cependant, les systèmes de tâches à plus de 80% de charge sont courants puisque l'on évite généralement de sur-dimensionner les architectures matérielles. Pour ceux-ci, il est donc nécessaire d'utiliser une classe d'ordonnanceurs plus performants.

Dans le chapitre 3, nous avons présenté deux nouvelles classes de politiques d'ordonnancement ($\overline{\text{PFI}}$ et $\overline{\overline{\text{PFI}}}$). Nous avons ensuite montré que la politique EDF appartient à $\overline{\text{PFI}}$. Cette propriété confère à $\overline{\text{PFI}}$ une certaine puissance d'ordonnancement : EDF est optimal en monoprocesseur, ainsi dès qu'un système de tâches est ordonnançable en monoprocesseur, il existe une politique de $\overline{\text{PFI}}$ ordonnançant ce système. Par ailleurs, ces deux classes sont justes supérieures à PFX : elles se positionnent en intermédiaires entre les politiques à priorités fixes et celles à priorités dynamiques. Pour obtenir de nouvelles solutions d'ordonnancement, ces constats nous ont amenés à nous intéresser à ces deux classes de politiques d'ordonnancement.

Dans la section 8.2, nous déterminons expérimentalement la puissance d'ordonnancement de la classe PFI en contexte préemptif avec migration totale. Pour cela, nous adaptons la méthode développée dans le chapitre 7 au traitement des configurations de priorité fixe par instances. Dans la section 8.3, nous étudions le positionnement de EDF par rapport à $\overline{\overline{\text{PFI}}}$, et nous en déduisons un résultat d'optimalité pour cette classe de politiques d'ordonnancement. La classe $\overline{\overline{\text{PFI}}}$ présentant des propriétés intéressantes, nous avons approfondi notre étude. Dans la section 8.4, nous proposons plusieurs procédés permettant de calculer pour un système de tâches donné les configurations de priorité valides qui appartiennent à $\overline{\overline{\text{PFI}}}$.

8.2 Ordonnançabilité dans PFI

Dans cette section, nous ramenons une instance du problème PFI (déterminer les configurations PFI qui ordonnancent un système de tâches) à une instance du problème PFX (déterminer les configurations PFX qui ordonnancent un système de tâches). Ensuite, nous utilisons la méthode donnée dans le chapitre 7 pour résoudre le problème PFI.

8.2.1 Formulation de PFI dans PFX

Considérons un système de tâches τ et associons-lui un système, noté $\tau^\star$, composé de tâches *non périodiques* :

$$\tau_k \in \tau \Rightarrow \forall i \in \mathbb{N}, (r_{k,i}, C_k, d_{k,i}) \in \tau^\star$$

Ainsi, pour chaque tâche $\tau_k \in \tau$, on introduit une infinité de tâches dans $\tau^\star$ qui correspondent aux instances successives de τ_k (voir figure 8.1).

Le système $\tau^\star$ ne vérifie pas tout à fait les hypothèses utilisées pour la résolution du problème PFX : les tâches ne sont pas périodiques. Toutefois, ceci ne modifie en rien la manière dont sont traitées les relations de priorité entre les tâches. Effectivement, celles-ci ne dépendent que des tâches actives et des tâches exécutées à un instant donné, elles ne dépendent donc pas directement de la règle d'activation des tâches.

On peut donc appliquer les théorèmes 7.5 et 7.6 au système $\tau^\star$ afin de construire la base des relations de priorité minimales pour $\tau^\star$ engendrant une séquence valide. Cette base décrit les relations de priorité entre les tâches de $\tau^\star$. Or, les tâches de $\tau^\star$ correspondent aux instances des tâches de τ. Cette base permet donc en réalité de décrire les relations de priorité entres les instances des tâches de τ. Elle fournit donc les configurations de priorité fixe par instances qui ordonnancent correctement τ. La figure 8.2 résume cette méthode.

Ces considérations nous amènent au constat suivant : *toute configuration PFI de τ est une configuration PFX de $\tau^\star$, et réciproquement.* Il en découle naturellement que τ est ordonnançable par PFI si et seulement si $\tau^\star$ est ordonnançable par PFX.

8.2.2 Algorithme de décision

La transformation de τ et $\tau^\star$ introduit une infinité de tâches. Dans cette section, nous dérivons un algorithme qui détermine s'il existe une configuration PFI valide à partir de celui donné dans le chapitre 7 pour les configurations PFX. Grâce à lui, on peut éviter d'utiliser le système $\tau^\star$, et donc le nombre de tâches considérées reste $|\tau|$.

Dans le cadre de PFX, les relations de priorité engendrées à un instant donné sont valables à chaque instant. Dans celui de PFI, les relations de priorité engendrées à un instant donné ne sont valables que pour les instances courantes de chaque tâche. Ainsi, au lieu d'utiliser des relations de priorité portant sur l'ensemble des instances des tâches, comme le préconise la transformation

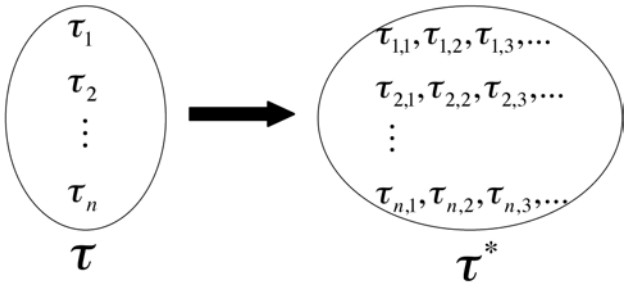


FIG. 8.1 – Association de $\tau^\star$ à τ

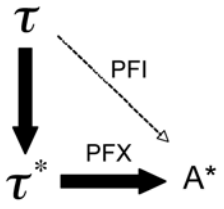


FIG. 8.2 – Résolution de PFI par PFX

correspondant au système $\tau^\star$, on peut utiliser des relations de priorité portant seulement sur l'instance courante de chaque tâche.

Or, l'instance courante des tâches change à chaque réactivation, ainsi les instances représentées dans les relations de priorité évoluent au cours du temps. Ceci introduit une nouvelle étape dans l'algorithme : à chaque terminaison d'une instance, on doit réinitialiser les relations de priorité correspondant à cette tâche, afin que l'instance suivante puisse avoir n'importe quelle priorité.

Cependant, cette réinitialisation peut engendrer des redondances : un même triplet (R, I, M) peut être ajouté à B de plusieurs manières. Dans PFX, ce cas de figure est impossible, l'algorithme de recherche des configurations de priorité fixe valides (voir section 7.5.3) ne contient donc pas de mécanisme de détection des doublons. Il est alors nécessaire d'en ajouter un, dans la suite nous le

```
rec(t, R, I, M, B', a, k) :
 s := Σ_{k=1}^{|τ|} a_k
 si k > |τ| alors
  si s = min{p, |act(I,t)|} alors
   valid := VRAI
   pour tout τ_i ∈ τ tel que a_i = 1,
   pour tout τ_j ∈ act(I,t) tel que a_j = 0
    si (τ_j, τ_i) ∈ R alors valid := FAUX
   si valid = VRAI alors
    b := ∪(R ∪ RP_{act(I,t)}(a), I + a, M)
    pour tout τ_i ∈ τ tel que I_i = C_i − 1 et a_i = 1,
    pour tout τ_j ∈ τ faire
     b := b \ {(τ_i, tau_j), (τ_j, tau_i)}
    B' := B' ∪ b
 sinon
  si s < p et τ_k ∈ act(I,t) alors
   rec(t, R, I, M, B', (a_1, .., a_{k−1}, 1, a_{k+1}, .., a_n), k + 1)
  si lax_k(I,t) > 0 alors
   rec(t, R, I, M, B', (a_1, .., a_{k−1}, 0, a_{k+1}, .., a_n), k + 1)
```

FIG. 8.3 – Algorithme (1) adapté pour PFI

```
t := 0
B := {(∅, (0, ..., 0), (−1, ..., −1))}
res := FALSE
tant que B ≠ ∅ ∧ res = FALSE faire
 pour tout k tel que t ∈ r_k + T.ℕ alors
  pour tout (R, I, M) ∈ B faire I_k := 0
 si t ∈ r + P.ℕ alors
  pour tout (R, I, M) ∈ B faire
   si I = M alors res := TRUE
   si M = (−1, ..., −1) alors M := I
 si res = FAUX alors
  B' := ∅
  pour tout (R, I, M) ∈ B faire
   rec(t, R, I, M, B', (0, ..., 0), 1)
  B := norm(B')
  t := t + 1
return res
```

FIG. 8.4 – Algorithme (2) adapté pour PFI

notons `norm(B)`.

L'algorithme de la section 7.5.3 est décomposé en deux procédures.

1. La première construit à partir d'une relation de priorité (et implicitement de la séquence d'exécution qu'elle engendre) toutes les suites (de longueur 1) de séquences possibles en ne conservant que celles respectant toutes les contraintes. Cette première procédure doit être modifiée pour effectuer la réinitialisation des relations de priorité.
2. La seconde constitue la procédure principale de l'algorithme. Elle implémente la récurrence : initialisation, passage de l'instant t à l'instant $t+1$, détection des cycles. Cette procédure aussi doit être modifiée pour intégrer le mécanisme de recherche des doublons.

La figure 8.3 indique l'algorithme (1) intégrant les modifications nécessaires pour PFI.

Dans cette section, nous étudions l'existence d'une configuration PFI valide. Or, l'algorithme de la section 7.5.3 détermine l'ensemble des configurations de priorité valides. Nous devons donc modifier l'implémentation de la procédure principale de la façon présentée ci-dessus, et également en changeant la condition de fin de l'algorithme. Dès qu'une configuration PFI valide est trouvée, l'algorithme doit se terminer. Cette implémentation ne peut pas fournir comme résultat la configuration en question puisque la relation correspondante ne contient que la description des priorités entre les instances courantes des tâches. La figure 8.4 indique l'algorithme (2) intégrant ces modifications.

Remarquons que dans notre formalisme (B et B' sont des ensembles), il est inutile de rechercher les doublons. Nous avons cependant indiqué cette opération dans la mesure où elle est importante dès que l'on fait une implémentation concrète de cet algorithme.

Etudions la terminaison de cet algorithme. Dans le cas où il n'existe pas de configuration de priorité valides, il existe un instant à partir duquel la base B est vide. Dans ce cas-ci, l'algorithme ci-dessus se termine donc toujours. Dans le cas contraire, la terminaison de l'algorithme dépend de l'existence d'une configuration de priorité engendrant une séquence cyclique et valide. Lorsque toutes les séquences valides sont acycliques, l'algorithme ci-dessus ne termine pas.

Toutefois, lors de nos expérimentations ce cas ne s'est jamais produit et cela nous paraît logique : il serait surprenant qu'il existe un système de tâches dont toutes les séquences d'exécution PFI valides soient acycliques. Ceci nous amène à conjecturer le point suivant : s'il existe une séquence d'exécution PFI valide, alors il existe aussi une séquence d'exécution PFI qui soit valide et cyclique de période P. Si cette conjecture est avérée, la terminaison de l'algorithme ci-dessus est toujours assurée. Remarquons finalement que EDF appartenant à $\overline{\text{PFI}}$ et étant optimale en monoprocesseur, la terminaison est toujours assurée dans ce contexte.

8.2.3 Evaluation de la puissance d'ordonnancement de PFI

En réutilisant la méthode d'expérimentation mise en place dans la section 7.6, nous avons évalué la puissance d'ordonnancement de PFI en la comparant à celle de PFX, de EDF et de LLF. Pour cela, nous avons utilisé l'algorithme de la section 8.2.2 qui détermine s'il existe une configuration PFI valide.

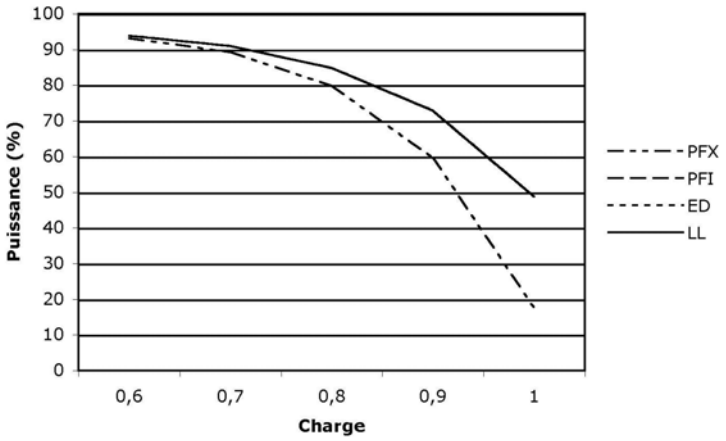
La figure 8.5 présente les résultats de nos expérimentations. La politique EDF appartient à PFI, ainsi les performances de PFI sont toujours supérieures à celles de EDF. Dans le cas monoprocesseur, la classe PFI est donc optimale : ainsi, les courbes de EDF, LLF et PFI coïncident. Dans le contexte biprocesseur, PFI conserve de bonnes performances. Plus important, il améliore significativement les performances obtenues avec PFX et EDF. L'utilisation des priorités fixes par instances permet donc d'augmenter la puissance d'ordonnancement par rapport aux politiques à priorités fixes sans recourir à celles à priorités dynamiques.

La figure 8.6 indique la durée moyenne de traitement d'un système de tâches par PFI et par PFX dans le cas monoprocesseur et dans le cas biprocesseur. On remarque que les temps de calcul de PFI sont du même ordre (seconde) que ceux de PFX. La décroissance du temps de calcul par rapport à la charge est un phénomène expliqué (voir chapitre 7). Par contre, l'augmentation du temps de calcul de PFI pour les systèmes monoprocesseurs (et pas pour ceux biprocesseurs !) ne l'est pas à ce jour et nécessite donc une étude supplémentaire. Il est aussi surprenant de constater

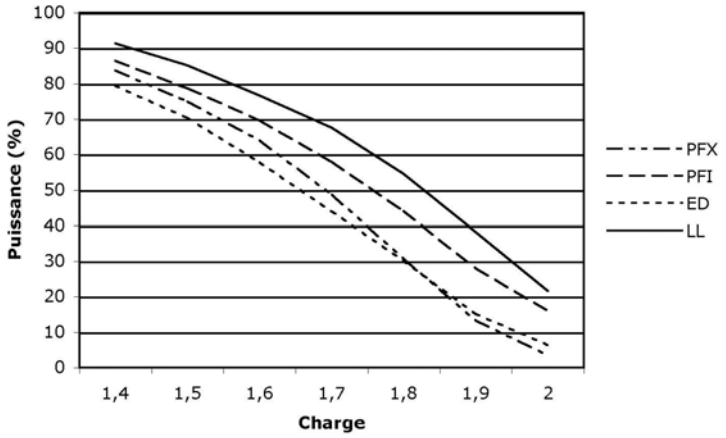
que le temps de calcul de PFI pour les systèmes biprocesseurs est généralement inférieur à celui pour les systèmes monoprocesseurs bien que ces derniers contiennent moins de tâches.

8.3 Domaine d'optimalité de $\overline{\overline{PFI}}$

Dans cette section, nous étudions les propriétés théoriques de la classe $\overline{\overline{PFI}}$. Nous énonçons tout d'abord une condition permettant de construire une politique $\overline{\overline{PFI}}$ produisant la même séquence d'exécution que EDF. Le procédé transformant EDF en une politique $\overline{\overline{PFI}}$ est utilisable en monoprocesseur et en multiprocesseur. Nous en déduisons ensuite un théorème d'optimalité pour $\overline{\overline{PFI}}$. Finalement, nous montrons par un exemple biprocesseur que la séquence produite par EDF n'est pas toujours productible par une politique de $\overline{\overline{PFI}}$.



Monoprocesseur



Biprocesseur

FIG. 8.5 – Puissance moyenne d'ordonnancement

8.3.1 Transformation de EDF en une politique $\overline{\overline{\text{PFI}}}$

Le théorème 3.1 montre que la politique EDF appartient à $\overline{\text{PFI}}$. Dans cette section, nous mettons en place un procédé permettant d'obtenir la séquence d'exécution produite par EDF en utilisant une politique de $\overline{\overline{\text{PFI}}}$. Toutefois, la validité de ce procédé dépend d'une condition. Le théorème suivant établit notre résultat.

Théorème 8.1 *Soient τ un système de tâches vérifiant $U \leq p$ et s la séquence d'exécution produite par EDF. S'il existe un instant $t \in \mathbb{N}$ postérieur à l'entrée dans le régime cyclique de s et vérifiant la condition donnée ci-dessous, alors la séquence s est aussi productible par une politique appartenant à $\overline{\overline{PFI}}$:*

$$\max\{e_s(\tau_{k,i}) | r_{k,i} < t \wedge e_s(\tau_{k,i}) > t\} \leq \min\{r_{k,i} | r_{k,i} \geq t\}$$

Démonstration :

D'après le théorème 3.1, la politique EDF appartient à $\overline{\text{PFI}}$. Etant donné que $U \leq p$, le corollaire 4.3 assure que la séquence s est cyclique.

Soient $t \geq r$ postérieur à l'entrée dans le régime cyclique, et π la politique d'ordonnancement attribuant les priorités suivantes :

$$\forall \tau_{k,i} \in \overline{\tau}, \pi(\tau_{k,i}) = \frac{1}{d_{k,i'}}$$

où $i' - i \bmod \delta_k = 0$ et $t \leq r_{k,i} < t + P$.

Autrement dit, la politique π attribue à une instance $\tau_{k,i}$ la priorité de l'instance équivalente qui est activée dans l'intervalle $[t, t+P[$. Remarquons que pour chaque instance $\tau_{k,i}$, l'instance $\tau_{k,i'}$ définie comme ci-dessus est unique. Par définition, la politique π appartient à $\overline{\overline{\text{PFI}}}$.

Posons E l'ensemble des instances qui ont été activées strictement avant t et qui sont encore actives à l'instant t.

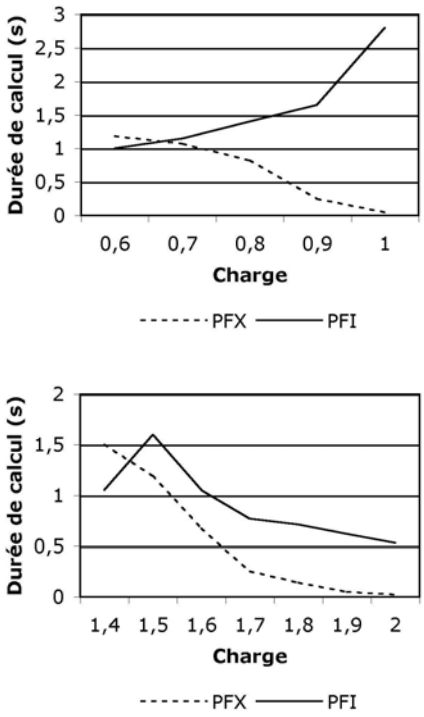


FIG. 8.6 – Temps de calcul moyen pour un système de tâches

Posons t' l'échéance la plus tardive des instances appartenant à E.

Par définition, entre t' et $t+P-1$, les priorités des instances actives attribuées par π sont les mêmes que celles attribuées par EDF.

Posons maintenant $\tau_{k',i'}$ la première instance activée après ou à l'instant t.

Par hypothèse, on a $r_{k',i'} \geq t'$. Ainsi, entre t et $t'-1$ les relations de priorité entre les instances actives induites par π sont les mêmes que celles induites par EDF.

Ainsi, pour toutes instances $\tau_{k,i}$ et $\tau_{k',i'}$ actives simultanément sur l'intervalle $[t, t+P[$, on a :

$$\pi(\tau_{k,i}) > \pi(\tau_{k',i'}) \Leftrightarrow EDF(\tau_{k,i}) > EDF(\tau_{k',i'})$$

Par définition, l'instant t est postérieur à l'entrée dans le régime cyclique, ainsi les temps de réponse des instances actives après t sont identiques à ceux des instances équivalentes des métapériodes suivantes. La propriété vérifiée à l'instant t est donc aussi vérifiée aux instants $t+P\mathbb{N}$. D'après le théorème 4.6, les temps de réponse ne peuvent pas décroître, il en est donc aussi de même pour les instants $t - P\mathbb{N}$.

Ainsi, à chaque instant, les relations de priorité entre les instances actives sont les mêmes pour EDF et pour π, ces deux politiques engendrent donc la même séquence d'exécution.

CQFD.

Remarquons que la condition suffisante utilisée dans le théorème 8.1 est aussi nécessaire dans le cas monoprocesseur, par contre elle ne l'est pas dans le cas multiprocesseur. La politique π utilisée dans la preuve du théorème 8.1 permet de "transformer" EDF en une politique $\overline{\overline{\text{PFI}}}$. Considérons deux instants t et t' postérieurs à r et distants d'un nombre entier de métapériodes. Alors, les relations de priorités entre la politique π_t et la politique $\pi_{t'}$ sont identiques :

$$\forall \tau_{k,i}, \tau_{k',i'} \in \overline{\tau}, \pi_t(\tau_{k,i}) > \pi_t(\tau_{k',i'}) \Leftrightarrow \pi_{t'}(\tau_{k,i}) > \pi_{t'}(\tau_{k',i'})$$

Ainsi, pour calculer la politique π, il n'est pas nécessaire de choisir un instant postérieur à l'entrée dans le régime cyclique, il suffit qu'il soit supérieur ou égal à r. D'autre part, les corollaires 4.3 et 4.4 fournissent une borne de la durée nécessaire pour atteindre le régime cyclique. Ainsi, en calculant cette borne, on peut déterminer si la politique π correspondant à un instant donné correspond bien à EDF en évaluant la condition utilisée dans le théorème 8.1. Pour les ordonnancements monoprocesseurs, on peut aussi utiliser les bornes fournies par [70, 21].

Le théorème suivant montre que notre procédé de transformation de EDF en une politique $\overline{\overline{\text{PFI}}}$ est utilisable dès qu'il existe un instant postérieur à l'entrée dans le régime cyclique auquel tous les processeurs sont oisifs.

Théorème 8.2 *Soient τ un système de tâches vérifiant $U \leq p$ et s la séquence d'exécution produite par EDF. S'il existe un instant $t \in \mathbb{N}$ postérieur à l'entrée dans le régime cyclique de s où tous les processeurs sont oisifs, alors la séquence s est aussi productible par une politique appartenant à $\overline{\overline{PFI}}$.*

Démonstration :

Soit $t \in \mathbb{N}$ vérifiant les conditions données ci-dessus. Sur l'intervalle $[t, t+1[$, il n'y a donc aucune instance en cours d'exécution. La politique EDF étant conservative, on en déduit qu'aucune instance n'est active à l'instant t, et donc qu'il n'y a aucune instance $\tau_{k,i} \in \overline{\tau}$ vérifiant :

$$r_{k,i} < t \wedge e_s(\tau_{k,i}) > t$$

En utilisant le théorème 8.1, on en déduit qu'il existe une politique appartenant à $\overline{\overline{\text{PFI}}}$ qui produit la séquence s.

CQFD.

Ainsi, en monoprocesseur, notre procédé de transformation de EDF en une politique $\overline{\overline{\text{PFI}}}$ est utilisable dès que $U < 1$. Par contre, en contexte multiprocesseur, il peut y avoir des temps creux sans que tous les processeurs soient oisifs. Ainsi, le simple fait que $U < p$ n'implique pas qu'il existe un instant où toutes les instances activées auparavant sont terminées.

8.3.2 Optimalité du contexte $\overline{\overline{\text{PFI}}}$ en monoprocesseur

Pour montrer que la classe $\overline{\overline{\text{PFI}}}$ est optimale en monoprocesseur, nous utilisons la séquence d'exécution produite par EDF. Pour les systèmes de tâches qui ne sont pas à pleine charge ($U < 1$), le régime cyclique des séquences d'exécution contient au moins un temps creux. Ainsi, le théorème 8.2 assure que l'on peut construire une politique $\overline{\overline{\text{PFI}}}$ correspondant à EDF. Avant de traiter le cas où $U = 1$, nous rappelons quelques résultats établis par [21] pour les ordonnancements produits par les politiques de DCL en contexte monoprocesseur et préemptif avec migration totale.

Les raisonnements menés dans [21] reposent sur l'étude des temps creux. La charge d'un système de tâches périodiques étant connue, on peut calculer le nombre de temps creux appartenant à chaque métapériode ($P - UP$). Ces temps creux sont appelés *cycliques*. Toutefois, [21] montre que d'autres temps creux peuvent apparaître dans la phase de montée en charge, ceux-ci sont appelés *acycliques*. La date du dernier temps creux acyclique est noté t_c, lorsqu'il n'y a pas de temps creux acyclique, on pose $t_c = -1$.

Théorème 8.3 *[21] Soit un système de tâches vérifiant $U = 1$. Le système est dans le même état aux instants $t_c + 1$ et $t_c + P + 1$.*

Pour établir ce résultat, [21] montre qu'à l'instant $t_c + 1$, ainsi qu'à l'instant $t_c + 1 + P$, les seules instances actives sont celles qui viennent juste de s'activer (s'il y en a). Cette propriété est intéressante pour nous puisqu'elle permet de trouver un instant où l'on peut définir la politique π - i.e. vérifiant la condition du théorème 8.1. Nous en déduisons donc le résultat suivant.

Théorème 8.4 *En contexte monoprocesseur et préemptif, la classe de politiques d'ordonnancement $\overline{\overline{PFI}}$ est optimale.*

Démonstration :

Pour montrer ce résultat, nous raisonnons sur la séquence s produite par EDF. Celle-ci étant optimale, on cherche à construire une politique appartenant à $\overline{\overline{\text{PFI}}}$ et produisant la même séquence d'exécution que EDF.

Lorsque la charge du système de tâches vérifie $U < 1$, le régime cyclique de s contient au moins un temps creux. Le théorème 8.2 assure alors qu'il existe une politique appartenant à $\overline{\overline{\text{PFI}}}$ qui produit la séquence s.

Supposons maintenant que $U = 1$. D'après le théorème 8.3, on sait qu'à l'instant $t_c + 1$, les seules instances actives sont celles qui viennent juste de s'activer. Ainsi, il n'existe aucune instance $\tau_{k,i} \in \overline{\tau}$ vérifiant :

$$r_{k,i} < t_c + 1 \wedge e_s(\tau_{k,i}) > t_c + 1$$

De plus, d'après le théorème 8.3, la séquence s est cyclique dès l'instant $t_c + 1$. Ainsi, le théorème 8.1 assure qu'il existe une politique appartenant à $\overline{\overline{\text{PFI}}}$ qui produit la séquence s.

CQFD.

Dans [21], les auteurs montrent aussi que les dates auxquelles se produisent les temps creux sont indépendantes de la politique d'ordonnancement DCL choisie. Ils montrent aussi que l'on peut déterminer ces dates à l'aide du diagramme de charge. En réalisant ce calul, les procédés utilisés dans les preuves des théorèmes 8.1 et 8.4 permettent de construire une politique $\overline{\overline{\text{PFI}}}$ à partir de la séquence produite par EDF pour tous les systèmes de tâches ordonnançables en contexte monoprocesseur et préemptif.

En environnement monoprocesseur, la classe $\overline{\overline{\text{PFI}}}$ est donc optimale. Par conséquent, le recours à des stratégies d'ordonnancement plus complexes (comme PFI, $\overline{\text{PFI}}$, DCL) n'est pas nécessaire. Par ailleurs, il existe des systèmes de tâches ordonnançables pour lesquels les politiques à priorités fixes ne permettent pas de produire une séquence valide. La classe $\overline{\overline{\text{PFI}}}$ est donc candidate pour être la plus petite classe d'ordonnancement optimale en monoprocesseur. Cette propriété la positionne comme un bon intermédiaire entre les politiques à priorités fixes et les politiques à priorités dynamiques.

8.3.3 Incompatibilité entre EDF et $\overline{\overline{\text{PFI}}}$ en multiprocesseur

Le théorème 8.2 suggère que la production d'une politique $\overline{\overline{\text{PFI}}}$ équivalente à EDF reste possible en multiprocesseur pour les systèmes de tâches ayant une charge pas trop élevée - i.e. il doit exister un instant appartenant au régime cyclique où tous les processeurs sont oisifs. Toutefois, l'exemple suivant montre qu'en multiprocesseur il n'est pas toujours possible de produire la séquence de EDF à l'aide d'une politique $\overline{\overline{\text{PFI}}}$. Il montre aussi que les classes $\overline{\text{PFI}}$ et $\overline{\overline{\text{PFI}}}$ sont distinctes puisque $EDF \in \overline{\text{PFI}}$ et $EDF \notin \overline{\overline{\text{PFI}}}$. On a donc l'inclusion stricte suivante :

$$\overline{\overline{\text{PFI}}} \subset \overline{\text{PFI}} \subset \text{PFI}$$

De plus, nous savons que PFI $\not\subset$ DCL et que DCL $\not\subset$ PFI. Par contre, nous ne savons pas comparer $\overline{\overline{\text{PFI}}}$ et $\overline{\text{PFI}}$ avec DCL. Cette étude nous semble intéressante pour mieux cerner les spécificités de ces classes de politiques d'ordonnancement.

Exemple 8.1 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k
τ_1	1	4	8	8
τ_2	0	3	8	8
τ_3	3	5	8	8
τ_4	2	3	8	8

La figure 8.7 présente la séquence d'exécution biprocesseur produite par EDF pour ce système de tâches. La séquence d'exécution devient cyclique à partir de l'instant $t = 15$. Puisqu'aucune échéance n'est manquée jusqu'à $t = 23$, EDF ordonnance donc ce système de tâches.

Remarquons que la condition du théorème 8.1 n'est jamais vérifiée, notre procédé de transformation de EDF en une politique $\overline{\overline{PFI}}$ ne peut donc pas être appliqué.

En outre, à l'instant $t = 3$, les instances $\tau_{1,0}$ et $\tau_{4,0}$ accèdent aux processeurs alors que l'instance $\tau_{3,0}$ est active, on a donc :

$$prio(\tau_{1,0}) > prio(\tau_{3,0}) \wedge prio(\tau_{4,0}) > prio(\tau_{3,0})$$

Or à l'instant $t = 9$, les instances $\tau_{2,1}$ et $\tau_{3,0}$ accèdent aux processeurs alors que l'instance $\tau_{1,1}$ est active, on a donc :

$$prio(\tau_{2,1}) > prio(\tau_{1,1}) \wedge prio(\tau_{3,0}) > prio(\tau_{1,1})$$

On en déduit donc que $prio(\tau_{1,0}) > prio(\tau_{1,1})$: ainsi cette séquence d'exécution ne peut pas être produite par une politique appartenant à $\overline{\overline{PFI}}$.

Remarquons que la durée de la montée en charge de la séquence présentée sur la figure 8.7 est strictement supérieure à $r + P = 12$. Ceci n'est pas dû au hasard : nous n'avons pas réussi à obtenir une séquence d'exécution valide produite par EDF qui ne soit pas productible par $\overline{\overline{\text{PFI}}}$ sans que la durée de montée en charge soit strictement supérieure à $r + P$.

Les configurations $\overline{\overline{\text{PFI}}}$ sont représentables à l'aide d'un nombre fini de niveaux de priorité. L'exemple donné ci-dessus montre que parfois la configuration de priorité de $\overline{\text{PFI}}$ correspondant à

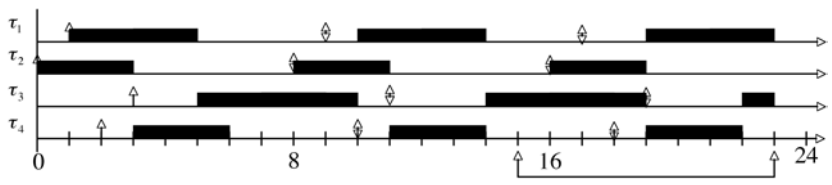


FIG. 8.7 – Ordonnancement biprocesseur par EDF du système de tâches de l'exemple 8.1

EDF ne peut pas être ramenée à un nombre fini de niveaux de priorité. La classe $\overline{\overline{\text{PFI}}}$ est donc candidate pour être la plus grande classe dont les politiques produisent des séquences représentables à l'aide d'un nombre fini de niveaux de priorité. Or, les exécutifs temps-réel offrent généralement un nombre de niveaux de priorité relativement restreint. La classe $\overline{\overline{\text{PFI}}}$ paraît donc particulièrement adaptée à leur architecture et devrait permettre d'exploiter pleinement leur puissance d'ordonnancement.

Ces différentes remarques nous amènent à la conjecture suivante, pour un système de tâches donné, on a :

la séquence produite par EDF est engendrable par une politique de $\overline{\overline{\text{PFI}}}$
$\Leftrightarrow$
la séquence produite par EDF est engendrable
à l'aide d'un nombre fini de niveaux de priorité fixe par instances
$\Leftrightarrow$
la durée de montée en charge de la séquence produite par EDF est inférieure ou égale à $r + P$

L'étude de la réalité ou non de ces équivalences nous semble une piste de recherche intéressante pour mieux cerner les propriétés des ordonnancements multiprocesseurs.

8.4 Ordonnancements $\overline{\overline{\text{PFI}}}$ valides

Contrairement aux cas des priorités fixes, le nombre de configurations PFI valides est potentiellement infini. La recherche et l'énumération de ces configurations est donc impossible. Ce constat est aussi valable pour les configurations $\overline{\text{PFI}}$. Par ailleurs, la classe $\overline{\overline{\text{PFI}}}$ présente des propriétés intéressantes. Dans cette section, nous nous restreignons donc à elle et nous étudions plusieurs méthodes permettant de calculer des configurations de priorité valides.

La classe $\overline{\overline{\text{PFI}}}$ correspond aux configurations de priorité qui attribuent la même priorité à deux instances équivalentes - i.e. appartenant à la même tâche et distantes d'exactement une métapériode. Cette restriction paraît très raisonnable en terme d'ordonnancement et aussi en terme d'exploitation : les exécutifs temps-réel utilisant des priorités fixes peuvent être adaptés aux configurations $\overline{\overline{\text{PFI}}}$.

8.4.1 Algorithme de résolution

Dans la section 8.2.1, nous avons représenté une configuration PFI à l'aide d'une configuration PFX en utilisant un système de tâches intermédiaire $\tau^\star$. Pour les configurations de $\overline{\overline{\text{PFI}}}$, le système $\tau^\star$ peut être simplifié. Pour chaque tâche $\tau_k \in \tau$, on introduit les tâches suivantes dans $\tau^\#$:

$$\forall i \in \{1, \ldots, \delta_k\}, (r_{k,i}, C_k, D_k, P) \in \tau^\#$$

Ce nouveau système de tâches contient un nombre fini de tâches périodiques, il correspond donc exactement au contexte d'étude de la section 7, et les résultats qui y sont développés peuvent être appliqués directement à $\tau^\#$. Dans ce contexte, on se ramène donc à un système de $\sum_{i=1}^{n} P/T_k$ tâches périodiques. L'algorithme donné dans la section 7.5.3 détermine l'ensemble des configurations de priorité fixe valides. En l'appliquant au système $\tau^\#$, on détermine donc les configurations de priorité fixe par instances de $\overline{\overline{\text{PFI}}}$ qui ordonnancent τ.

D'après l'étude menée dans le chapitre 4, les séquences engendrées par des configurations de $\overline{\overline{\text{PFI}}}$ sont cycliques de période P. Le contexte $\overline{\overline{\text{PFI}}}$ permet donc d'éviter les anomalies que nous avons rencontrées dans le contexte général PFI (voir chapitre 3). De plus, en utilisant les résultats que nous avons obtenus dans le chapitre 5, on peut majorer la durée de montée en charge des séquences engendrées par $\overline{\overline{\text{PFI}}}$.

8.4.2 Hybridation PFX/$\overline{\overline{\text{PFI}}}$

L'algorithme de calcul des configurations $\overline{\overline{\text{PFI}}}$ valides a une complexité élevée puisque l'on doit considérer chaque instance comme une tâche indépendante. Dans cette section, nous proposons une méthode incrémentale pour trouver une configuration $\overline{\overline{\text{PFI}}}$ valide.

Au préalable, on vérifie qu'il existe une configuration PFI valide grâce à l'algorithme de la section 8.2.2. Ensuite, on passe progressivement de PFX à $\overline{\overline{\text{PFI}}}$ en permettant à certaines instances d'une même tâche d'avoir des priorités différentes. Pour cela, on procède de la manière suivante :

1. on détermine s'il existe une configuration PFX valide,
2. si oui, alors les configurations de priorité fournies par l'algorithme PFX constituent le résultat,
3. sinon, on choisit une tâche τ_k pour permettre à ses instances d'avoir des priorités différentes : on la décompose en δ_k tâches ayant les paramètres temporels des instances de τ_k.

Cette approche permet de profiter progressivement et seulement en cas de besoin de la puissance d'ordonnancement de $\overline{\overline{\text{PFI}}}$. Plusieurs règles peuvent être utilisées pour traiter l'étape (3). Tout d'abord, il faut déterminer un mécanisme pour choisir la tâche à décomposer. Ensuite, il faut choisir les instances auxquelles on permet d'avoir une priorité propre. Remarquons qu'en modifiant légèrement l'algorithme de la section 7.5.3, on peut permettre à seulement certaines instances d'une même tâche d'avoir des priorités différentes. On peut le réaliser en introduisant la notion de groupe d'instances : les instances d'un même groupe partageant la même priorité et ne pouvant jamais être actives simultanément. Cette extension permet de mieux maîtriser la complexité lors du passage de PFX à $\overline{\overline{\text{PFI}}}$.

Donnons un exemple de règle permettant d'implémenter l'étape (3). Pour cela, considérons une relation R appartenant à la base minimale correspondant à l'instant t, et supposons qu'aucune action ne permet de prolonger $s(R, p, t)$ en une séquence $(Q, t+1)$-valide. Lorsque les tâches sont indépendantes, on peut alors affirmer qu'il existe plus de p tâches dont la laxité est nulle (voir section 7.5.1). Ces p tâches sont à l'origine de l'échec d'ordonnancement par la relation R. On peut donc associer à chaque relation non prolongeable les tâches qui sont responsables de l'échec. Lors du traitement par PFX, on peut alors déterminer le nombre de fois où chaque tâche est responsable d'un échec d'ordonnancement. Plus précisément, on peut même réaliser ce calcul, non pas par tâches, mais par instances. On obtient ainsi une description des tâches (ou instances) à l'origine des échecs d'ordonnancement. On peut alors choisir de décomposer la tâche qui y apparaît le plus souvent.

Cette hybridation PFX/$\overline{\overline{\text{PFI}}}$ paraît prometteuse pour plusieurs raisons. Tout d'abord, elle permet d'exploiter la puissance d'ordonnancement de $\overline{\overline{\text{PFI}}}$ tout en limitant la complexité. Ensuite, de nombreuses règles peuvent être élaborées pour déterminer la ou les tâches à décomposer : elle admet donc une certaine flexibilité. Finalement, le concepteur peut intervenir lors des choix à effectuer pour guider l'algorithme de recherche : elle permet donc aussi une certaine interactivité.

8.5 Conclusion

Afin d'obtenir une puissance d'ordonnancement plus importante que celle fournie par les politiques à priorités fixes, nous avons étudié la classe PFI. Nous avons adapté l'algorithme donné dans le chapitre 7 pour qu'il détermine s'il existe une configuration de priorité fixe par instances valides. Ce nouvel algorithme nous a permis d'évaluer expérimentalement la puissance d'ordonnancement de PFI. Celle-ci s'est révélée être bien supérieure à celle de PFX et de EDF.

La classe PFI paraissant intéressante, nous avons approfondi notre étude. Nous avons alors élaboré un mécanisme permettant de construire, sous certains conditions, une politique $\overline{\overline{\text{PFI}}}$ produisant la même séquence que EDF. Nous avons ensuite montré que la classe $\overline{\overline{\text{PFI}}}$ est optimale en contexte monoprocesseur et préemptif. Cette classe semble aussi posséder d'autres propriétés intéressantes :

- $\overline{\overline{\text{PFI}}}$ est la plus petite classe optimale en monoprocesseur,

– $\overline{\overline{\text{PFI}}}$ est la plus grande classe dont les politiques engendrent des séquences représentables à l'aide d'un nombre fini de niveaux de priorité fixe attribués aux instances.

Certains problèmes théoriques restent donc en suspens. La comparaison entre les trois classes de politiques à priorités fixes par instances (PFI, $\overline{\text{PFI}}$, $\overline{\overline{\text{PFI}}}$) permettrait de mieux cerner les possibilités apportées par chacune.

L'utilisation d'une politique $\overline{\overline{\text{PFI}}}$ pour produire la séquence d'exécution de EDF présente plusieurs avantages. Les priorités attribuées aux instances par les politiques $\overline{\overline{\text{PFI}}}$ sont constantes, la politique $\overline{\overline{\text{PFI}}}$ équivalente à EDF est donc plus facilement implémentable que EDF. Les protocoles de gestion de ressources peuvent aussi lui être plus facilement adaptés que pour EDF. En termes de prédictabilité, il est aussi plus facile d'obtenir des majorations des temps de réponse des instances avec les politiques $\overline{\overline{\text{PFI}}}$ qu'avec EDF.

La classe $\overline{\overline{\text{PFI}}}$ paraît particulièrement adaptée aux exécutifs temps-réel actuels. Nous avons alors étendu la méthode développée dans le chapitre 7 pour déterminer les configurations de priorité valides qui appartiennent à $\overline{\overline{\text{PFI}}}$. Pour diminuer la complexité, nous avons aussi proposé une méthode incrémentale pour construire une telle configuration de priorité. Cette approche nous paraît prometteuse puisqu'elle permet de maîtriser la complexité tout en offrant une certaine modularité et une certaine interactivité. Effectivement, on peut imaginer de nombreux procédés pour passer progressivement d'une configuration PFX à une configuration $\overline{\overline{\text{PFI}}}$. Leur élaboration et leur évaluation semble être une voie intéressante de recherche. D'autre part, on peut aussi chercher à construire des configurations $\overline{\overline{\text{PFI}}}$ engendrant des séquences d'exécution produites par d'autres méthodes (politiques en-ligne, méthodes hors-ligne, etc). Cette transformation simplifierait l'intégration de ces séquences dans les exécutifs temps-réel, notamment lorsqu'elles sont produites par une méthode hors-ligne. En utilisant le système $\tau^{\#}$ (voir section 8.4.1) et en appliquant la méthode fournie dans la section 7.3.3, on peut effectuer cette transformation. D'autres méthodes plus spécifiques, comme celle que nous avons utilisée pour transformer EDF dans la section 8.3.1, peuvent aussi être utilisées.

Chapitre 9

Systèmes de tâches avec relations de précédence

9.1 Introduction

Dans ce chapitre, nous abordons le problème de l'ordonnancement selon un angle différent. Dans les deux chapitres précédents, nous avons repris l'approche suivie par [9] : l'élaboration de méthodes exactes attribuant des paramètres d'ordonnancement en-ligne, comme la priorité des tâches par exemple. Nous suivons ici l'une des autres approches possibles pour traiter un problème NP : la recherche en temps polynomial de solution approchée. Afin d'initier notre approche, nous avons repris des travaux existants ainsi que les problématiques qu'ils traitent.

L'utilisation des architectures multiprocesseurs a introduit de nouveaux problèmes dans le domaine de l'ordonnancement. L'un d'eux consiste à ordonnancer un ensemble de tâches liées par des relations de précédence tout en minimisant la durée totale d'exécution. Ce problème est connu pour être dans la classe NP-dur. Toutes les méthodes exactes ont donc une complexité exponentielle. L'utilisation des méthodes d'approximation (algorithmes génétiques, méthode taboue, processus de décision markovien, etc) permet d'obtenir une solution approchée (sous-optimale) en un temps polynomial. Ces différentes méthodes ont déjà été utilisées avec succès, par exemple pour le problème de l'allocation de fréquence [53, 54] et pour celui de la planification de trajectoires [65].

Pour le problème de l'ordonnancement multiprocesseur d'un système de tâches liées par des relations de précédence, plusieurs méthodes d'approximation basées sur les algorithmes génétiques ont été développées pour obtenir une solution approchée, par exemple [23, 52, 87, 108]. Dans ce chapitre, nous proposons tout d'abord plusieurs améliorations de l'algorithme génétique défini par [52]. Ensuite, nous développons une nouvelle approche basée sur la descente de gradient et sur la méthode taboue. Finalement, nous comparons expérimentalement les différentes méthodes que nous proposons avec celles définies par [23, 52].

9.2 Contexte d'étude

Nous utilisons le modèle proposé par [52] pour représenter les systèmes de tâches liés par des contraintes de précédence (voir figure 9.1). Un ensemble de tâches partiellement ordonnées est représenté par un graphe orienté et acyclique (DAG) constitué d'un ensemble fini de nœuds V et d'un ensemble fini d'arcs E. Chaque noeud $\tau_i \in V$ correspond à une tâche et est associé à sa durée d'exécution C_i. Chaque arc $e_{ij} \in E$ correspond à une relation de précédence $\tau_i \succ \tau_j$: la tâche τ_i devant être terminée avant que la tâche τ_j ne puisse commencer.

Le problème de l'ordonnancement de ces systèmes de tâches consiste à trouver une séquence d'exécution respectant les contraintes de précédence posées et minimisant la durée d'exécution totale. La figure 9.2 donne un exemple d'ordonannncement du système défini dans la figure 9.1.

9.3 Algorithmes génétiques appliqués à l'ordonnancement

Les algorithmes génétiques (GA) permettent d'obtenir une solution approchée d'un problème posé. [41] présente en détail les principes de fonctionnement propres à ces algorithmes. Nous en rappelons ici les concepts de base. Les algorithmes génétiques consistent à améliorer successivement un ensemble de solutions relativement à une *fonction de coût*. Cette fonction représente l'objectif à atteindre. Chaque solution est appelée un *individu*, et l'ensemble des solutions forme la *population*. Le codage des solutions, et donc la représentation des individus, est organisé en plusieurs *chromosomes* eux-mêmes divisés en plusieurs *gènes*. Le choix du codage et de la fonction de coût sont des éléments importants lors de la conception d'un algorithme génétique puisqu'ils fixent la modélisation du problème. Pour faire évoluer la population, les algorithmes génétiques sont munis de trois opérateurs :

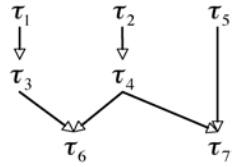
- le *croisement* : en combinant les gènes appartenant à deux solutions différentes, on construit une nouvelle solution,
- la *mutation* : on modifie les gènes d'une solution afin d'en produire une nouvelle,
- la *sélection* : on choisit parmi les solutions courantes celles qui formeront la population suivante.

Les opérateurs de croisement et de mutation sont complémentaires. En général, on utilise l'un d'eux avec une forte probabilité pour apporter des améliorations importantes, et l'autre avec une faible probabilité pour sortir des minima locaux. En fonction des problèmes, ces deux opérateurs remplissent l'un ou l'autre rôle.

Les algorithmes génétiques ont déjà été appliqués au problème de l'ordonnancement multi-processeur. Dans notre étude, nous réutilisons les travaux de [52] et de [23]. Nous présentons maintenant les méthodes proposées dans ces deux articles. Nous ne revenons pas plus en détail sur les principes généraux, et renvoyons au besoin le lecteur vers [41].

9.3.1 L'algorithme génétique proposé par [52]

Le premier élément à définir pour mettre en place un algorithme génétique est le codage des solutions. Il faut alors répondre aux deux questions suivantes : que représente un chromosome, et que représente un gène. Dans [52], chaque processeur est modélisé par un chromosome, et ses gènes correspondent aux tâches exécutées sur ce processeur. D'autres codages ont aussi été proposés [1]. La figure 9.3 présente le codage de [52] pour la séquence d'exécution donnée dans la figure 9.2.



Contraintes de précédence

τ_1	τ_2	τ_3	τ_4
2	5	4	3

τ_5	τ_6	τ_7
3	3	4

Durée d'exécution

FIG. 9.1 – Un exemple de système de tâches

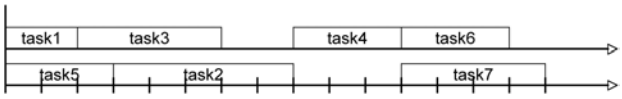


FIG. 9.2 – Un exemple de séquence d'exécution

$P1$	τ_1	τ_3	τ_4	τ_6
$P2$	τ_5	τ_2	τ_7	

FIG. 9.3 – Codage d'un ordonnancement

Un tel codage ne représente pas toujours une séquence d'exécution faisable, prenons un exemple pour illustrer ce point :

$P1$	τ_3	τ_2	$\cdots$
$P2$	τ_4	τ_1	$\cdots$

D'une part, la tâche τ_3 est précédée par τ_1, le processeur P1 est donc bloqué jusqu'à la terminaison de τ_1. D'autre part, P2 est bloqué par τ_4 jusqu'à la terminaison de τ_2. Les deux processeurs sont donc bloqués et cette sequence d'exécution n'est pas faisable.

Pour se garantir contre les interblocages de ce type, [52] introduit la notion de hauteur d'une tâche. Si une tâche τ_i précède une autre tâche τ_j, alors on doit avoir $height(\tau_i) < height(\tau_j)$. Cette fonction de hauteur conserve les relations de précédence entre les tâches. [52] propose plusieurs définitions de la hauteur d'une tâche, toutes vérifiant la propriété précédente. La plus simple d'entre elles consiste à poser :

$$\begin{cases} height(\tau_i) = 0, \text{ si } \{\tau_j | \tau_j \succ \tau_i\} = \emptyset \\ height(\tau_i) = 1 + \max_{\tau_j \succ \tau_i} height(\tau_j), \text{ sinon} \end{cases}$$

Ainsi, tous les codages pour lesquels les tâches sont rangées sur chaque processeur par ordre croissant de hauteur sont faisables.

Le second élément à définir est la fonction de coût représentant la propriété que l'on souhaite optimiser. Pour l'ordonnancement multiprocesseur, on désire minimiser la durée d'exécution requise pour exécuter toutes les tâches. La date de terminaison T_i d'une tâche τ_i dépend de celle de tous ses prédécesseurs, posons :

$$\Delta = max\{T_j | \tau_j \succ \tau_i\}$$

Elle dépend aussi de la date de terminaison des tâches qui sont associées au même processeur que τ_i. Plus précisément, elle dépend de celle qui la précède directement. Notons δ, la date de terminaison de cette tâche. La date de terminaison de la tâche τ_i est alors donnée par la formule suivante :

$$T_i = C_i + \max\{\delta, \Delta\}$$

La fonction de coût utilisée dans [52] est $\max\{T_i\}$, elle sert donc à minimiser la durée d'exécution totale. Les opérateurs génétiques utilisés dans [52] pour produire de nouvelles solutions sont classiques :

croisement	aléatoire à un point
mutation	permutation aléatoire de deux tâches
sélection	roulette aléatoire

Précisons seulement que leurs opérateurs sont adaptés pour que les tâches attribuées à un même processeur restent rangées par ordre croissant de hauteur. Dans la suite, nous désignons par HAR l'algorithme génétique défini par [52].

9.3.2 Les algorithmes génétiques proposés par [23]

Pour assurer qu'un codage correspond à une séquence d'exécution faisable, [52] a introduit la notion de hauteur. Nous avons vu que si les tâches sont rangées par ordre croissant de hauteur sur chaque processeur, alors la séquence d'exécution correspondante est faisable. Cependant, comme l'a fait remarquer [23], la réciproque n'est pas vraie en général. Ainsi, cette méthode ne permet pas de représenter toutes les solutions possibles. Pour corriger cet inconvénient, [23] a proposé une autre approche.

Présentons maintenant la notion utilisée par [23] pour assurer qu'un codage est faisable. Pour cela, reprenons l'exemple de la figure 9.3. Dans ce codage, les tâches exécutées sur le processeur P1 sont dans l'ordre : $\tau_1, \tau_3, \tau_4, \tau_6$. Cette affectation introduit de nouvelles relations de précédence : $\tau_1 \succ \tau_3$, $\tau_3 \succ \tau_4$, $\tau_4 \succ \tau_6$. Pour déterminer si un codage est faisable, il faut aussi tenir compte des précédences induites par le placement des tâches sur les processeurs : le graphe composé des relations de précédence entre les tâches et de celles engendrées par le placement des tâches sur

les processeurs doit être acyclique. La vérification de cette propriété amène à utiliser la clôture transitive de ce graphe. Comme le remarque [23], ceci conduit à une complexité plus importante qu'avec l'utilisation de la hauteur. L'exhaustivité a un coût important en terme de complexité algorithmique.

Le premier algorithme génétique proposé par [23] utilise les mêmes opérateurs génétiques que [52], toutefois, en les ayant préalablement adaptés à leur critère de faisabilité. Le second utilise un algorithme de liste appelé ED/MISF (earliest date/most immediate successors first). L'utilisation de ED/MISF dans l'opérateur de croisement et de mutation permet de guider rapidement l'algorithme génétique vers de bonnes solutions.

Dans la suite, nous désignons par FSG (resp. CGL) le premier (resp. second) algorithme génétique défini par [23].

9.4 Contributions

Dans cette section, nous présentons les modifications que nous proposons pour l'algorithme génétique HAR. Ensuite, nous développons une nouvelle approche à partir de la descente de gradient et de la méthode taboue.

9.4.1 Améliorations du GA proposé par [52]

[52] a proposé un algorithme génétique pour résoudre le problème de l'ordonnancement multiprocesseurs. Comme il est indiqué dans [23], leur méthode présente des défauts. Dans cette section, nous présentons deux améliorations qui permettent de répondre à ces difficultés. Nous avons préféré nous baser sur la méthode proposée par [52] plutôt que sur celle proposée par [23] pour éviter l'augmentation de la complexité qu'elle engendre.

La méthode utilisée par [52] pour générer la population initiale attribue les tâches aux processeurs de manière incrémentale. Posons $T(h)$ l'ensemble des tâches de hauteur h et max_h la profondeur du graphe de précédence. Le premier processeur se voit attribuer un nombre aléatoire n_h de tâches pour chacune des hauteurs h. Ce nombre est choisi uniformément dans l'intervalle $[0, h]$. Ainsi, en moyenne, le nombre de tâches attribuées au premier processeur est :

$$\frac{n_1}{2} + \frac{n_2}{2} + \ldots + \frac{n_{max_h}}{2} = \frac{n}{2}$$

Pour les processeurs suivants, ce processus est répété avec l'ensemble des tâches qui n'ont pas encore été attribuées. Le dernier processeur se voit attribuer l'ensemble des tâches restantes. Le nombre de tâches attribuées au $k^{\grave{e}}$ processeur est donc en moyenne $n/2^k$. Les tâches ne sont donc pas réparties uniformément entre les processeurs. Ce fait a un impact véritable sur les performances (voir la figure 9.6) : lorsque le nombre de processeurs devient grand, les performances se dégradent très vite.

Pour améliorer la qualité de la population initiale, nous proposons deux schémas de répartition :

1. Répartition uniforme du nombre de tâches attribuées à chaque processeur : nous bornons le nombre total de tâches attribuables à un même processeur par $\lceil n/p \rceil$.
2. Répartition uniforme de la charge attribuée à chaque processeur : nous bornons la charge maximale attribuable à un même processeur par $\lceil \frac{\sum_{k=1}^{n} C_k}{p} \rceil$.

Ces modifications permettent d'améliorer la qualité des solutions obtenues lorsque le nombre de processeurs devient grand. Dans la suite, nous appelons GAN la variante de l'algorithme HAR obtenue en utilisant le premier schéma de répartition, et GAU celle obtenue en utilisant le second.

9.4.2 Descente de gradient

Dans cette section, nous présentons une méthode pour approximer le problème de l'ordonnancement multiprocesseur basée sur la descente de gradient. Pour représenter les solutions, nous

conservons le modèle utilisé par [52]. La fonction de coût à minimiser est aussi définie de la même manière : elle correspond à la durée nécessaire pour exécuter toutes les tâches selon l'ordre indiqué.

La méthode de descente de gradient repose sur l'amélioration successive d'une solution. Pour cela, on évalue les solutions voisines en ne retenant que la meilleure. Lorsque l'algorithme ne peut plus améliorer la solution, celle-ci correspond alors à un minimum local, l'algorithme la retourne comme résultat. L'un des éléments clé d'une descente de gradient se situe dans la définition du voisinage d'une solution : celui-ci doit être le plus petit possible (pour que la méthode soit performante en terme de temps d'exécution) tout en permettant d'explorer la plus grande partie possible de l'espace des solutions (pour que la solution fournie soit de bonne qualité).

Considérons la séquence d'exécution indiquée sur la figure 9.2. Pour la transformer en une solution voisine, on considère deux types de modifications :

1. la permutation de deux tâches adjacentes attribuées à un même processeur (voir la figure 9.4),
2. le changement du processeur attribué à une tâche (voir la figure 9.5).

Dans la suite, nous appelons DG la descente de gradient correspondante.

9.4.3 Méthode taboue

La méthode taboue est une descente de gradient munie de mécanismes pour échapper aux minima locaux. Nous en avons expérimenté deux qui reposent sur la descente de gradient présentée dans la section 9.4.2. Les mécanismes permettant de sortir des minima locaux reposent sur l'interdiction de certaines solutions ou de certaines caractéristiques. Par exemple, lorsque la solution locale est un minimum local, il doit être possible de dégrader la qualité de la solution courante, et aussi d'interdire le retour vers ce minimum local pour un certain temps.

La première implémentation (appelée TBS) de la méthode taboue que nous avons réalisée consiste à interdire définitivement les minima locaux que l'algorithme a déjà atteints. La seconde (appelée TBC) repose sur l'interdiction de certaines caractéristiques des solutions.

Dans la section 9.4.2, nous avons défini le voisinage d'une solution à l'aide de deux types de modifications. A chacune d'elles, nous allons associer une caractéristique taboue spécifique :

1. pour un processeur attribué à k tâches, il y a $k-1$ permutations possibles entre tâches adjacentes que nous identifions par leur position. Lorsque l'algorithme sort d'un minimum local à l'aide de cette modification, la permutation utilisée devient taboue.

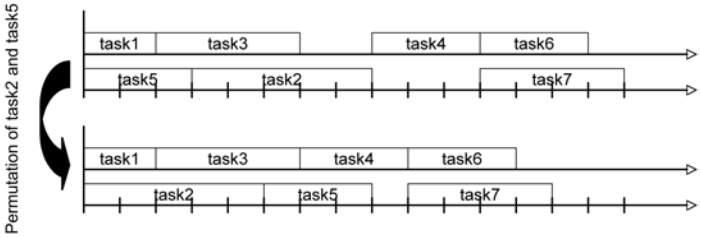


FIG. 9.4 – Permutation de deux tâches adjacentes

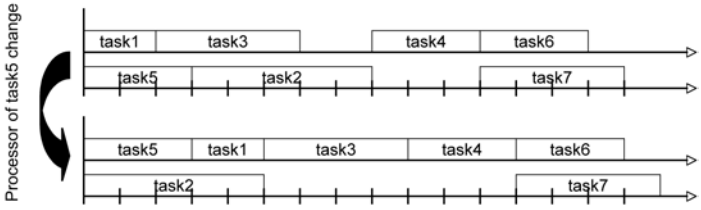


FIG. 9.5 – Changement du processeur attribué à une tâche

2. Lorsque l'algorithme sort d'un minimum local en changeant le processeur attribué à une tâche, l'ancienne attribution devient taboue : cette tâche ne peut pas être réattribuée à son ancien processeur.

Lorsque l'on interdit certaines caractéristiques, il n'est pas intéressant de les interdire définitivement. Pour déterminer le nombre d'itérations où une caractéristique doit être interdite, nous nous sommes inspirés de la complexité du voisinage d'une solution. Dans la section suivante, nous déterminons cette grandeur.

9.4.4 Etude de la complexité du voisinage d'une solution

Les deux modifications que l'on peut appliquer à une solution définissent le voisinage de cette solution. Dans cette section, nous dénombrons le nombre de voisins d'une solution. Pour cela, nous supposons que *chaque processeur est attribué à au moins une tâche.*

Dans la suite, on note $\sigma(i)$ le nombre de tâches attribuées au $i^{\text{è}}$ processeur.

Pour le $i^{\text{è}}$ processeur, il y $\sigma(i)-1$ permutations de tâches possibles, dès que chaque processeur est attribué à au moins une tâche. Pour l'ensemble des processeurs, on obtient donc V_1 voisins, dès que toutes les tâches sont attribuées à exactement un processeur :

$$V_1 = \sum_{k=1}^{p}(\sigma(i)-1) = \left(\sum_{k=1}^{p}\sigma(i)\right) - p = n - p$$

Considérons maintenant la modification consistant à changer le processeur attribué à une tâche. Pour une tâche j donnée et attribuée au processeur i, on peut l'attribuer à $p-1$ autres processeurs. Supposons qu'elle soit attribuée au $k^{\text{è}}$, il y a alors $\sigma(k)+1$ positions possibles pour l'insérer. Pour les $p-1$ processeurs possibles, on obtient donc V'_j voisins possibles :

$$V'_j = \sum_{k=1,k\neq j}^{p}(\sigma(k)+1) = n+p-1-\sigma(i)$$

Pour l'ensemble des tâches associées au $i^{\text{è}}$ processeur, on obtient donc $V_i^{\star}$ voisins :

$$V_i^{\star} = \sigma(i)(n+p-1-\sigma(i))$$

La deuxième modification d'une solution engendre donc V_2 voisins :

$$\begin{aligned} V_2 &= \sum_{k=1}^{p}\sigma(k)(n+p-1-\sigma(k)) \\ &= (n+p-1)\sum_{k=1}^{p}\sigma(k) - \sum_{k=1}^{p}\sigma(k)^2 \\ &= n(n+p-1) - \sum_{k=1}^{p}\sigma(k)^2 \end{aligned}$$

Finalement, dès qu'un processeur est attribué à au moins une tâche et que toutes les tâches sont attribuées à au moins un processeur, le nombre de voisins V d'une solution est :

$$V = V_1 + V_2 = n(n+p) - p - \sum_{k=1}^{p}\sigma(k)^2$$

L'expression ci-dessus dépend de la solution courante. En utilisant le fait que $\sum_{k=1}^{p}\sigma(k)^2 \leq n^2$ puisque l'on a $\sum_{k=1}^{p}\sigma(k) = n$, on obtient un encadrement indépendant de la solution courante :

$$(n-1)p \leq V \leq n(n+p) - p$$

Pour implémenter une méthode taboue, il faut préciser le nombre d'itérations pour lesquelles une caractéristique est taboue. Lors de nos expérimentations, nous avons remarqué que cette valeur dépendait aussi bien du nombre de tâches que du nombre de processeurs. L'encadrement obtenu pour V nous a incités à fixer cette valeur à np.

9.5 Expérimentations

Pour comparer les différentes méthodes d'ordonnancement, nous avons généré aléatoirement des systèmes composés de 50 tâches. Pour étudier l'impact du nombre de relations de précédence, nous avons mené trois campagnes de test : les systèmes contiennent 25, 50, ou 100 relations de précédence. Pour chacune de ces campagnes, nous avons fait varier le nombre de processeurs de l'architecture destinée à héberger le système de tâches : chaque échantillon contient 100 systèmes de tâches.

L'indicateur que nous avons utilisé pour évaluer la qualité des solutions fournies par une méthode est le rapport entre la durée minimale théorique pour exécuter toutes les tâches composant un système et la durée d'exécution de la solution fournie. Pour définir la durée minimale théorique, nous calculons deux minorants de la durée d'exécution d'un système de tâches :

- le premier est défini par un fonctionnement à plein régime des processeurs disponibles (sans tenir compte des relations de précédence) :

$$\left\lfloor \frac{\sum_{k=1}^{n} C_k}{p} \right\rfloor$$

- le second est défini par la durée d'exécution de la plus longue chaîne de précédence

La durée minimale théorique est définie comme la plus grande valeur des deux quantités définies ci-dessus.

La figure 9.6 présente la qualité des solutions fournies par chacune des méthodes par rapport à la durée d'exécution minimale. Chacun des points de ces graphiques correspond à la moyenne de la qualité de la solution obtenue pour chacun des 100 systèmes de tâches composant un échantillon. La figure 9.7 présente le temps de calcul moyen de chaque méthode d'ordonnancement.

Les variantes de HAR que nous avons définies augmentent considérablement les performances de cet algorithme génétique, surtout lorsque le nombre de processeurs augmente. D'autre part, GAU domine GAN et HAR : il est donc préférable de répartir uniformément la charge par processeur, plutôt que le nombre de tâches par processeur. Nos méthodes de génération de la population initiale permettent pratiquement d'élever le niveau de HAR à celui de FSG, tout en admettant un temps de calcul beaucoup plus court.

Notre méthode basée sur la descente de gradient donne de meilleurs résultats que HAR, GAN, GAU et FSG. Ses temps de calcul sont plus longs que ceux de HAR, GAN et GAU, mais plus courts que ceux de FSG. L'algorithme DG est donc un compromis intéressant entre la durée de calcul et la qualité des solutions fournies.

Les trois méthodes CGL, TBS et TBC donnent des solutions de bonne qualité dont la durée d'exécution est proche de la valeur minimale théorique. Toutefois, TBC domine toujours les deux autres et est généralement au-dessus de 90% d'optimalité. Les temps de calcul de TBS et TBC sont bien inférieurs à ceux de CGL lorsque le nombre de processeurs reste "faible". Dans les autres cas, on remarque que les temps de calcul de ces trois méthodes sont relativement similaires.

La figure 9.8 présente la variance obtenue pour 100 exécutions de chacune des méthodes sur un système composé de 50 tâches, 50 relations de précédence, et destiné à être hébergé sur une architecture composée de 5 processeurs. La variance est un indicateur de qualité d'une méthode, elle "mesure" sa stabilité : en utilisant deux fois la même méthode sur un même problème, obtient-on des solutions similaires ? Pour HAR, GAN, GAU et DG, la réponse est non, on pourra certaines fois obtenir de bonnes solutions et d'autres fois des solutions mauvaises. Il est alors impératif pour exploiter en pratique ces méthodes de les appliquer un grand nombre de fois, et de ne retenir que la meilleure solution trouvée. Pour TBS, TBC, FSG et CGL, la variance est beaucoup plus faible. Plusieurs utilisations de ces méthodes à un même problème fournissent donc des solutions proches les unes des autres, on peut alors se contenter de les appliquer une seule fois.

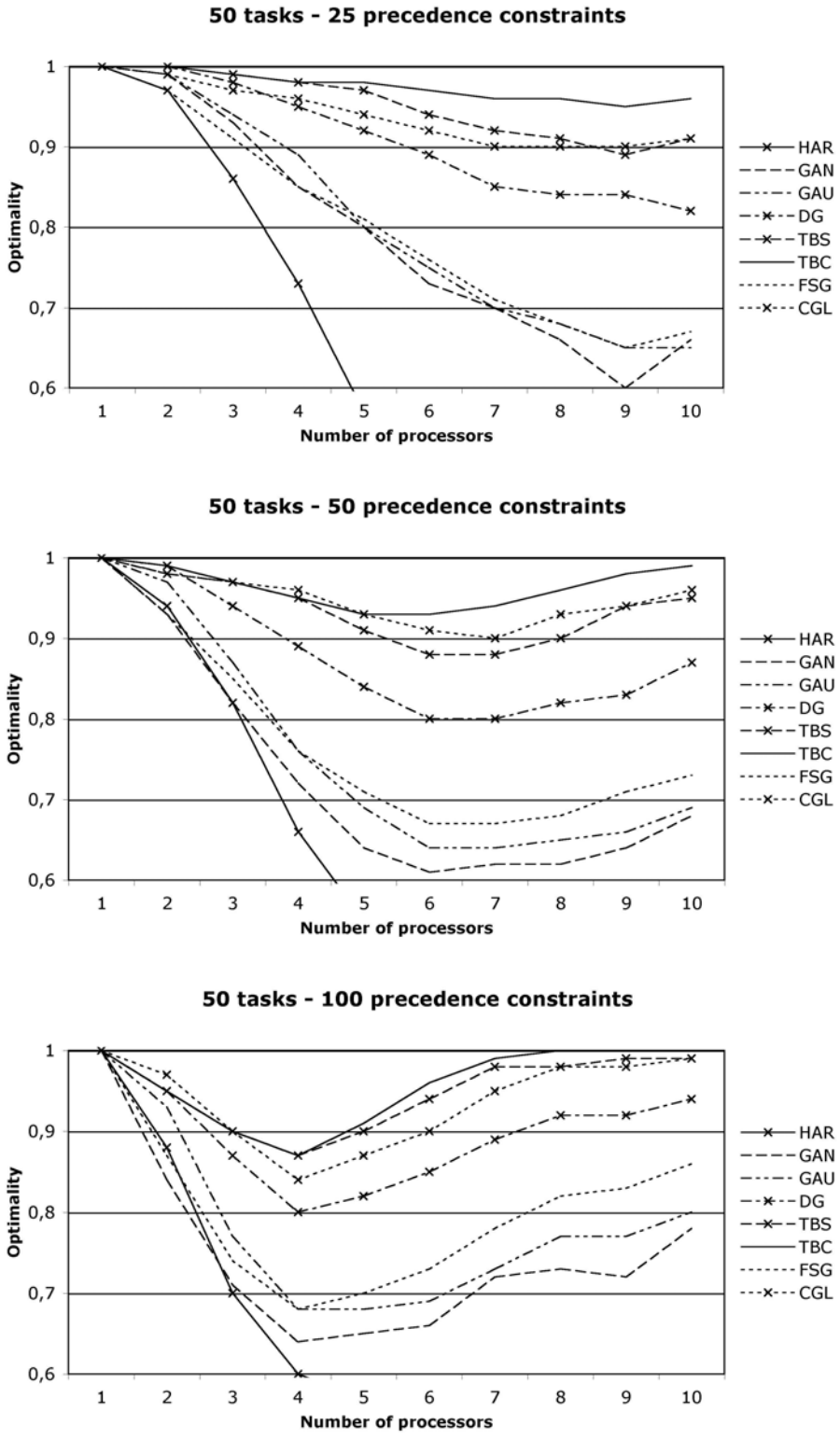


FIG. 9.6 – Performances de chaque méthode d'ordonnancement

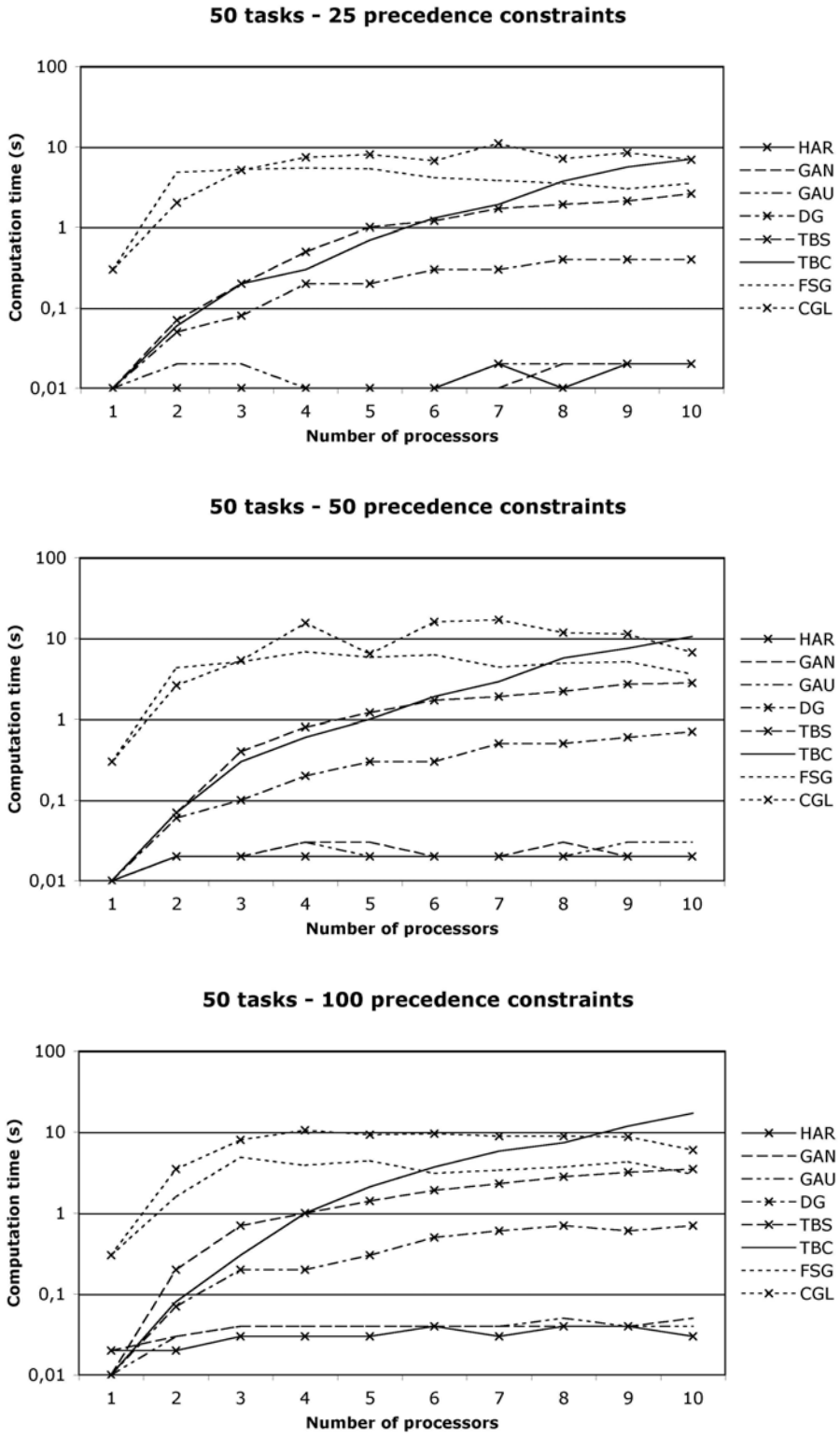


FIG. 9.7 – Temps de calcul moyen de chaque méthode d'ordonnancement

9.6 Conclusion

Nous avons présenté plusieurs méthodes d'approximation pour résoudre le problème de l'ordonnancement multiprocesseur des systèmes de tâches liées par des contraintes de précédence. Plusieurs d'entre elles fournissent des solutions de bonne qualité - i.e. proche de l'optimum. Les meilleures solutions sont celles obtenues par les approches basées sur la méthode taboue (TBS et TBC) et sur l'hybridation avec ED/MISF (CGL).

Néanmoins, la complexité algorithmique de ces trois méthodes est assez élevée. Le développement de méthodes hybrides semble être une voie de recherche intéressante pour obtenir une meilleure maîtrise de la complexité. L'utilisation des processus de décision markovien (MDP) est aussi une alternative possible. Par exemple, pour le problème de l'allocation de fréquence, l'utilisation des MDP a permis de diminuer le temps de calcul par rapport à un algorithme génétique, tout en obtenant des solutions de qualités identiques [53, 54].

Une autre voie de recherche consiste à étendre le modèle. Plusieurs enrichissements sont possibles :

1. la durée des communications entre deux tâches liées par une relation de précédence n'est pas nulle, elle peut être constante ou dépendre par exemple du placement des tâches sur les processeurs,
2. chaque tâche possède une échéance avant laquelle elle doit avoir terminé son exécution,
3. chaque tâche possède une date d'activation avant laquelle elle ne peut être exécutée.

Plusieurs travaux traitent de la première extension, par exemple [23, 35, 59, 60, 85, 87, 108]. Elle est importante dans le domaine des systèmes distribués puisqu'elle prend en compte le coût de communication entre deux tâches placées sur des processeurs différents. De plus, certains travaux prennent aussi en compte le débit des différentes liaisons réseaux entre les processeurs.

Les deuxième et troisième extensions revêtent une importance particulière puisqu'elles rendent possible l'application aux systèmes temps-réel. Plusieurs travaux les ont déjà traitées, par exemple [35, 85, 87]. En majorité, ces travaux sont issus de méthodes développées pour le contexte où les seules contraintes sont des relations de précédence. Généralement, il s'agit alors d'adapter la fonction de coût aux contraintes d'activation et d'échéances. En suivant la même approche, on peut adapter notre méthode à ces deux contraintes, et donc appliquer nos résultats au cas des systèmes temps-réel. Moyennant cette extension, les méthodes d'approximation que nous avons développées peuvent donc permettre de produire de nouvelles solutions d'ordonnancement pour les systèmes temps-réel.

Les travaux menés dans ce chapitre ont fait l'objet d'une publication dans les actes de la conférence IASTED Computational Intelligence 2007.

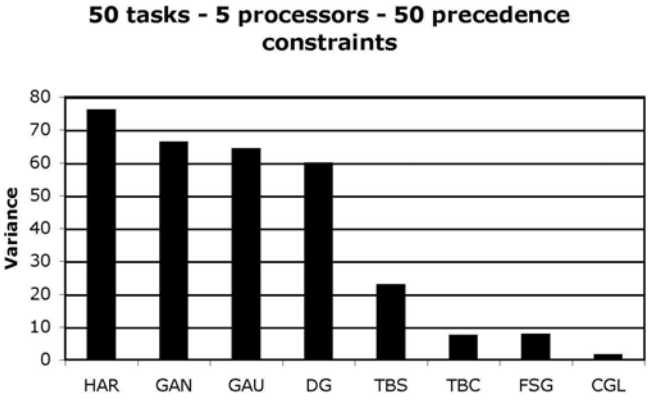


FIG. 9.8 – Variance de chaque méthode d'ordonnancement

Quatrième partie

Contributions au diagnostic des systèmes temps-réel

Chapitre 10

Modélisation par les chaînes de Markov

10.1 Introduction

Les noyaux temps-réel pilotent les tâches à l'aide d'un algorithme d'ordonnancement en-ligne dont les décisions ne s'appuient que sur la connaissance du présent. Pour certains cas particuliers, il existe des algorithmes en-ligne optimaux : dès qu'un système de tâches est ordonnançable, alors l'algorithme optimal réussit à l'ordonnancer. Cependant, dès que les tâches partagent plusieurs ressources, il est montré dans [34] qu'il n'existe pas d'algorithme optimal. Dans ce contexte, l'analyse de la validité d'une application temps-réel est un problème NP-dur [13, 70]. Plusieurs approches ont été développées, certaines par des méthodes exactes et d'autres par des méthodes approchées.

Le problème étant NP-dur, les méthodes exactes ont une complexité exponentielle. Les méthodes approchées sont apparues pour contourner cette difficulté : elles permettent d'obtenir une solution approchée d'un problème NP en un temps polynomial. Ces techniques sont basées sur le recuit simulé, la méthode taboue, les algorithmes génétiques et les processus de décision markoviens [89]. A notre connaissance, seuls les algorithmes génétiques ont été appliqués au problème de la validation des applications temps-réel [87] ; cependant, ils ne l'ont été que dans le cas particulier du contexte non-préemptif. Les articles [53, 54] présentent une comparaison entre ces méthodes appliquées au problème de l'allocation de fréquence ; [65] propose une application des processus de décision markoviens au problème de la planification de trajectoire en robotique.

Des méthodes de validation exactes ont été développées à partir de plusieurs formalismes : les réseaux de Petri [18, 45, 72], les automates finis [2, 3, 4, 63], la géométrie discrète [62, 64]. L'objectif de ces méthodes consiste généralement à produire une séquence d'exécution valide destinée à être implantée dans un séquenceur. Les chaînes de Markov [36, 82] sont utilisées dans [75] pour la validation temps-réel dans le cadre des contraintes souples, la violation d'une contrainte n'entraînant alors aucune conséquence grave pour le procédé.

Nous nous proposons de mettre en place une méthode exacte basée sur les chaînes de Markov pour étudier la validation des systèmes temps-réel critiques - i.e. à contraintes strictes. A notre connaissance, c'est leur première application dans ce contexte. Par rapport aux autres méthodes exactes, elles permettent d'obtenir des renseignements sur le comportement moyen d'un système de tâches en fonction des hypothèses d'ordonnancement choisies. Nous en attendons une amélioration substantielle des diagnostics fournis. Dans ce chapitre, nous traitons la modélisation des systèmes temps-réel par les chaînes de Markov. Nous insistons particulièrement sur la prise en compte des contraintes dues aux interactions entre les tâches et aux spécificités de l'exécutif temps-réel ciblé. Nous supposons que le lecteur est familier des chaînes de Markov. Au besoin, on peut consulter [36, 82].

10.2 Contexte d'étude

Les caractéristiques temporelles des tâches peuvent être quelconques : départs différés ou simultanés, échéance avant ou sur requête. Toutefois, nous n'étudions pas les tâches dont l'échéance est plus grande que la période. Dans la section 10.7, nous intégrons la plupart des contraintes liées aux interactions entre les tâches, par exemple les relations de précédence et l'exclusion mutuelle. Dans la section 10.6, nous étudions les contraintes dues aux caractéristiques des exécutifs temps-réel. Nous suivons ici l'approche globale : les tâches ne sont pas attribuées statiquement aux processeurs. Nous étudions chacun des trois contextes possibles : non-préemptif, préemptif avec migration partielle, et préemptif avec migration totale.

10.3 Modélisation des ordonnancements

Nous modélisons l'exécution d'une application temps-réel par des marches aléatoires engendrées par des chaînes de Markov. Les variables aléatoires que nous utilisons représentent l'état d'avancement des instances de chaque tâche. Dans le contexte des systèmes de tâches à échéances contraintes ($D_k \leq T_k$), on peut représenter chaque tâche en considérant seulement son instance courante puisque dans toute séquence d'exécution valide, il ne peut exister à chaque instant qu'au plus une instance active de chaque tâche. Pour $t \geq r_k$, l'instance courante de τ_k est $\tau_{k,\lfloor (t-r_k)/T_k \rfloor}$. Nous supposons que le temps est discret, ainsi chaque instant $t \in \mathbb{N}$ représente en fait l'intervalle temporel $[t, t+1[$.

On peut représenter l'exécution d'un système de tâches à l'aide de différentes variables aléatoires. La première que nous avons utilisée est une variable indicatrice d'événements (voir figure 10.1) :

- $M_t^k = 0 \Leftrightarrow$ l'instance courante de τ_k n'est pas exécutée à l'instant t
- $M_t^k = 1 \Leftrightarrow$ l'instance courante de τ_k est exécutée à l'instant t

Cette variable aléatoire permet de représenter l'exécution d'une tâche τ_k par une suite de variables aléatoires $(M_t^k)_{t\in\mathbb{N}}$. Pour modéliser l'ensemble du système de tâches, on utilise alors un n-uplet de chaînes de variables aléatoires.

Les valeurs des variables $M_t^1, \ldots, M_t^n$ déterminent les tâches exécutées à l'instant t. Elles permettent d'exprimer certaines contraintes comme le respect du nombre de processeurs par exemple. Cependant, pour vérifier si les échéances des instances sont respectées, on doit connaître le nombre de fois où chaque instance a eu accès à un processeur. Beaucoup de contraintes dépendent de l'état d'avancement des tâches. Pour cette raison, nous utilisons aussi la variable aléatoire suivante (voir figure 10.1) :

$$I_t^k = \sum_{u=0}^{t-1} M_u^k$$

La variable I_t^k comptabilise le nombre d'accès au processeur obtenus par les instances de la tâche τ_k sur l'intervalle $[0, t[$.

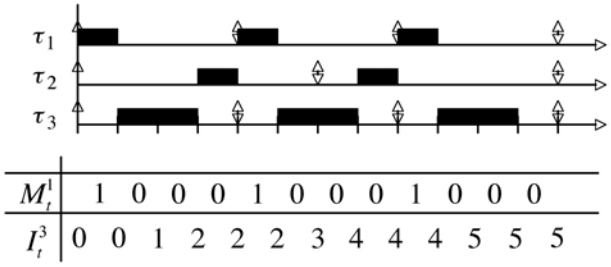


FIG. 10.1 – Illustration des variables aléatoires

Dans toute la suite, nous désignons par θ l'ensemble de tous les ordonnancements possibles, et par Θ le sous-ensemble restreint à ceux respectant toutes les contraintes imposées au système de tâches.

10.4 Mesure d'ordonnançabilité

Dans cette section, nous introduisons les aspects probabilistes de notre approche. Nous considérons que les choix d'ordonnancement possibles lors de l'exécution d'un système de tâches sont probabilistes. Ainsi, chaque loi de probabilité représente une classe de séquences d'exécution. Notre objectif se traduit par la mise en place d'une loi de probabilité sur θ destinée à l'étude des ordonnancements valides (Θ).

Examinons les propriétés nécessaires pour qu'une loi de probabilité renseigne effectivement sur l'ordonnançabilité d'un système de tâches. Pour cela, considérons le cas d'un algorithme d'ordonnancement en-ligne, par exemple RM. Nous pouvons construire une loi de probabilité qui lui correspond : les choix d'ordonnancements correspondant à ceux de RM sont associés à une probabilité non nulle, et les autres à une probabilité nulle. Or, RM n'est pas optimal dans tous les contextes : il existe donc des systèmes de tâches ordonnançables pour lesquels RM produit une séquence ne respectant pas les contraintes. Ainsi, la probabilité que certains systèmes de tâches soient ordonnançables par RM peut être nulle, bien que ceux-ci soient ordonnançables. Une telle loi de probabilité n'est donc pas satisfaisante pour étudier leur ordonnançabilité : il n'y a pas équivalence entre $\mathbb{P}(\Theta) > 0$ et le fait que le système de tâches soit ordonnançable.

Pour une loi de probabilité $\mathbb{P}$ quelconque sur θ, lorsque l'on a $\mathbb{P}(\Theta) > 0$, le système de tâches est forcément ordonnançable. Pour qu'une loi de probabilité soit une mesure d'ordonnançabilité, on impose que la réciproque soit aussi vérifiée.

Définition 10.1 *Une loi de probabilité $\mathbb{P}$ sur θ est une mesure de l'ordonnançabilité de τ si et seulement si :*

$$\mathbb{P}(\Theta) > 0 \Leftrightarrow \tau \text{ est ordonnançable}$$

La loi équiprobable attribue la même probabilité à tous les ordonnancements. Ainsi, dès qu'il existe un ordonnancement valide, elle fournit une probabilité d'ordonnançabilité non nulle. La loi équiprobable est donc une mesure d'ordonnançabilité. Un avantage de cette loi est son équité : aucun choix d'ordonnancement n'est favorisé par rapport à un autre.

On peut aussi définir d'autres mesures d'ordonnançabilité. Un exemple intéressant de loi est celui fourni par les séquences conservatives : un ordonnancement a une probabilité non nulle si et seulement si il est conservatif. D'un point de vue temps-réel, cette loi est très intéressante car les exécutifs actuels n'engendrent que des séquences conservatives : elle permet donc de se positionner dans leur contexte. Lorsque les tâches sont indépendantes, on sait que dès qu'il existe une séquence valide, alors il existe une séquence conservative valide. Ainsi, dans le contexte des tâches indépendantes, cette loi est une mesure d'ordonnançabilité.

Cette approche permet de mettre en place un processus de validation indépendamment de la loi de probabilité utilisée tant que les variables aléatoires $(I_t^k)_{k\in\{1,\ldots,n\}}$ vérifient la propriété de Markov. Cette modularité est intéressante puisqu'elle permet d'appliquer notre approche à des contextes spécifiques, et même d'aborder l'étude de l'ordonnançabilité d'un système de tâches selon différentes vues (lois).

10.5 Intégration des contraintes de validation

10.5.1 Contraintes de validation

Considérons une contrainte imposée au système de tâches et représentée par l'ensemble H des ordonnancements qui la satisfont. On peut en tenir compte de deux manières différentes :

- par condition : on détermine la probabilité qu'un ordonnancement soit valide sachant qu'il vérifie la contrainte, on calcule donc $\mathbb{P}(\Theta|H)$.
- par conjonction : on détermine la probabilité qu'un ordonnancement soit valide et qu'il vérifie la contrainte, on calcule donc $\mathbb{P}(\Theta \wedge H)$.

Ces deux méthodes d'intégration d'une contrainte sont complémentaires. Elles sont aussi équivalentes pour la validation, car dès que l'on a $\mathbb{P}(H) > 0$, la définition des probabilités conditionnelles impose :

$$\mathbb{P}(\Theta|H) > 0 \Leftrightarrow \mathbb{P}(\Theta \wedge H) > 0$$

Dans la méthode par condition, les contraintes sont prises en compte au niveau de la loi de probabilité : dès qu'un ordonnancement a une probabilité non nulle, alors il respecte la contrainte. Cette méthode rend l'utilisation de la loi de probabilité plus technique puisque les variables aléatoires ne sont généralement plus indépendantes. Cependant, cette méthode permet de focaliser l'étude sur une classe d'ordonnancement spécifique. Elle est particulièrement adaptée pour intégrer les contraintes propres à l'exécutif temps-réel : migration totale ou partielle, conservatisme, préemptivité, etc.

La prise en compte par conjonction revient à contraindre les marches aléatoires autorisées par la loi de probabilité en leur imposant un parcours spécifique. Ainsi, les contraintes prises en compte de cette manière n'influent pas directement sur la loi de probabilité. Cette méthode offre donc une certaine modularité puisque l'introduction d'une nouvelle contrainte n'impose pas une réétude complète de la loi de probabilité. Elle est très utile pour intégrer les contraintes propres aux interactions entre les tâches : relation de précédence, exclusion mutuelle, etc.

10.5.2 Intégration par conjonction

Dans toute cette section, nous considérons :
- une chaîne de Markov X à temps et espace d'états discrets,
- l'ensemble E des valeurs des variables X_t,
- une bijection f de $\{1, \ldots, |E|\}$ dans E.

Comme dans la section précédente, considérons une contrainte imposée aux marches aléatoires de X représentée par l'ensemble H des marches qui la satisfont. Pour intégrer cette contrainte par conjonction, on doit l'exprimer en utilisant seulement les transitions effectuées lors d'une marche aléatoire. Lorsque c'est possible, nous disons que cette contrainte est markovienne.

Définition 10.2 *Une contrainte markovienne est un prédicat Q dépendant des transitions et du temps, i.e. de la forme $Q_t(X_t, X_{t+1})$.*

Pour intégrer H par conjonction, on recherche donc une contrainte markovienne Q vérifiant :

$$(X_t)_{t\in\mathbb{N}} \in H \Leftrightarrow \forall t \in \mathbb{N}, Q_t(X_t, X_{t+1})$$

Cette décomposition d'une contrainte portant sur l'ensemble d'une marche aléatoire en un prédicat portant sur les transitions effectuées lors de cette marche constitue la principale source de difficultés lors de la modélisation des ordonnancements valides.

Le résultat suivant fournit une méthode pour calculer la probabilité qu'une contrainte markovienne soit satisfaite. Il permet d'intégrer par conjonction toutes les contraintes de validation qui sont décomposables en contraintes markoviennes.

Théorème 10.1 *Soient Q une contrainte markovienne et $\lambda \in \mathbb{N}$. On a :*

$$\mathbb{P}(\bigwedge_{t=0}^{\lambda-1} Q_t(X_t, X_{t+1})) = \sum_{k=1}^{|E|} (D.M_0 \ldots M_{\lambda-1})^k$$

où :
- *D est la distribution initiale : $\forall k \in \{1, \ldots, |E|\}, D^k = \mathbb{P}(X_0 = f(k))$*

– $M_t = [m_{ij}]_{i,j\in\mathbb{N}}$ *est la matrice de transition :*

$$m_{ij} = \mathbb{P}(X_{t+1} = f(j)|X_t = f(i)) \times \mathbb{P}(Q_t(f(i), f(j)))$$

Démonstration :
Posons D_t la distribution de probabilité de la chaîne de Markov $(X_n)_{n\in\mathbb{N}}$ à un instant t quelconque lorsque la contrainte Q est respectée :

$$\forall k \in \{1, \ldots, |E|\}, D_t^k = \mathbb{P}(X_t = f(k), Q_0(X_0, X_1), \ldots, Q_{t-1}(X_0, X_t))$$

Pour établir le résultat, nous établissons une relation permettant de calculer D_{t+1} à partir de D_t. Pour alléger l'écriture, nous utilisons la notation suivante :

$$Q(X_0, X_1, \ldots, X_t) \Leftrightarrow \bigwedge_{t'=0}^{t-1} Q_t(X_{t'}, X_{t'+1}))$$

En décomposant D_{t+1} par la règle des probabilités totales sur les valeurs des variables $X_0, \ldots, X_t$, on obtient pour tout $k \in \{1, \ldots, |E|\}$:

$$D_{t+1}^k = \sum_{v_0,\ldots,v_t\in E} \mathbb{P}(Q(X_0, \ldots, X_{t+1}), X_{t+1} = f(k), X_t = v_t, \ldots, X_0 = v_0)$$

Or Q est un prédicat, donc pour une suite d'états $e_0, \ldots, e_t \in E$ et un instant $t \in \mathbb{N}$ donnés, on a : $\mathbb{P}(Q(e_0, \ldots, e_t)) \in \{0, 1\}$. Dans l'expression précédente de D_{t+1}^k, les valeurs des variables $X_0, \ldots, X_t, X_{t+1}$ sont connues, on en déduit pour tout $k \in \{1, \ldots, |E|\}$:

$$D_{t+1}^k = \sum_{v_0,\ldots,v_t\in E} \mathbb{P}(Q(v_0, \ldots, v_t, f(k))) \times \mathbb{P}(X_{t+1} = f(k), X_t = v_t, \ldots, X_0 = v_0)$$

En utilisant la relation des probabilités conditionnelles, puis la propriété de Markov, on obtient successivement pour tout $k \in \{1, \ldots, |E|\}$:

$$D_{t+1}^k = \sum_{v_0,\ldots,v_t\in E} \left(\begin{array}{l} \mathbb{P}(Q(v_0, \ldots, v_t, f(k))) \times \mathbb{P}(X_t = v_t, \ldots, X_0 = v_0) \\ \times \mathbb{P}(X_{t+1} = f(k)|X_t = v_t, \ldots, X_0 = v_0) \end{array} \right)$$

$$D_{t+1}^k = \sum_{v_0,\ldots,v_t\in E} \left(\begin{array}{l} \mathbb{P}(Q(v_0, \ldots, v_t, f(k))) \times \mathbb{P}(X_t = v_t, \ldots, X_0 = v_0) \\ \times \mathbb{P}(X_{t+1} = f(k)|X_t = v_t) \end{array} \right)$$

En factorisant v_t et en réutilisant le fait que Q est un prédicat, on obtient pour tout $k \in \{1, \ldots, |E|\}$:

$$D_{t+1}^k = \sum_{v_t\in E} \left(\begin{array}{l} \mathbb{P}(Q_t(v_t, f(k))) \times \mathbb{P}(X_{t+1} = f(k)|X_t = v_t) \\ \times \sum_{v_0,\ldots,v_{t-1}\in E} \mathbb{P}(Q(v_0, \ldots, v_t), X_t = v_t, \ldots, X_0 = v_0) \end{array} \right)$$

En réutilisant la règle des probabilités totales sur les variables $X_0, \ldots, X_{t-1}$, on obtient alors pour tout $k \in \{1, \ldots, |E|\}$:

$$D_{t+1}^k = \sum_{v_t\in E} \left(\begin{array}{l} \mathbb{P}(Q_t(v_t, f(k))) \times \mathbb{P}(X_{t+1} = f(k)|X_t = v_t) \\ \times \mathbb{P}(Q(X_0, \ldots, X_t), X_t = v_t) \end{array} \right)$$

En introduisant l'élément $D_t^{f^{-1}(v_t)}$, on obtient enfin une relation entre D_t et D_{t+1} :

$$D_{t+1}^k = \sum_{v_t\in E} \mathbb{P}(Q_t(v_t, f(k))) \times \mathbb{P}(X_{t+1} = f(k)|X_t = v_t) \times D_t^{f^{-1}(v_t)}$$

En utilisant la matrice M_t, on peut réécrire l'expression ci-dessus de la manière suivante :

$$D_{t+1} = M_t \times D_t$$

En l'utilisant récursivement, on obtient le résultat recherché.
CQFD.

10.6 Contraintes liées à l'exécutif

10.6.1 Migration totale

Pour assurer qu'un ordonnancement est faisable sur une architecture à p processeurs sous l'hypothèse de migration totale, il est suffisant de vérifier qu'à chaque instant t au plus p tâches s'exécutent simultanément :

$$\sum_{k=1}^{n}(I_{t+1}^k - I_t^k) \leq p$$

Remarquons que par définition, on a $0 \leq I_{t+1}^k - I_t^k \leq 1$, puisqu'une même tâche ne peut exécuter qu'une instruction pendant une unité de temps. Ce prédicat permet d'exprimer le respect de l'architecture (dans le contexte de l'hypothèse de migration totale) sous la forme de contraintes markoviennes

Exemple 10.1 *Considérons un système composé de trois tâches identiques de paramètres $r_k = 0$, $C_k = 2$, $D_k = T_k = 3$, et étudions la séquence d'exécution présentée sur la figure 10.2. A chaque instant, il y a exactement 2 tâches qui s'exécutent, le nombre de processeurs disponibles est donc respecté. La validité temporelle restreint les états possibles seulement aux instants $t \in 3\mathbb{N}$: on doit avoir $I_t^k = C_k.\lfloor t/3 \rfloor$. Dans la séquence donnée ci-dessus, on a bien pour t=0, $I_0^k = 2.\lfloor 0/3 \rfloor = 0$, et pour t=3, $I_3^k = 2.\lfloor 3/3 \rfloor = 2$. Cette séquence respecte donc aussi les échéances des tâches.*

10.6.2 Migration partielle

Sous l'hypothèse de migration totale, les tâches peuvent se répartir sans contrainte entre les processeurs, pourvu qu'à chaque instant il n'y ait pas plus de tâches à exécuter que de processeurs : c'est la contrainte exprimée dans la section 10.6.1. Sous l'hypothèse de migration partielle, une même instance ne peut pas changer de processeur, il devient nécessaire de représenter la répartition des instances sur les processeurs.

Comme toutes les instructions d'une même instance sont exécutées sur le même processeur, on peut associer chaque instance au processeur sur lequel elle est exécutée. Nous devons alors vérifier qu'à chaque instant le nombre d'instances exécutées et associées à un même processeur est inférieur ou égal à 1. Définissons un vecteur de placement $(Y_t^1, \ldots, Y_t^n)$ de la façon suivante :

- $Y_t^k = 0 \Leftrightarrow$ l'instance courante de la tâche τ_k n'est pas placée, ce cas correspond à deux situations possibles : soit elle est activée mais n'a pas encore exécuté d'instruction, soit elle a exécuté toutes ses instructions (elle est donc inactive).
- $Y_t^k = i \Leftrightarrow$ l'instance courante de la tâche τ_k est attribuée au $i^{\text{è}}$ processeur.

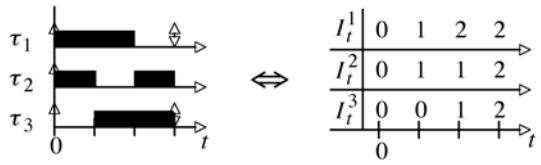


FIG. 10.2 – Exemple d'ordonnancement du système de tâches de l'exemple 10.1

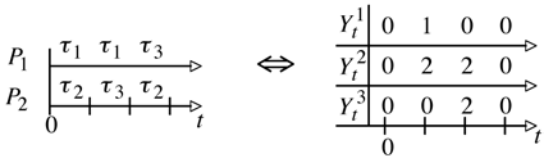


FIG. 10.3 – Exemple de placement du système de tâches de l'exemple 10.2

Déterminons maintenant les conditions correspondant au respect de la migration partielle. Dans la définition du vecteur de placement, nous avons contraint les valeurs des variables $(Y_t^1, \ldots, Y_t^n)$ en fonction de celles de $(I_t^1, \ldots, I_t^n)$:

$$\forall t \in \mathbb{N}, \forall k \in \{1, \ldots, n\}, Y_t^k = 0 \Leftrightarrow I_t^k \in C_k\mathbb{N}$$

Une instance ne peut pas changer de processeurs en cours d'exécution, cette contrainte s'exprime par la condition suivante :

$$\forall t \in \mathbb{N}, \forall k \in \{1, \ldots, n\}, \left(I_t^k \notin C_k\mathbb{N} \wedge I_{t+1}^k \notin C_k\mathbb{N}\right) \Rightarrow Y_t^k = Y_{t+1}^k$$

Nous pouvons maintenant formuler la contrainte signifiant qu'au plus une tâche associée à un processeur peut être exécutée à un instant donné :

$$\forall t \in \mathbb{N}, \forall i \in \{1, \ldots, p\}, \sum_{k \in E_i} (I_{t+1}^k - I_t^k) \leq 1$$

où E_i est défini par : $E_i = \{k \in \{1, \ldots, n\} | Y_t^k = i \vee Y_{t+1}^k = i\}$

Par définition, les variables de placement Y_t^k associées à des tâches τ_k dont le paramètre C_k vaut 1 sont toujours égales à 0. Effectivement, avant que ces tâches ne s'exécutent, leur variable respective vaut 0 (tâche non placée), et après leur première exécution, leur variable respective vaut encore 0 (tâche inactive). Ainsi, le prédicat ci-dessus n'est pas suffisant pour traiter ce type de tâche. Pour celles-ci, on doit seulement s'assurer qu'il existe un processeur de disponible au moment auquel elles sont exécutées. On doit alors vérifier en plus :

$$\forall t \in \mathbb{N}, \sum_{k=1}^{n} (I_{t+1}^k - I_t^k) \leq p$$

Ces prédicats permettent d'exprimer la contrainte de migration partielle sous la forme d'une contrainte markovienne.

Exemple 10.2 *Reprenons l'exemple 10.1 en se plaçant sous l'hypothèse de migration partielle. Considérons la répartition des tâches sur les processeurs indiquée sur la figure 10.3. Dès l'instant $t = 2$, les tâches τ_2 et τ_3 sont attribuées au processeur n°2. Or, entre les instants 2 et 3, elles s'exécutent simultanément, ainsi la dernière condition que nous avons formulée pour exprimer la contrainte de migration partielle n'est pas vérifiée : ce placement n'est donc pas réalisable.*

10.6.3 Non-préemption

Les exécutifs non-préemptifs ne permettent pas qu'une instance en cours d'exécution soit interrompue : dès qu'une instance débute son exécution, elle est exécutée jusqu'à terminaison. Cette contrainte correspond à la propriété suivante :

$$\forall t \in \mathbb{N}, \forall k \in \{1, \ldots, n\}, I_t^k \notin C_k\mathbb{N} \Rightarrow I_{t+1}^k = I_t^k + 1$$

Cette condition est une contrainte markovienne, elle permet donc d'intégrer par conjonction l'hypothèse de non-préemption.

10.7 Contraintes liées aux tâches

10.7.1 Validité temporelle

Un ordonnancement est valide temporellement si toutes les tâches respectent toutes leurs échéances. Ainsi, à certains instants précis, les tâches doivent être dans un état précis (voir figure 10.4).

Cette contrainte porte sur les états atteints à certains instants. Elle est équivalente à la conjonction, pour tout $k \in \{1, \ldots, n\}$ et tout $t \geq r_k$, des deux conditions suivantes :

- $(t - r_k) = D_k \Rightarrow I_t^k = C_k.\lceil (t - r_k)/T_k \rceil$
- $(t - r_k) = T_k \Rightarrow I_t^k = C_k.\lfloor (t - r_k)/T_k \rfloor$

La première implication assure que chaque instance respecte son échéance, et la seconde qu'aucune instruction n'est exécutée lorsqu'aucune instance n'est active. Ces conditions permettent donc d'exprimer la validité temporelle sous la forme de contrainte markovienne. Pour les tâches à échéances sur requête ($D_k = T_k$), seule la seconde condition est pertinente.

10.7.2 Relations de précédence

Considérons deux tâches liées par la relation suivante : τ_i précède τ_j . Nous traitons ici seulement le cas de la précédence simple, nous pouvons donc supposer $T_i = T_j$. La contrainte de précédence entre τ_i et τ_j impose que la $k^{\text{è}}$ instance de τ_j ne peut débuter son exécution qu'à partir du moment où la $k^{\text{è}}$ instance de τ_i a terminé la sienne. Ceci conduit à la contrainte suivante :

$$\forall t \in \mathbb{N}, I_t^j > C_j.\lfloor (t - r_j)/T_j \rfloor \Rightarrow I_t^i = C_i.\lceil (t - r_j)/T_j \rceil$$

Ce prédicat permet d'exprimer la contrainte de précédence entre deux tâches sous la forme d'une contrainte markovienne.

Exemple 10.3 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k
τ_1	0	1	3	3
τ_2	0	2	6	6
τ_3	0	1	3	3

La séquence d'exécution présentée sur la figure 10.5 vérifie la contrainte τ_1 précède τ_3, mais pas la contrainte τ_3 précède τ_1. Etudions maintenant la suite des variables $(I_t^1, \ldots, I_t^n)$ qui correspond à cette séquence (voir figure 10.5).

La condition équivalente à la contrainte τ_1 précède τ_3 s'écrit :

$$\forall t \in \mathbb{N}, I_t^3 > \lfloor t/3 \rfloor \Rightarrow I_t^1 = \lceil t/3 \rceil$$

De $t = 1$ à $t = 3$, on a $I_t^1 = 1$, et de $t = 4$ à $t = 6$, on a $I_t^1 = 2$, la condition donnée ci-dessus est donc bien vérifiée et la contrainte "τ_1 précède τ_3" est bien validée.

Par contre, on a $I_1^1 = 1$ et $I_1^3 = 0$. Ainsi, on a $I_1^1 > \lfloor 1/3 \rfloor$ sans avoir $I_1^1 = \lceil 1/3 \rceil$, la contrainte "$\tau_3$ précède τ_1" n'est donc pas respectée.

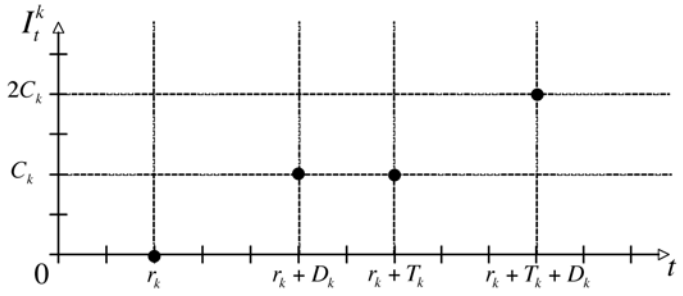


FIG. 10.4 – "Passages" imposés par la validité temporelle

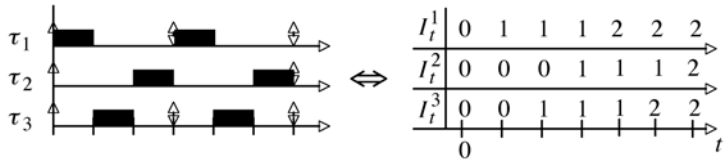


FIG. 10.5 – Exemple d'ordonnancement du système de tâches de l'exemple 10.3

10.7.3 Exclusion mutuelle

Nous supposons ici qu'une même tâche réalise au plus un accès aux ressources partagées, et nous décrivons cet accès par les deux paramètres suivants :

- α_k : la première instruction de la section critique,
- β_k : la dernière instruction de la section critique.

Définissons, pour chaque tâche τ_k, sa zone critique Z_k qui correspond aux valeurs de I_t^k strictement incluses dans l'accès à la ressource partagée :

$$Z_k = \{(i_1, \ldots, i_n) | \alpha_k \leq i_k \bmod C_k < \beta_k\}$$

Les zones critiques permettent de définir une première condition, puisqu'à chaque instant une seule tâche peut être en zone critique :

$$\forall t \in \mathbb{N}, (I_t^1, \ldots, I_t^n) \notin \bigcup_{\substack{k,k' \in \{1,\ldots,n\} \\ k \neq k'}} Z_k \cap Z_{k'}$$

Cette condition est une contrainte markovienne. Cependant, elle ne permet pas de protéger totalement les accès aux ressources partagées (voir exemple 10.4) : nous devons étendre la notion de zone critique aux transitions.

Exemple 10.4 *Considérons le système de tâches suivant :*

	r_k	C_k	D_k	T_k	α_k	β_k
τ_1	0	2	3	3	1	2
τ_2	0	2	3	3	1	1

La zone critique de ce système est vide. Considérons maintenant la séquence d'exécution biprocesseur présentée sur la figure 10.6. Le système de tâches ne passe jamais par des états appartenant à la zone critique, cependant cette séquence n'est pas valide puisque entre $t = 1$ et $t = 2$, deux accès aux ressources partagées sont réalisés simultanément.

Pour une transition (I_t^k, I_{t+1}^k), notons Δ_k le nombre d'instructions exécutées par les instances de la tâche τ_k précédant l'instant t : $\Delta_k = C_k.\lfloor I_t^k / C_k \rfloor$. Uune tâche dont la section critique contient une seule instruction exécute son instruction critique si et seulement si la condition suivante est vérifiée :

$$I_t^k - \Delta_k = \alpha_k - 1 \wedge I_{t+1}^k = \beta_k$$

En généralisant cette condition, on étend la notion de zone critique aux transitions :

$$Z'_k = \{(i_1, \ldots, i_n) \times (i'_1, \ldots, i'_n) | \alpha_k - 1 \leq i_k - \Delta_k < \beta_k \wedge \alpha_k - 1 < i'_k - \Delta_k \leq \beta_k\}$$

Nous pouvons maintenant formuler une nouvelle condition :

$$\forall t \in \mathbb{N}, (I_t^1, \ldots, I_t^n) \times (I_{t+1}^1, \ldots, I_{t+1}^n) \notin \bigcup_{\substack{k,k' \in \{1,\ldots,n\} \\ k \neq k'}} Z'_k \cap Z'_{k'}$$

Cette condition est une contrainte markovienne et permet d'assurer l'exclusion mutuelle des sections critiques.

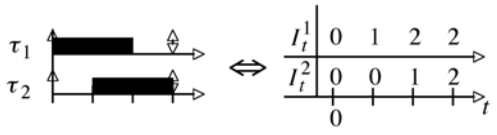


FIG. 10.6 – Exemple d'ordonnancement du système de tâches de l'exemple 10.4

Exemple 10.5 *Reprenons l'exemple 10.4. Les transitions critiques de la tâche* τ_1 *sont* $(0,1)$ *et* $(1,2)$*, et celle de* τ_2 *est* $(0,1)$*. La zone critique sur les transitions contient les transitions interdites suivantes :*

$$((0,0),(1,1)),((1,0),(1,1)),((1,0),(2,1))$$

Or, entre l'instant $t=1$ *et* $t=2$*, le système réalise la transition* $((1,0),(2,1))$*. Cette transition appartient à la zone critique, la condition permettant de prendre en compte l'accès aux ressources partagées invalide donc cette séquence.*

10.8 Conclusion

Le modèle que nous avons défini permet d'intégrer le comportement temporel des tâches périodiques et de prendre en compte un grand nombre d'interactions entre les tâches comme les relations de précédence et l'exclusion mutuelle par exemple. De plus, nous avons aussi pris en compte les spécificités de l'exécutif temps-réel utilisé, par exemple : préemption des tâches, migration totale ou partielle entre les processeurs. Notre chaîne de Markov permet donc de prendre en compte la plupart des contraintes généralement utilisées dans les systèmes temps-réel.

Les chaînes de Markov offrent une certaine richesse de modélisation. La prise en compte des interactions entre les tâches est réalisée à partir de contraintes markoviennes exprimées à partir des transitions du modèle. Si nécessaire, elles peuvent dépendre du temps. La loi de probabilité utilisée pour déterminer les probabilités de transition permet d'intégrer les contraintes dues à l'architecture de l'exécutif ciblé. Ainsi, ses spécificités déterminent le comportement probabiliste du système de tâches. Cette approche permet de déterminer la probabilité pour que les contraintes dues aux interactions entre les tâches soient respectées sachant l'architecture matérielle utilisée.

Chapitre 11

Algorithme de calcul

11.1 Introduction

Dans le chapitre précédent, nous avons modélisé l'exécution d'un système de tâches à l'aide des chaînes de Markov. Les variables aléatoires $I_t = (I_t^1, \ldots, I_t^n)$ permettent d'intégrer la plupart des contraintes des systèmes temps-réel. Toutefois, le nombre des valeurs possibles de ces variables est infini. On ne peut donc pas les utiliser directement. Pour ramener leur étude à une analyse finie, nous avons envisagé plusieurs solutions.

Tout d'abord, nous avons développé un modèle où chaque tâche est représentée par deux informations : la charge CPU déja exécutée par son instance courante, et la laxité de son instance courante. Ces informations varient respectivement dans les intervalles $\{0, \ldots, C_k\}$ et $\{0, \ldots, D_k - C_k\}$ pour la tâche τ_k. Ce modèle permet de représenter de manière finie le comportement d'un système de tâches. Généralement, on peut aussi utiliser une matrice de transition ne dépendant pas du temps. Le modèle est alors une chaîne de Markov homogène par rapport au temps. De nombreux résultats sont connus pour ce type de chaîne de Markov, ce modèle permet de les réutiliser. Cependant, la complexité du nombre d'états de ce modèle est très élévée, de l'ordre de $O(C^n \times (D-C)^n)$. Une telle complexité rend ce modèle inutilisable en pratique.

Ensuite, nous nous sommes tournés vers des solutions basées sur les chaînes de Markov dynamiques : la matrice de transition dépend alors du temps. Ce type de chaînes de Markov est moins étudié, cependant certains travaux en traitent, par exemple [48]. Pour ramener le modèle à un nombre d'états fini, nous avons développé une méthode d'agrégation dynamique des états.

Dans la section 11.2, nous présentons tout d'abord les généralités sur notre méthode. Ensuite, dans les sections 11.3 et 11.4, nous développons deux réductions pour les variables $I_t = (I_t^1, \ldots, I_t^n)$ ayant des propriétés différentes. Finalement dans la section 11.5, nous évaluons expérimentalement la complexité algorithmique de notre méthode.

11.2 Réduction de l'espace d'états

La méthode que nous proposons pour réduire l'espace d'états est une agrégation dynamique : à chaque instant, on "fusionne" les états similaires. Tout d'abord, nous présentons la technique que nous avons utilisée, ensuite nous donnons deux réductions possibles du modèle $(t, I_t^1, \ldots, I_t^n)$ en discutant leurs avantages et leurs différences.

11.2.1 Agrégation dynamique d'états

Dans toute cette section, nous adoptons les notations suivantes :
- $(X_t)_{t \in \mathbb{N}}$: une chaîne de Markov,
- E : l'ensemble des valeurs des variables aléatoires X_t,
- f : une bijection de $\{1, \ldots, |E|\}$ dans E,

– Q : une contrainte markovienne de X.

Définition 11.1 *Une réduction de X est un triplet (Π, g, h) tel que :*
– *Π est une suite $(\Pi_t)_{t\in\mathbb{N}}$ de sous-ensembles de E,*
– *g est une fonction de $\mathbb{N}$ dans $\mathbb{N}$,*
– *h est une suite $(h_t)_{t\in\mathbb{N}}$ de fonctions injectives h_t de $\Pi_t \cup \Pi_{t+1}$ dans E.*

Les ensembles Π_t décrivent les états représentés à chaque instant t, la fonction g correspond à la réduction dans le temps, et les fonctions h_t à la réduction des états. On associe à une réduction (Π, g, h) un sous-ensemble de E correspondant aux états réduits, noté $E(\Pi, g, h)$, et défini par :

$$E(\Pi, g, h) = \bigcup_{t\in\mathbb{N}} h_t(\Pi_t \cup \Pi_{t+1})$$

Une réduction (Π, g, h) permet d'étudier X en se restreignant aux états réduits appartenant à $E(\Pi, g, h)$. A chaque instant t, elle indique les états à prendre en compte pour évaluer le passage à l'instant $t + 1$. La figure 11.1 illustre le principe de fonctionnement de cette technique.

Pour être utilisable, une réduction doit vérifier certaines propriétés. La définition suivante donne les trois caractéristiques nécessaires pour assurer que le comportement d'une chaîne de Markov réduite est identique à celui de la chaîne de Markov initiale.

Définition 11.2 *Une réduction (Π, g, h) est* compatible *avec X et Q si et seulement si elle satisfait les trois propriétés suivantes :*
– *Compatibilité avec les états :*

$$\forall t \in \mathbb{N}, \forall e \in E, e \notin \Pi_t \Rightarrow \mathbb{P}(X_t = e \wedge \forall u < t, Q_t(X_u, X_{u+1}) = 0$$

– *Compatibilité avec le modèle :*

$$\forall t \in \mathbb{N}, \forall e \in \Pi_t, \forall e' \in \Pi_{t+1}, \mathbb{P}(X_{t+1} = e' | X_t = e) = \mathbb{P}(X_{g(t)+1} = h(t, e') | X_{g(t)} = h(t, e))$$

– *Compatibilité avec les contraintes :*

$$\forall t \in \mathbb{N}, \forall e \in \Pi_t, \forall e' \in \Pi_{t+1}, Q_t(e, e') \Leftrightarrow Q_{g(t)}(h(t, e), h(t, e'))$$

La compatibilité avec les états assure que tous les états significatifs (de probabilité non nulle) sont bien pris en compte par la réduction. La compatibilité avec le modèle et celle avec les contraintes assurent que les probabilités de transition (i.e. le comportement de la chaîne de Markov) sont identiques dans le modèle initial et dans le modèle réduit. Les notions de compatibilité dépendent de la contrainte de validation Q : c'est en exploitant les propriétés de cette contrainte que l'on peut mettre en place une réduction efficace.

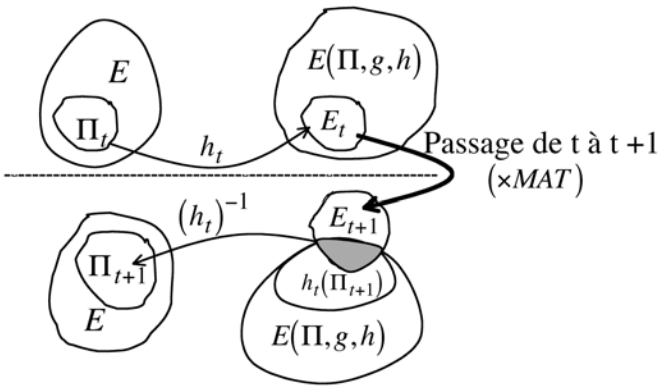


FIG. 11.1 – Technique de réduction

Dans toute la suite de cette section, nous désignons par (Π, g, h) une réduction de X vérifiant les trois conditions de compatibilité ci-dessus, et par f' une bijection de $\{1, \ldots, |E(\Pi, g, h)|\}$ dans $E(\Pi, g, h)$,

Deux fonctions de passage sont utilisées dans le procédé indiqué sur la figure 11.1 : l'une permet de passer de Π_t à $E(\Pi, g, h)$, et l'autre de $E(\Pi, g, h)$ à Π_{t+1}. Définissons formellement ces deux fonctions.

Définition 11.3 *La fonction de passage $red_t(D) = d$ d'une distribution D à sa forme réduite d est définie de la façon suivante :*

$$\forall k \in \{1, \ldots, |E(\Pi, g, h)|\}, \begin{cases} f'(k) \in h_t(\Pi_t) \Rightarrow d_k = D_{f^{-1}(h_t^{-1}(f'(k)))} \\ f'(k) \notin h_t(\Pi_t) \Rightarrow d_k = 0 \end{cases}$$

Définition 11.4 *La fonction de passage $aug_t(d) = D$ de la distribution réduite d à la distribution D est définie de la façon suivante :*

$$\forall k \in \{1, \ldots, |E|\}, \begin{cases} f(k) \in \Pi_t \Rightarrow D_k = d_{f'^{-1}(h_t(f(k)))} \\ f(k) \notin \Pi_t \Rightarrow D_k = 0 \end{cases}$$

Le théorème 11.1 montre que les distributions de probabilité obtenues avec et sans réduction coïncident dès que la réduction est compatible. Il fournit une méthode pour évaluer la distribution à l'instant t en utilisant la réduction, et donc par une méthode de calcul dont la complexité est inférieure.

Théorème 11.1 *Soient D_t et D_{t+1} les distributions de probabilité, respectivement à l'instant $t \in \mathbb{N}$ et à l'instant $t+1$. Lorsque la contrainte Q est respectée, on a :*

$$D_{t+1} = aug_t(red_t(D_t) \times M'_t)$$

où $M'_t = [m'_{ij}]_{i,j \in \{1,\ldots,|E(\Pi,g,h)|\}}$ est la matrice de transition définie sur $E(\Pi, g, h)$:

$$m'_{ij} = \mathbb{P}(X_{g(t)+1} = f'(j) | X_{g(t)} = f'(i)) \times \mathbb{P}(Q_{g(t)}(f'(i), f'(j)))$$

Démonstration :

La relation entre D_t et D_{t+1} est donnée dans la preuve du théorème 10.1. Pour tout $k \in \{1, \ldots, |E|\}$, on a :

$$D^k_{t+1} = \sum_{v_t \in E} m_{f^{-1}(v_t),k} \times D_t^{f^{-1}(v_t)}$$

La réduction étant compatible avec les états supprimés, tous les états $e \in E \setminus \Pi_t$ vérifient $\mathbb{P}(X_t = e \wedge (Q_u(X_u, X_{u+1}))_{0 \leq u < t}) = 0$, et donc $D_t^{f^{-1}(e)} = 0$. On peut alors réduire la somme précédente aux seuls éléments $e \in \Pi_t$. On obtient :

$$D^k_{t+1} = \sum_{v_t \in \Pi_t} m_{f^{-1}(v_t),k} \times D_t^{f^{-1}(v_t)}$$

Pour $k \in \{1, \ldots, |E(\Pi, g, h)|\}$ vérifiant $f'(k) \in h(t, \Pi_t)$, on a par définition :

$$red_t(D_t)_k = D_t^{k'}, \text{ où } k' = f^{-1}(h_t^{-1}(f'(k)))$$

Remarquons la propriété suivante :

$$\begin{aligned} & f^{-1}(h_t^{-1}(f'(\{k \in \{1, \ldots, |E(\Pi, g, h)|\} | f'(k) \in h(t, \Pi_t)\}))) \\ &= f^{-1}(h_t^{-1}(h(t, \Pi_t))) \\ &= f^{-1}(\Pi_t) \end{aligned}$$

En substituant k' par $f^{-1}(v_t)$, on obtient pour $v_t \in \Pi_t$:

$$red_t(D_t)_k = D_t^{f^{-1}(v_t)}, \text{ où } k = f'^{-1}(h_t(v_t))$$

On peut donc exprimer D^k_{t+1} en fonction de $red_t(D_t)$, on a pour $k \in \{1, \ldots, |E|\}$:

$$D^k_{t+1} = \sum_{v_t \in \Pi_t} m_{f^{-1}(v_t),k} \times red_t(D_t)_{f^{-1}(v_t)}$$

Par compatibilité de la réduction avec le modèle et avec les contraintes, on obtient pour tout $t \in \mathbb{N}$ et $i, j \in \{1, \ldots, |E|\}$ vérifiant $f(i) \in \Pi_t$ et $f(j) \in \Pi_{t+1}$:

$$m_{t,i,j} = m_{g(t),f^{-1}(h_t(f(i))),f^{-1}(h_t(f(j)))} = m'_{g(t),f'^{-1}(h_t(f(i))),f'^{-1}(h_t(f(j)))}$$

On en déduit, pour $k \in \{1, \ldots, |E|\}$ vérifiant $f(k) \in \Pi_{t+1}$:

$$D^k_{t+1} = \sum_{v_t \in \Pi_t} m'_{g(t),f'^{-1}(h_t(v_t)),f'^{-1}(h_t(f(k)))} \times red_t(D_t)_{f^{-1}(v_t)}$$

Et donc, pour $k \in \{1, \ldots, |E|\}$ vérifiant $f(k) \in \Pi_{t+1}$:

$$D^k_{t+1} = (red_t(D_t) \times M'_t)_{f^{-1}(h_t(f(k)))}$$

On aboutit donc à :

$$D_{t+1} = aug_t(red_t(D_t) \times M'_t)$$

CQFD.

Pour exploiter pleinement une réduction, on doit passer directement de la distribution réduite à l'instant t à celle réduite à l'instant $t+1$. Détaillons le processus de passage de l'instant t à l'instant $t+1$:

1.	d_t	$\rightarrow$ distribution à l'instant t représentée selon la réduction à l'instant t
2.	$d_t.M_t$	$\rightarrow$ distribution à l'instant $t+1$ représentée selon la réduction à l'instant t
3.	$aug_t(d_t.M_t)$	$\rightarrow$ distribution à l'instant $t+1$
4.	$red_{t+1}(aug_t(d_t.M_t))$	$\rightarrow$ distribution à l'instant $t+1$ représentée selon la réduction à l'instant $t+1$

Pour utiliser pleinement une réduction, il est nécessaire d'établir un moyen pour passer directement de l'étape 2 à l'étape 4.

Définition 11.5 *La fonction de passage tsf_t du modèle réduit à l'instant t au modèle réduit à l'instant $t+1$ est définie par :*

$$tsf_t(d) = red_{t+1}(aug_t(d))$$

Le théorème 11.2 permet de passer de l'étape 2 à l'étape 4 par un procédé dont la complexité dépend de $|E(\Pi, g, h)|$ et non de $|E|$. On obtient une méthode d'évaluation des distributions de X lorsque Q est vérifiée dont la complexité ne dépend que de $|E(\Pi, g, h)|$: la réduction est donc pleinement exploitée.

Théorème 11.2 *Soit d la distribution réduite à l'instant $t+1$ représentée selon la réduction à l'instant t, alors pour tout $k \in \{1, \ldots, |E(\Pi, g, h)|\}$, on a :*

$$f'(k) \in h_{t+1}(\Pi_{t+1}) \Rightarrow (tsf_t(d))_k = d_{f'^{-1}\left(h_t\left(h^{-1}_{t+1}(f'(k))\right)\right)}$$

Lorsque le modèle réduit à l'instant $t+1$ n'est pas le même que celui à l'instant t, le passage de l'étape 2 à l'étape 4 n'est pas trivial. La définition et le théorème suivants permettent d'optimiser le calcul de tsf_t dans certains cas.

Définition 11.6 *Une réduction (Π, g, h) effectue un saut à l'instant t s'il existe $e \in \Pi_{t+1}$ tel que $h_t(e) \neq h_{t+1}(e)$.*

Théorème 11.3 *Soit d la distribution réduite à l'instant $t+1$ représentée selon la réduction à l'instant t, si (Π, g, h) n'effectue pas de saut à l'instant $t+1$, alors pour tout $k \in \{1, \ldots, |E(\Pi, g, h)|\}$, on a :*

$$f'(k) \in h_{t+1}(\Pi_{t+1}) \Rightarrow (tsf_t(d))_k = d_k$$

11.2.2 Algorithme générique de calcul

Considérons les éléments suivants :
- D_t : la distribution à l'instant t,
- M : la matrice de transition,
- d, m : les éléments réduits correspondants.

On détermine la distribution à l'instant $t+1$ à partir de celle à l'instant t par le calcul suivant :

$$D_{t+1} = D_t \times M$$

En utilisant une réduction (Π, g, h), on effectue :

$$d_{t+1} = red_{t+1}(aug_t(d_t \times m)) = tsf_t(d_t \times m)$$

Lorsque la réduction (Π, g, h) n'effectue pas de saut à l'instant $t+1$, on a directement :

$$d_{t+1} = d_t \times m$$

L'algorithme générique pour calculer la distribution de probabilité de la chaîne de Markov X à l'instant t lorsque la contrainte Q est satisfaite est présentée sur la figure 11.2. La condition d'arrêt dépend du calcul effectué. Dans la suite, nous proposons plusieurs solutions.

11.3 Réduction aux premières instances

11.3.1 Principe de réduction utilisé

Notre technique de réduction exploite les interdictions imposées par les contraintes de validation. La réduction que nous proposons ici repose sur les contraintes temporelles des tâches. La figure 11.3 montre que tous les états possibles de la chaîne de Markov I_t^k ne sont pas valides. A chaque instant, on peut donc se contenter d'examiner un nombre restreint d'états. Cependant, la réduction que nous utilisons va plus loin : elle stipule aussi que le comportement de chaque instance est similaire à celui de la première instance. Cet argument est justifié par le fait que les différentes instances d'une même tâche exécutent les mêmes instructions : elles exécutent le même programme, seul le

```
creer_distribution(d1)
creer_distribution(d2)
d1[all] := 0
d2[all] := 0
d1[f(0,...,0)] := 1
t := 0
while(run) :
    d2 := d1 × m
    permut(d1, d2)
    d2[all] := 0
    si saut(t) alors :
        d2 := tsf_t(d1)
        permut(d1, d2)
        d2[all] := 0
    t := t + 1
    run := condition_arret()
detruire_distribution(d1)
detruire_distribution(d2)
```

FIG. 11.2 – Algorithme générique d'évaluation

contexte d'exécution change. Formellement, cette dernière propriété se retrouve dans les conditions de compatibilité avec cette réduction. Celle-ci ramène donc les états de toutes les instances d'une tâche à ceux de la première (voir figure 11.4). Pour la définir, nous devons construire un triplet (Π, g, h) correspondant à cette transformation. On obtient alors le triplet donné dans la définition suivante.

Définition 11.7 *La réduction aux premières instances est le triplet* (Π, g, h) *défini ainsi :*

- Π_t *est l'ensemble* $\Pi_t^1 \times \cdots \times \Pi_t^n$ *où*

$$\Pi_t^k = \begin{cases} \{\lfloor (t-r_k)/T_k \rfloor . C_k, \ldots, \lfloor (t-r_k)/T_k \rfloor . C_k + C_k\}, \textit{ si } t \geq r_k \\ \{0\}, \textit{ sinon} \end{cases}$$

- $g_k(t) = \begin{cases} r_k + (t-r_k) \bmod T_k, \textit{ si } t \geq r_k \\ t, \textit{ sinon} \end{cases}$
- $h_t(i_1, \ldots, i_n) = (i'_1, \ldots, i'_n)$, *où* $i'_k = \begin{cases} i_k - C_k . \lfloor (t-r_k)/T_k \rfloor, \textit{ si } t \geq r_k \\ i_k, \textit{ sinon} \end{cases}$

L'ensemble des états réduits correspondant à cette réduction est :

$$E(\Pi, g, h) = \{0..C_1\} \times \cdots \times \{0..C_n\}$$

Ainsi défini, le triplet (Π, g, h) effectue donc une réduction de l'ensemble $\{0, \ldots, \lambda\}^n$ dans l'ensemble $\{0, \ldots, C_1\} \times \cdots \times \{0, \ldots, C_n\}$, où λ est la durée d'exécution considérée. L'ensemble $E(\Pi, g, h)$ contient exactement $\prod_{k=1}^{n} (C_k + 1)$ états. La complexité spatiale de cette réduction est donc en $O(C^n)$.

11.3.2 Conditions de compatibilité

Pour que la réduction aux premières instances soit utilisable, elle doit vérifier les trois conditions de compatibilité définies dans la section 11.2.1. Les états supprimés par cette réduction ne vérifient pas les contraintes temporelles des tâches. Cette réduction est donc compatible avec les états

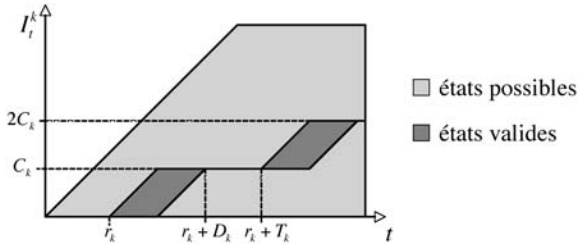


FIG. 11.3 – Etats valides et états possibles pour une tâche τ_k

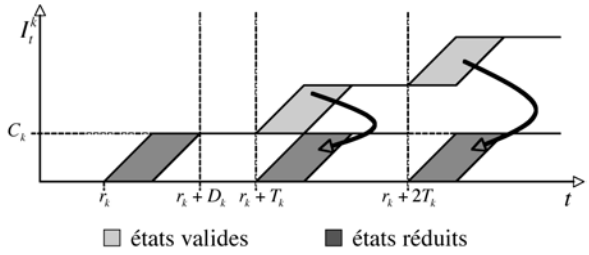


FIG. 11.4 – Réduction aux premières instances

supprimés. La condition de compatibilité avec le modèle dépend de la loi de probabilité choisie : cette condition est généralement vérifiée, en particulier avec la loi équiprobable.

La réduction aux premières instances caractérise le comportement périodique des tâches, elle permet alors de déterminer les contraintes compatibles avec ce comportement. Les contraintes sont exprimées à partir des données t, I_t et I_{t+1}. Cependant, une connaissance réduite de ces paramètres est souvent suffisante, par exemple : $g_1(t), \ldots, g_n(t)$ peut souvent remplacer t. Dans la suite de cette section, nous étudions les contraintes compatibles avec la réduction aux premières instances.

Considérons une contrainte markovienne Q et une fonction φ dépendant de t, I_t et I_{t+1}. On dit que Q ne dépend que de l'image des paramètres t, I_t et I_{t+1} fournie par φ si et seulement si on a :

$$\forall t \in \mathbb{N}, Q_t(I_t, I_{t+1}) \Leftrightarrow Q(\varphi(t, I_t, I_{t+1}))$$

Si l'image fournie par φ correspond aux paramètres réduits $g_1(t), \ldots, g_n(t)$ et $h(t, I_t)$, alors toute contrainte Q qui ne dépend que de l'image fournie par φ est compatible avec la réduction aux premières instances, puisque l'on a :

$$\forall t \in \mathbb{N}, Q_t(I_t, I_{t+1}) \Leftrightarrow Q(\varphi(t, I_t, I_{t+1})) \Leftrightarrow Q(g_1(t), \ldots, g_n(t), h_t(I_t), h_t(I_{t+1}))$$

Supposons que l'image fournie par φ contienne aussi la quantité $I_{t+1} - I_t$. Nous avons $I_{t+1} - I_t = h_t(I_{t+1}) - h_t(I_t)$. Donc, l'information $I_{t+1} - I_t$ est exprimable à partir de $h_t(I_t)$ et de $h_t(I_{t+1})$. Ainsi, toute contrainte qui dépend seulement de cette information est compatible avec la réduction aux premières instances. En raisonnant de manière similaire, on montre que les contraintes qui ne dépendent que des paramètres suivants sont compatibles avec la réduction aux premières instances :

- $I_{t+1} - I_t$,
- $(I_t^1 \bmod C_1, \ldots, I_t^n \bmod C_n)$ et $(I_{t+1}^1 \bmod C_1, \ldots, I_{t+1}^n \bmod C_n)$,
- $(I_t^1 - \Delta_1, \ldots, I_t^n - \Delta_n)$ et $(I_{t+1}^1 - \Delta_1, \ldots, I_{t+1}^n - \Delta_n)$,
  où $\Delta_k = C_k.\lfloor I_t^k / C_k \rfloor$.

Toutes les contraintes que nous avons exprimées sous la forme de contrainte markovienne dans le chapitre 10 sont donc compatibles avec la réduction aux premières instances. Remarquons que cette réduction est aussi utilisable avec le modèle $Y_t^1, \ldots, Y_t^n$ que nous avons défini pour le contexte préemptif avec migration partielle.

11.3.3 Etude des sauts

Considérons un instant $t \in \mathbb{N}$ et un état $e = (i_1, \ldots, i_n)$, et étudions les situations où la réduction aux premières instances effectue un saut. Pour une tâche τ_k et pour $t < r_k$, on a par définition :

$$h_t(e)_k = h_{t+1}(e)_k$$

Pour $t \geq r_k$, on obtient :

$$h_{t+1}(e)_k - h_t(e)_k = C_k.(\lfloor (t + 1 - r_k)/T_k \rfloor - \lfloor (t - r_k)/T_k \rfloor)$$

Or $\lfloor (t - r_k + 1)/T_k \rfloor \neq \lfloor (t - r_k)/T_k \rfloor$ si et seulement si $(t + 1 - r_k) \bmod T_k = 0$. Donc, la réduction aux premières instances effectue un saut à l'instant t si et seulement si au moins une tâche se réactive à cet instant-là.

Pour utiliser la réduction aux premières instances, nous devons définir une bijection de $\{1, \ldots, |E(\Pi, g, h)|\}$ dans $E(\Pi, g, h)$. Celle que nous utilisons permet de facilement faire correspondre les états réduits à l'instant $t - 1$ avec ceux à l'instant t lorsque la réduction effectue un saut.

Considérons la bijection f' de $\{1..|E(\Pi, g, h)|\}$ dans $E(\Pi, g, h)$ définie par :

- $\Delta_k = \prod_{i=1}^{k-1}(C_i + 1)$
- $f'^{-1}(i_1, ..., i_n) = \sum_{k=1}^{n}(i_k \times \Delta_k)$

Considérons un instant $t \in \mathbb{N}$ auquel la réduction aux premières instances effectue un saut. D'après ce qui précède, il existe au moins une tâche qui se réactive à cet instant. Notons R le

sous-ensemble de τ contenant les tâches qui se réactivent à l'instant t. Pour tout état $e \in E$, on obtient :

$$f'^{-1}(h_{t-1}(e)) - f'^{-1}(h_t(e)) = \sum_{\tau_k \in R} (C_k \times \Delta_k)$$

Cette propriété montre que la transformation à appliquer à la distribution réduite lors d'un saut revient à un simple décalage d'indices. On obtient alors le résultat suivant.

Théorème 11.4 *Soient $t \in \mathbb{N}$ un instant auquel la réduction aux premières instances effectue un saut et d la distribution réduite à l'instant t représentée selon la réduction à l'instant $t-1$. Pour tout $k \in \{1, \ldots, |E(\Pi, g, h)|\}$, on a :*

$$f'(k) \in h_t(\Pi_t) \Rightarrow (tsf_{t-1}(d))_k = d_{k'}, \text{ où } k' = k + \sum_{\tau_k \in R} (C_k \times \Delta_k)$$

où $R = \{\tau_k \in \tau | t \geq r_k \wedge (t - r_k) \bmod T_k = 0\}$.

Le décalage d'indice effectué dépend seulement de l'instant où se produit le saut. Ainsi, le terme $\sum_{\tau_k \in R}(C_k \times \Delta_k)$ est calculé une seule fois lors du traitement d'un saut.

11.3.4 Algorithme de passage de l'instant t à l'instant $t+1$

L'algorithme 11.2 présente la structure générale permettant d'utiliser une réduction. Dans cette section, nous énonçons l'algorithme réalisant l'opération "$d2 := d1 \times m'$" (voir figure 11.5).

Le terme $m'_{f'^{-1}(i), f'^{-1}(i+D)}$ correspond à la loi de probabilité choisie. Par exemple avec la loi équiprobable, sa valeur est $1/2^n$. En général, il est nécessaire d'utiliser deux distributions, l'une pour l'instant t que l'on accède en lecture, et l'autre pour l'instant $t+1$ que l'on accède en écriture. Cependant, la marche réalisée ici est croissante, ainsi en commençant par les plus grands éléments, on peut n'utiliser qu'une seule distribution. La figure 11.6 présente l'algorithme utilisé pour le calcul de "$d := d \times m'$" (noté $PAS_t(d)$ dans la suite).

11.3.5 Algorithme de gestion des sauts

Dans la section 11.3.3, nous avons montré que la réduction aux premières instances effectue un saut à l'instant t si et seulement si au moins une tâche se réactive à l'instant t. Par définition, une

```
parcours des états réduits i = (i_1, ..., i_n) vérifiant 0 ≤ i_k ≤ C_k :
   si d1[f'^{-1}(i)] > 0
   alors
      parcours des directions D = (d_1, ..., d_n) vérifiant 0 ≤ d_k ≤ 1 ∧ 0 ≤ i_k + d_k ≤ C_k :
         si Q_{g(t)}(i, i + D)
         alors d2[f'^{-1}(i + D)] += m'_{f'^{-1}(i), f'^{-1}(i+D)} × d1[f'^{-1}(i)]
```

FIG. 11.5 – Traitement des transitions sortant d'un état réduit

```
parcours décroissant des états réduits i = (i_1, ..., i_n) vérifiant 0 ≤ i_k ≤ C_k :
   si d[f'^{-1}(i)] > 0 alors
      tmp := d[f'^{-1}(i)]
      d[f'^{-1}(i)] := 0
      parcours des directions D = (d_1, ..., d_n) vérifiant 0 ≤ d_k ≤ 1 ∧ 0 ≤ i_k + d_k ≤ C_k :
         si Q_{g(t)}(i, i + D)
         alors d[f'^{-1}(i + D)] += m'_{f'^{-1}(i), f'^{-1}(i+D)} × tmp
```

FIG. 11.6 – Traitement des transitions sortant d'un état réduit en utilisant une seule distribution

tâche τ_k se réactive à l'instant t si et seulement si les deux conditions suivantes sont vérifiées :

$$(t > r_k) \wedge ((t - R_k) \bmod T_k = 0)$$

L'algorithme permettant de déterminer si un instant t est un instant de réactivation est présenté dans la figure 11.7. Dans la suite, on la note $REA(t)$.

Comme pour l'algorithme de passage de l'instant t à l'instant $t+1$, on peut réaliser un saut en utilisant seulement une distribution. Pour un instant de réactivation t, l'algorithme de calcul de "$d := tsf_t(d)$" est présenté sur la figure 11.8. Dans la suite, on le note $PRJ_t(d)$.

11.3.6 Algorithme réalisant la marche aléatoire correspondant à l'exécution d'un système temps-réel

En utilisant la structure générique que nous avons donnée dans la section 11.2.2 avec les algorithmes présentés dans les sections précédentes, on obtient l'algorithme de validation présenté dans la figure 11.9.

Cet algorithme utilise une seule distribution portant sur l'ensemble $E(\Pi, g, h)$, sa complexité spatiale est donc :

$$\prod_{k=1}^{n} (C_k + 1)$$

Etudions maintenant la complexité temporelle de cet algorithme :

Algorithme	Complexité au pire
$PAS_t(d)$	$D \times \prod_{k=1}^{n}(C_k + 1)$
$REA(t)$	n
$PRJ_t(d)$	$\prod_{k=1}^{n}(C_k + 1)$

où D est le nombre de directions

$saut := false$
$k := 1$
`tant que` $k \leq n \wedge \neg saut$
  `si` $(t > r_k) \wedge ((t - r_k) \bmod T_k = 0)$
  `alors` $saut := true$
  $k := k + 1$
`return saut`

FIG. 11.7 – Algorithme déterminant si la réduction à la laxité effectue un saut à l'instant t

$\Delta := 0$
`pour` k `de` 1 `à` n :
  `si` $(t > r_k)$ `et` $((t - r_k) \bmod T_k = 0)$
  `alors` $\Delta := \Delta + C_k \times \Delta_k$
`parcours de tous les états réduits` $i = (i_1, ..., i_n)$ `vérifiant`

$$\begin{cases} 0 \leq i_k \leq C_k \\ (t > r_k) \wedge ((t - r_k) \bmod T_k = 0) \Rightarrow i_k = C_k \end{cases}$$

$d[f'^{-1}(i) - \Delta] := d[f'^{-1}(i)]$
$d[f'^{-1}(i)] := 0$

FIG. 11.8 – Algorithme réalisant le saut effectué à l'instant t

La complexité au pire de notre algorithme est équivalente à :

$$O(L \times D \times C^n), \text{ où } L \text{ est la longueur de la marche aléatoire.}$$

Le nombre maximal de directions dépend du nombre de processeurs disponibles. On a :

$$D \leq \sum_{k=0}^{p} C_n^k \leq 2^n$$

Généralement, le terme D est de complexité polynomiale, par exemple : en monoprocesseur, on a $D = n + 1$, et en biprocesseur $D = \frac{n^2}{2} + \frac{n}{2} + 1$.

La longueur de l'intervalle d'étude d'un système temps-réel est généralement du même ordre de grandeur que P. La complexité au pire de notre algorithme est donc du même ordre de grandeur que :

En monoprocesseur	En biprocesseur	$\cdots$	En général
$n.P.C^n$	$n^2.P.C^n$	$\cdots$	$2^n.P.C^n$

11.4 Réduction à la laxité

11.4.1 Principe de réduction utilisé

La réduction à la laxité est un raffinement de la réduction aux premières instances. Nous tenons compte ici de l'ensemble des états possibles pour chaque instant t. Dans le cadre de la réduction aux premières instances, l'espace des états réduits est indépendant du temps et correspond à la première instance de chaque tâche, soit à l'ensemble $\{0, \ldots, C_1\} \times \cdots \times \{0, \ldots, C_n\}$. Dans le cadre de la réduction à la laxité, l'espace des états réduits dépend du temps et correspond exactement aux états permis par les contraintes temporelles des tâches (voir figure 11.3). La transformation temporelle g de la réduction à la laxité est la même que celle de la réduction aux premières instances. Seuls les paliers et la transformation spatiale diffèrent entre ces deux réductions.

Cette réduction repose sur les deux notions suivantes, à chaque instant et pour chaque tâche (voir figure 11.10) :

- il existe un état minimal pour chaque instant,
- les états possibles appartiennent à l'ensemble :

  { état minimal, ..., état minimal+laxité }

Tout d'abord, décrivons formellement l'état minimal de chaque tâche à un instant donné.

Définition 11.8 *Pour chaque tâche τ_k, et chaque instant t, on définit l'état minimal, noté $bas_t(k)$, par :*

- *si* $0 \leq g_k(t) \leq r_k + D_k - C_k$, *alors* $bas_t(k) = 0$

```
creer_distribution(d)
d[all] := 0
d[f'^{-1}(0,...,0)] := 1
t := 0
while(run) :
    PAS_t(d)
    si REA(t+1) alors PRJ_{t+1}(d)
    t := t + 1
    run := condition_arret()
detruire_distribution(d)
```

FIG. 11.9 – Algorithme de validation

- *si* $r_k + D_k - C_k \leq g_k(t) \leq r_k + D_k$, *alors* $bas_t(k) = C_k - (r_k + D_k - g_k(t))$
- *si* $r_k + D_k \leq g_k(t) < r_k + T_k$, *alors* $bas_t(k) = C_k$

On a aussi :

$$bas_t(k) = min\{C_k, max\{0, C_k - (r_k + D_k - g_k(t))\}\}$$

La notion d'état maximal d'une tâche à un instant donné est donnée par la définition suivante.

Définition 11.9 *Pour chaque tâche* τ_k, *et chaque instant* t, *on définit l'état maximal, noté* $top_t(k)$, *par :*

- *si* $0 \leq g_k(t) \leq r_k$, *alors* $top_t(k) = 0$
- *si* $r_k \leq g_k(t) \leq r_k + C_k$, *alors* $top_t(k) = g_k(t) - r_k$
- *si* $r_k + C_k \leq g_k(t) \leq r_k + T_k$, *alors* $top_t(k) = C_k$

Remarquons que l'on a aussi :

$$top_t(k) = min\{C_k, max\{0, g_k(t) - r_k\}\}$$

Nous définissons la laxité d'une tâche à un instant donné par la différence entre l'état maximal et l'état minimal :

$$lax_t(k) = top_t(k) - bas_t(k)$$

La réduction que l'on utilise ici ne considère que les états compris entre $bas_t(k)$ et $top_t(k)$, au lieu de représenter tous les états dans l'intervalle $\{0..C_k\}$ comme le fait la réduction aux premières instances. Munis des notions précédentes, nous pouvons maintenant définir la réduction à la laxité.

Définition 11.10 *La réduction à la laxité est le triplet* (Π, g, h) *défini ainsi :*

- $\Pi_t = \Pi_t^1 \times \cdots \times \Pi_t^n$, *où chaque* Π_t^k *est égal à* $\{bas_k(t), \ldots, top_k(t)\}$
- $g_k(t) = \begin{cases} r_k + (t - r_k) \bmod T_k, & \text{si } t \geq r_k \\ t, & \text{sinon} \end{cases}$
- $h_t(i_1, \ldots, i_n) = (i_1 - bas_1(t), \ldots, i_n - bas_n(t))$

La réduction à la laxité colle exactement aux états permis par la validité temporelle (voir la figure 11.11). Dans certains cas particuliers (voir la figure 11.12), cette réduction permet même de représenter une tâche τ_k par moins de $C_k + 1$ états.

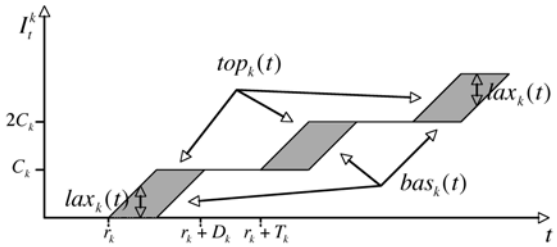


FIG. 11.10 – Fonctions $bas_t(k)$, $top_t(k)$ et $lax_t(k)$ d'une tâche

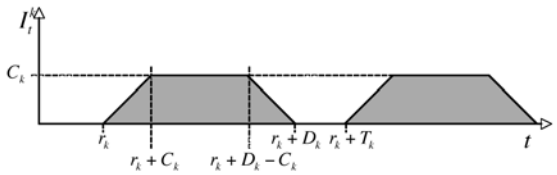


FIG. 11.11 – Réduction à la laxité

11.4.2 Conditions de compatibilité

Comme pour la réduction aux premières instances, nous devons montrer que la réduction à la laxité vérifie les trois conditions de compatibilité définies dans la section 11.2.1. Tout d'abord, les états supprimés par cette réduction sont ceux qui ne vérifient pas les contraintes temporelles des tâches. Cette réduction est donc compatible avec les états supprimés. La condition de compatibilité avec le modèle dépend de la loi de probabilité choisie. Toutefois, cette condition est généralement vérifiée, en particulier avec la loi équiprobable.

On montre de la même manière que pour la réduction aux premières instances que les contraintes dépendant seulement des paramètres suivants sont compatibles avec la réduction à la laxité :

- $I_{t+1} - I_t$,
- $(I_t^1 \bmod C_1, \ldots, I_t^n \bmod C_n)$ et $(I_{t+1}^1 \bmod C_1, \ldots, I_{t+1}^n \bmod C_n)$,
- $(I_t^1 - \Delta_1, \ldots, I_t^n - \Delta_n)$ et $(I_{t+1}^1 - \Delta_1, \ldots, I_{t+1}^n - \Delta_n)$,
  où $\Delta_k = C_k.\lfloor I_t^k / C_k \rfloor$.

Toutes les contraintes que nous avons exprimées sous la forme de contrainte markovienne dans le chapitre 10 sont donc compatibles avec la réduction à la laxité. Comme la réduction aux premières instances, celle à la laxité est utilisable avec le modèle $Y_t^1, \ldots, Y_t^n$ correspondant au contexte préemptif avec migration partielle.

11.4.3 Etude des sauts

Les sauts effectués par la réduction à la laxité dépendent de la fonction $bas_k(t)$. Lorsque celle-ci change de valeur entre deux instants t et $t+1$, alors la réduction à la laxité effectue un saut à l'instant $t+1$. Ainsi, cette réduction effectue un saut à l'instant t si et seulement si :

$$\exists k \in \{1, \ldots, n\}, r_k + D_k - C_k < g_t(k) \leq r_k + D_k$$

Les sauts de cette réduction sont donc différents de ceux de la réduction en première instance, en particulier la réduction à la laxité ne fait pas de saut aux instants $r_k + T_k.\mathbb{N}$.

Pour exploiter cette réduction, nous n'utilisons pas l'ensemble $E(\Pi, g, h)$. Celui-ci peut toutefois contenir moins de $\prod_{k=1}^{n}(C_k + 1)$ éléments (voir figure 11.12). Ainsi, cette réduction offre déjà un gain par rapport à celle aux premières instances. Toutefois, pour minimiser la complexité spatiale correspondant à cette réduction, nous utilisons l'ensemble Π_t propre à chaque instant t plutôt que l'ensemble $E(\Pi, g, h)$ (propre à tous les instants).

Considérons la bijection de $\{1, \ldots, |\Pi_t|\}$ dans Π_t définie ainsi :

- $\Delta_k = \prod_{l=1}^{k-1}(top_t(k) - bas_t(k) + 1)$
- $f'^{-1}(i_1, ..., i_n) = \sum_{k=1}^{n}((i_k - bas_t(k)) \times \Delta_k)$

Cette bijection permet d'utiliser à chaque instant une distribution dont la taille est parfaitement adaptée aux états réduits. La complexité spatiale engendrée est donc minimisée. Cependant, lorsque le cardinal de Π_t varie entre deux instants, les distributions correspondant aux états réduits à ces deux instants n'ont pas le même format (selon f'). Ainsi, contrairement à la réduction aux premières instances, on ne peut pas toujours utiliser une seule distribution pour implémenter l'algorithme de calcul. Plus précisément, le cardinal de Π_t dépend des fonctions $lax_k(t)$, il varie donc seulement aux instants t pour lesquels il existe une tâche τ_k vérifiant l'une des deux conditions suivantes :

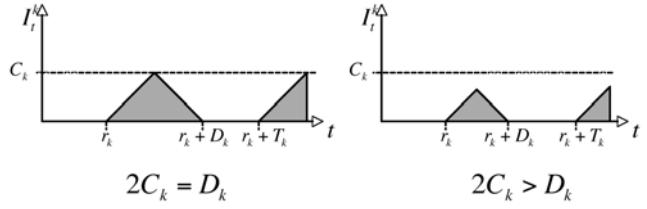


FIG. 11.12 – Cas particulier de la réduction à la laxité

- $r_k < g_t(k) \leq r_k + C_k$
- $r_k + D_k - C_k < g_t(k) \leq r_k + D_k$

On en déduit qu'aux autres instants, on peut calculer la distribution à l'instant $t+1$ à partir de celle à l'instant t en utilisant une seule distribution.

Toutefois, aux instants où l'une des fonctions $bas_k(t)$ varie, la réduction à la laxité effectue un saut, ces instants t sont ceux vérifiant :

$$\exists k \in \{1, \ldots, n\}, r_k + D_k - C_k < g_k(t) \leq r_k + D_k$$

D'après ce qui précède, ces instants correspondent aussi à des variations des fonctions $lax_k(t)$. Or, on est obligé d'utiliser deux distributions pour réaliser le calcul du passage de la distribution de l'instant t à celle de l'instant $t+1$. La deuxième distribution étant par construction au format correspondant à l'instant $t+1$, il n'est pas nécessaire de réaliser de saut à ces instants-là : ils sont implicitement pris en compte.

11.4.4 Algorithme réalisant la marche aléatoire correspondant à l'exécution d'un système temps-réel

L'implémentation que nous réalisons ne nécessite pas de gestion des sauts. Nous devons seulement tenir compte des instants où les paliers Π_t évoluent. A ces instants-là, nous devons utiliser deux distributions différentes.

L'algorithme de calcul de "$d2 := d1 \times m'$" est présenté sur la figure 11.13. Dans la suite, on le note $PAS_t(d1, d2)$. Pour les instants t vérifiant $\Pi_t = \Pi_{t+1}$, on peut utiliser une version de l'algorithme $PAS_t(d1, d2)$ utilisant seulement une distribution. Cet algorithme est présenté sur la figure 11.14. Dans la suite, on le note $PAS_t(d)$. On obtient alors l'algorithme de validation présenté par la figure 11.15.

La complexité spatiale au pire de cet algorithme est en $O(C^n)$. Toutefois en moyenne, lors de

`parcours des états` $i = (i_1, ..., i_n)$ `vérifiant` $bas_t(k) \leq i_k \leq top_t(k)$:
  `si` $d1[f'^{-1}(i_1, ..., i_n)]$
  `alors parcours des directions` $D = (d_1, ..., d_n)$ `vérifiant`

$$\begin{cases} 0 \leq d_k \leq 1 \\ bas_{t+1}(k) \leq i_k + d_k \leq top_{t+1}(k) \end{cases}$$

    `si` $Q_{g(t)}(i, i+D)$
    `alors` $d2[f'(i+D)] += m'_{f'^{-1}(i), f'^{-1}(i+D)} \times d1[f'^{-1}(i)]$

FIG. 11.13 – Traitement des transitions sortant d'un état réduit

`parcours décroissant des états` $i = (i_1, ..., i_n)$ `vérifiant` $bas_t(k) \leq i_k \leq top_t(k)$:
  `si` $d[f'^{-1}(i)] > 0$ `alors`
    $tmp := d[f'^{-1}(i)]$
    $d[f'^{-1}(i)] := 0$
    `parcours des directions` $D = (d_1, ..., d_n)$ `vérifiant`

$$\begin{cases} 0 \leq d_k \leq 1 \\ bas_{t+1}(k) \leq i_k + d_k \leq top_{t+1}(k) \end{cases}$$

      `si` $Q_{g(t)}(i, i+D)$
      `alors` $d[f'^{-1}(i+D)] += m'_{f'^{-1}(i), f'^{-1}(i+D)} \times tmp$

FIG. 11.14 – Traitement des transitions sortant d'un état réduit en utilisant une seule distribution

chaque période, le nombre moyen d'états réduits pour chaque tâche τ_k est :

$$\frac{(D_k - C_k + 1) \times (C_k + 1) + T_k - D_k - 1}{T_k}$$

Ainsi, la complexité spatiale moyenne de cet algorithme est équivalente à $0((D-C)^n(C/T)^n)$. Remarquons que le terme C/T est l'ordre de grandeur de la charge des tâches, il est généralement strictement inférieur à 1. Ainsi, le facteur qui engendre véritablement de la complexité est la laxité des tâches $(D-C)$.

De la même manière que pour la réduction aux premières instances, on montre que la complexité temporelle moyenne de cet algorithme pour l'étude des systèmes temps-réel est équivalente à $O(2^n.P.(D-C)^n.(C/T)^n)$. La métapériode P étant toujours inférieure au produit des périodes des tâches, la complexité moyenne est donc inférieure à :

$$O(2^n.C^n.(D-C)^n)$$

Rappelons que pour les systèmes monoprocesseurs, le terme 2^n peut être changé par n, et pour les systèmes biprocesseurs par n^2.

11.5 Expérimentations

Avant de mettre en œuvre les capacités probabilistes de notre modèle, nous avons évalué ses performances algorithmiques. Pour cela, nous l'avons tout d'abord comparé au modèle géométrique défini dans [62, 64]. Ensuite, nous avons étudié le rapport entre le temps de calcul utilisé par notre méthode et la formule donnant la complexité temporelle moyenne que nous avons établie dans la section 11.4.4. Finalement, nous avons déterminé le gain apporté par la réduction à la laxité par rapport à la réduction aux premières instances.

Pour comparer notre méthode avec le modèle géométrique, nous avons généré un ensemble de 100 systèmes de tâches indépendantes destinés à une cible monoprocesseur. Les paramètres des tâches sont choisis aléatoirement selon la méthode suivante :

1. le paramètre C_k est choisi uniformément dans l'intervalle $\{1, \ldots, 50\}$,
2. le paramètre T_k est choisi uniformément dans l'intervalle $\{C_k, \ldots, 50\}$,
3. le paramètre D_k est choisi uniformément dans l'intervalle $\{C_k, \ldots, T_k\}$,

```
creer_distribution(0, d1)
d1[all] := 0
d1[f(0,...,0)] := 1
t := 0
while(run) :
   si ∃k ∈ {1,...,n}, r_k < g_t(k) ≤ r_k + C_k ou r_k + D_k − C_k < g_t(k) ≤ r_k + D_k
   alors
      creer_distribution(t + 1, d2)
      d2[all] := 0
      PAS_t(d1, d2)
      detruire_distribution(d1)
      d1 := d2
   sinon PAS_t(d1)
   t := t + 1
   run := condition_arret()
detruire_distribution(d1)
```

FIG. 11.15 – Algorithme de validation

4. le paramètre r_k vaut toujours 0.

Le nombre moyen de tâches composant un des systèmes générés est approximativement 3, la métapériode est toujours inférieure à 1000. La figure 11.16 indique les temps de calcul en seconde utilisés par notre méthode (VLD) et le modèle géométrique (GEO). Le temps de calcul utilisé par la méthode géométrique varie autour de 1s, alors que celui utilisé par notre méthode varie autour de 0,01s. En terme de complexité temporelle, notre méthode apporte donc un gain de facteur 100.

Pour chacune des expérimentations suivantes, nous avons généré aléatoirement 100 systèmes de tâches de la même manière que précédemment. Toutefois, le paramètre C_k est choisi entre 1 et 20, et le paramètre T_k est choisi entre C_k et 100. De plus, les systèmes de tâches ne sont plus à départs simultanés, le paramètre r_k est choisi entre 0 et T_k. Les systèmes générés contiennent 5 tâches en moyenne. Pour déterminer la durée d'étude nécessaire, nous avons utilisé les résultats obtenus dans le chapitre 6.

Dans la section 11.4.4, nous avons établi une formule correspondant la complexité temporelle moyenne utilisée par notre algorithme. Pour évaluer la précision de cet indicateur, nous avons déterminé le rapport entre le temps de calcul en seconde utilisé par notre méthode et la valeur fournie par cette formule (voir figure 11.17). Ce rapport évolue entre 10^{-7} et 10^{-8}. Il est intéressant de remarquer que ce rapport est relativement stable, notre formule fournit donc un indicateur significatif du temps de calcul.

Finalement, nous avons étudié la complexité spatiale utilisée par notre méthode. La figure 11.18 indique le rapport entre le maximum d'états alloués lorsque l'on utilise la réduction à la laxité et le maximum théorique possible ($\prod_{k=1}^{n}(C_k + 1)$). Ce nombre correspond à la complexité spatiale de la réduction aux premières instances, il permet donc de comparer les deux réductions que nous avons proposées : le gain apporté par celle à la laxité oscille généralement entre 10 et 20. D'autre part, la complexité spatiale du modèle géométrique est $P.\prod_{k=1}^{n}(C_k+1)$, ce modèle consomme donc beaucoup plus de mémoire que notre méthode.

Pour évaluer la qualité de la réduction à la laxité par rapport à l'optimal, nous avons évalué le nombre maximum d'états atteints à un instant donné. Nous l'avons ensuite comparé avec le

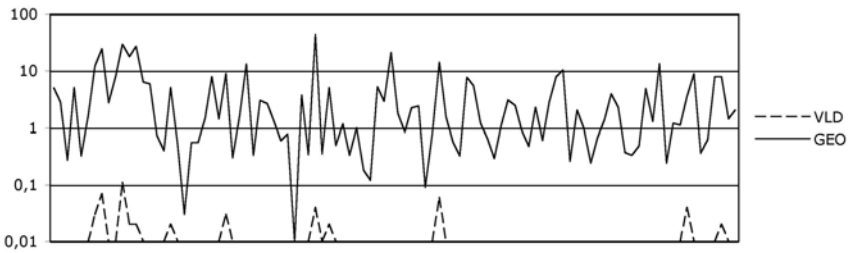


FIG. 11.16 – Comparaison entre le modèle basé sur les chaînes de Markov et celui basé sur la géométrie discrète

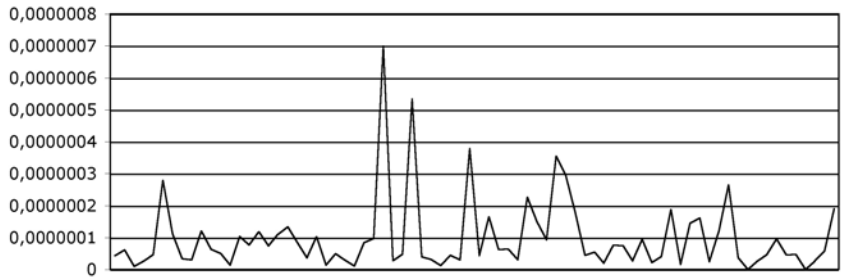


FIG. 11.17 – Rapport entre le temps de calcul et la complexité moyenne

nombre maximum d'états réduits selon la réduction à la laxité. La figure 11.19 indique le rapport entre ces deux valeurs pour un échantillon de 100 systèmes de tâches. On constate que ces deux quantités restent proches l'une de l'autre (autour de 0,8). Ainsi, la réduction à la laxité fournit une représentation efficace des états possibles.

11.6 Conclusion

Nous avons défini deux réductions permettant de ramener notre modèle à un nombre d'états finis. Grâce à elles, on a alors pu définir un algorithme permettant de calculer la probabilité que les contraintes intégrées par conjonction soient vérifiées sachant que celles intégrées par condition le sont. Cette méthode permet alors de calculer des caractéristiques probabilistes sur l'ordonnancement d'un système de tâches. Dans le chapitre suivant, nous l'utilisons pour calculer les distributions de probabilité du temps de réponse de chaque instance.

Nous avons aussi évalué la complexité théorique de notre algorithme. Le résultat que nous avons obtenu s'est révélé proche du temps de calcul réel. Il constitue donc un bon indicateur pour déterminer si un système de tâches est analysable par notre approche du point de vue complexité. Ce type d'indicateur est intéressant puisqu'il permet de prévoir le temps de calcul de notre algorithme. Les comparaisons que nous avons menées avec le modèle géométrique [62, 64] montrent que la complexité tant spatiale que temporelle de notre algorithme est plus intéressante.

Les travaux menés dans les chapitres 10 et 11 ont fait l'objet de deux publications, l'une dans la revue Technique et Science Informatiques, et l'autre dans les actes des Rencontres des Jeunes Chercheurs en Informatique Temps Réel 2005.

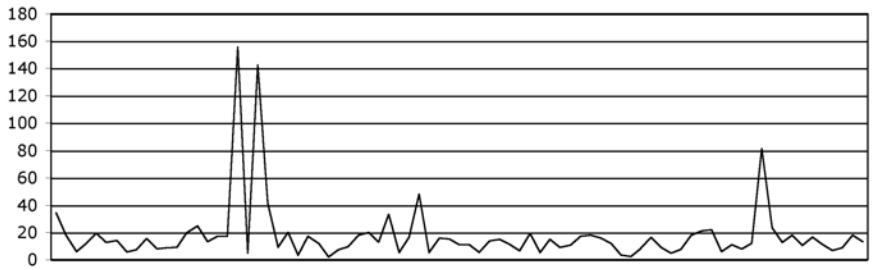


FIG. 11.18 – Gain apporté par la réduction à la laxité par rapport à la réduction aux premières instances

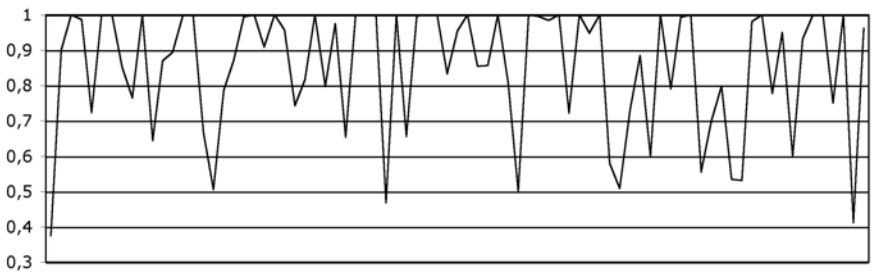


FIG. 11.19 – Rapport entre le maximum d'états utilisés et le maximum d'états alloués

Chapitre 12

Etude des temps de réponse

12.1 Introduction

Dans le chapitre précédent, nous avons développé une méthode de calcul de la probabilité qu'un système de tâches vérifie une propriété exprimée sous la forme d'une contrainte markovienne. Nous utilisons maintenant cette méthode pour évaluer la probabilité qu'une instance ait un temps de réponse donné. Plus précisément, on calcule la distribution de probabilité du temps de réponse de chaque instance. A partir de cette information, on peut déduire d'autres indicateurs de qualité comme la gigue par exemple.

La loi de probabilité utilisée pour représenter le comportement des tâches permet de sélectionner certains ordonnancements selon plusieurs critères. Par exemple, en fonction des caractéristiques de l'exécutif temps-réel ciblé, on définit une loi de probabilité adaptée, grâce à elle on obtient des informations probabilistes sur le comportement du système de tâches qui sont propres à son fonctionnement sur cet exécutif. Cette approche ne repose sur aucune politique d'ordonnancement particulière, elle permet donc d'obtenir un diagnostic sur le comportement des tâches en fonction de l'exécutif utilisé. De cette manière, on peut faire ressortir des propriétés propres à la spécification du système de tâches. En particulier, on peut évaluer si un système de tâches est facilement, ou au contraire, difficilement ordonnançable.

Dans la section 12.2, nous adaptons la méthode que nous avons développée dans le chapitre précédent pour calculer les distributions de probabilité du temps de réponse des instances. Dans les sections 12.3 et 12.4, nous déterminons une loi de probabilité représentant le comportement temporel des tâches. Dans les sections suivantes (12.5 à 12.7), nous lui intégrons les propriétés caractérisant les exécutifs temps-réel : préemptivité, conservativité, etc. Parallèlement, nous comparons les différentes lois que nous proposons à l'aide d'un exemple.

12.2 Intégration du calcul des temps de réponse

Pour cette section, on adopte les notations suivantes :
- E : l'ensemble des valeurs de $I_t = (I_t^1, \ldots, I_t^n)$,
- f : une bijection de E dans $\{1, \ldots, |E|\}$,
- D_t : la distribution de probabilité de la variable I_t.

Pour déterminer les distributions de probabilité du temps de réponse des instances des tâches, nous utilisons le modèle présenté dans les deux chapitres précédents. A partir de lui, nous définissons la variable aléatoire du temps de réponse d'une instance.

Définition 12.1 *On appelle $TR_{k,i}$ la variable aléatoire du temps de réponse de la $i^{\text{è}}$ instance de la $k^{\text{è}}$ tâche :*

$$TR_{k,i} = t \Leftrightarrow I_{t-1}^k = i.C_k - 1 \wedge I_{t-1}^k = i.C_k$$

Lorsque toutes les échéances des tâches sont respectées, on a :

$$\mathbb{P}(TR_{k,i} = t) > 0 \Leftrightarrow r_{k,i} + C_k \leq t \leq r_{k,i} + D_k$$

Les valeurs possibles du temps de réponse de chaque instance $\tau_{k,i}$ appartiennent donc à un intervalle de longueur $D_k - C_k$. Puisque nous considérons uniquement des valeurs entières, il y a donc $D_k - C_k + 1$ valeurs possibles.

La variable $TR_{k,i}$ dépend de la probabilité des transitions de la chaîne de Markov I_t^k. Plus précisément, on a :

$$\mathbb{P}(TR_{k,i} = t) = \sum_{\substack{v = (v_0, \ldots, i.C_k - 1, \ldots, v_n) \in \mathbb{N}^n \\ v' = (v'_0, \ldots, i.C_k, \ldots, v'_n) \in \mathbb{N}^n}} D_{t-1}^{f(v)} \times \mathbb{P}(I_t = v' | I_{t-1} = v)$$

Cette formule permet d'intégrer le calcul de la distribution de probabilité du temps de réponse de chaque instance aux algorithmes présentés dans le chapitre 11. Nous avons réalisé cette intégration pour l'algorithme reposant sur la réduction à la laxité (voir section 11.4.4).

Tout d'abord, nous ajoutons une variable `tdr` permettant de calculer la probabilité pour que l'instance courante de chaque tâche termine son exécution à l'instant courant. Cette variable est naturellement implémentée par un tableau à n éléments.

Ensuite, on doit intégrer le calcul de `tdr` dans les algorithmes $PAS_t(d1, d2)$ et $PAS_t(d)$. Cela revient à introduire dans l'algorithme $PAS_t(d1, d2)$ la formule de $TR_{k,i}$ donnée ci-dessus. Lorsque la condition `si` $Q_{g(t)}(i, i + D)$ est vérifiée, on ajoute alors le traitement suivant :

`si` $i_k + bas_t k = C_k - 1 + \lfloor (t - r_k)/T_k \rfloor$ `et` $d_k = 1$
`alors tdr[k] +=` $m'_{f'^{-1}(i), f'^{-1}(i+D)} \times d1[f'^{-1}(i)]$

L'algorithme $PAS_t(d)$ se modifie de la même manière. Ces instructions permettent de calculer la probabilité que l'instance courante de chaque tâche termine son exécution à l'instant courant. A la fin de l'exécution de l'algorithme PAS_t, la valeur de `tdr` vérifie :

$$\forall k \in \{1, \ldots, n\}, tdr[k] = \mathbb{P}(TR_{k, \lfloor (t - r_k)/T_k \rfloor} = t)$$

Dans le chapitre 10, nous avons indiqué que l'on peut intégrer une contrainte de deux manières, soit par conjonction, soit par condition. Supposons que CNJ représente les contraintes prises en compte par conjonction, et CND celles prises en compte par condition. Les distributions de probabilité que l'on obtient alors avec notre méthode correspondent à :

$$\mathbb{P}(I_t = i_t \wedge CNJ | CND)$$

Les contraintes intégrées par condition interviennent directement sur les probabilités de transition $\mathbb{P}(I_t = i_t | I_{t-1} = i_{t-1})$. Elles permettent donc d'établir la matrice de transition. Celles intégrées par conjonction interviennent a posteriori sur la matrice de transition : les probabilités associées aux transitions invalides sont annulées (voir section 10.5.2). Il en découle que la somme de la probabilité de toutes les transitions sortant d'un état n'est plus obligatoirement 1. En conséquence, la somme des probabilités de la distribution de I_t n'est plus forcément 1. La différence indique la probabilité pour les conditions CNJ de ne pas être vérifiées sachant que les contraintes CND le sont. Nous avons basé nos expérimentations sur l'algorithme reposant sur la réduction à la laxité. Cette réduction impose que les échéances des tâches soient respectées : elle correspond à une prise en compte par conjonction.

12.3 Loi équiprobable

Notre objectif est de définir une loi de probabilité caractérisant l'impact des interactions entre les tâches en fonction de l'exécutif temps-réel ciblé. Avant de chercher à élaborer une loi complexe,

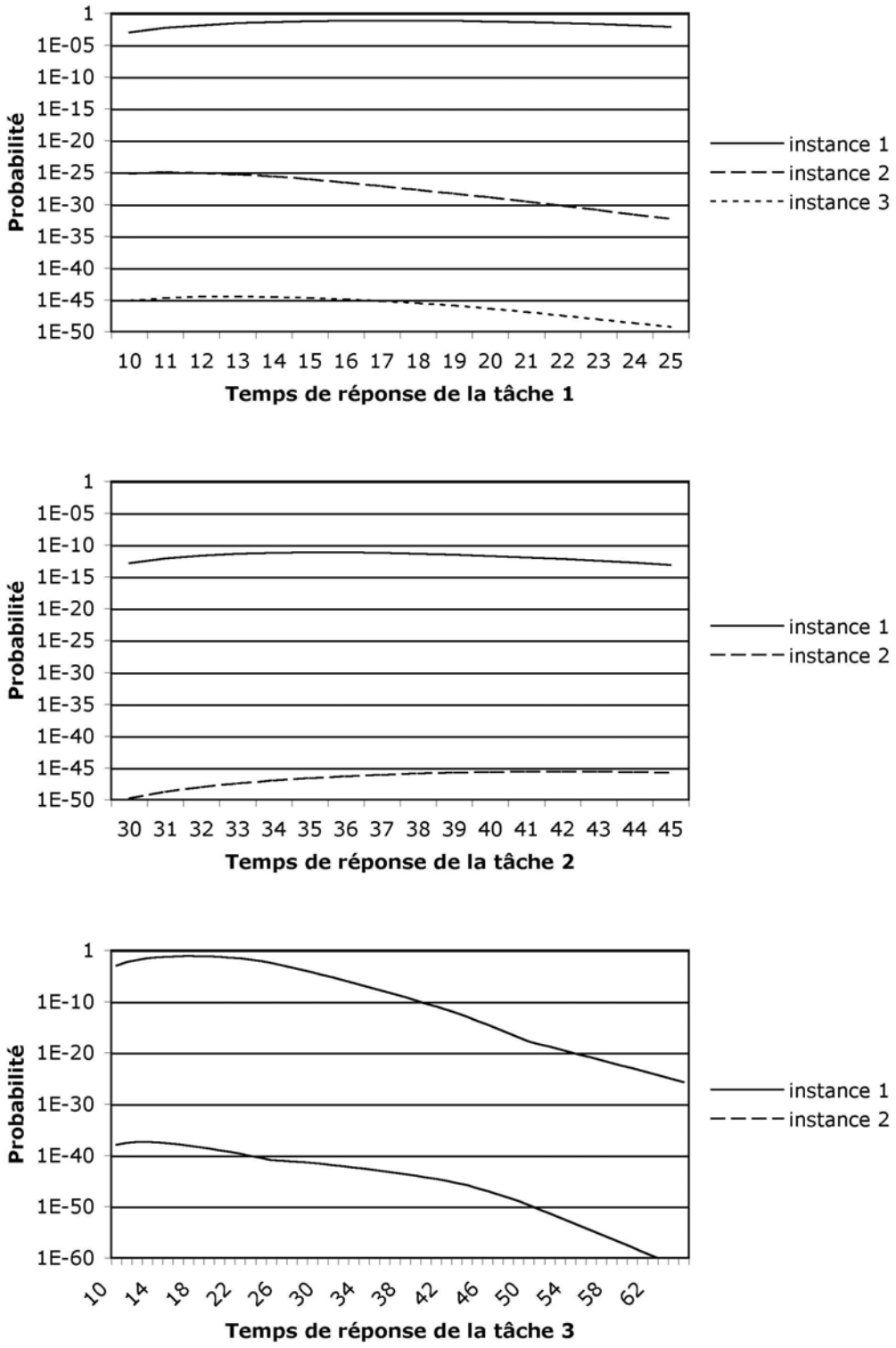


FIG. 12.1 – Distribution de probabilité du temps de réponse de chaque instance

nous avons expérimenté la loi équiprobable : tous les ordonnancements (valides ou non) ont la même probabilité. Dans cette section, nous supposons que les tâches sont indépendantes au sens probabiliste du terme. Elles n'interagissent donc pas entre elles directement (communication ou synchronisation) ou indirectement par le biais de l'exécutif temps-réel. Cela revient à supposer que chaque tâche possède son propre processeur. D'autre part, la méthode que nous utilisons repose sur la réduction à la laxité. Ainsi, la contrainte due aux échéances des tâches est implicitement prise en compte par conjonction. Les probabilités que nous déterminons correspondent donc à la probabilité que les échéances soient respectées sachant que tous les ordonnancements possibles (valides ou non) sont équiprobables.

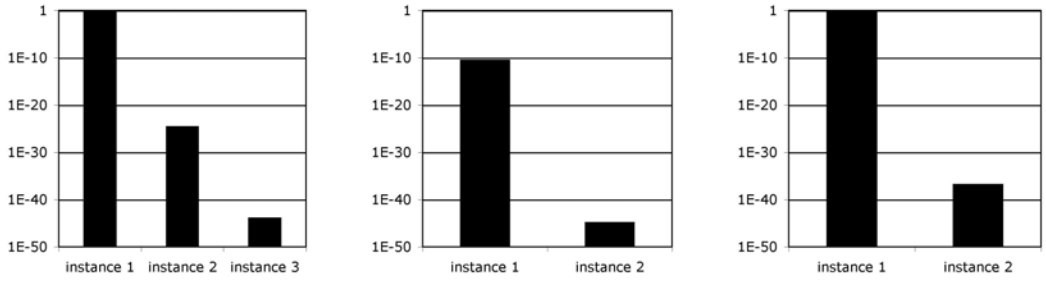
Dans ce contexte, les variables $(M_t^k)_{t\in\mathbb{N},k\in\{1,\ldots,n\}}$ sont indépendantes. En choisissant la loi équiprobable, on obtient les probabilités de transitions suivantes :

- $\mathbb{P}(I_{t+1}^k = x | I_t^k = x) = \frac{1}{2}$
- $\mathbb{P}(I_{t+1}^k = x+1 | I_t^k = x) = \frac{1}{2}$

Pour évaluer expérimentalement les lois de probabilité que nous définissons, nous utilisons le système de tâches indépendantes suivant :

	r_k	C_k	D_k	T_k
τ_1	0	10	25	50
τ_2	0	30	45	75
τ_3	0	10	65	75

La charge de ce système de tâches est $0,73$. Les tâches composant ce système de tâches sont à départs simultanés, la durée d'étude nécessaire est donc $PPCM(50,75,75) = 150$. Ainsi, la tâche τ_1 engendre trois instances par métapériode, et les tâches τ_2 et τ_3 en engendrent deux chacune.



Tâche τ_1 Tâche τ_2 Tâche τ_3

FIG. 12.2 – Probabilité de respect des échéances

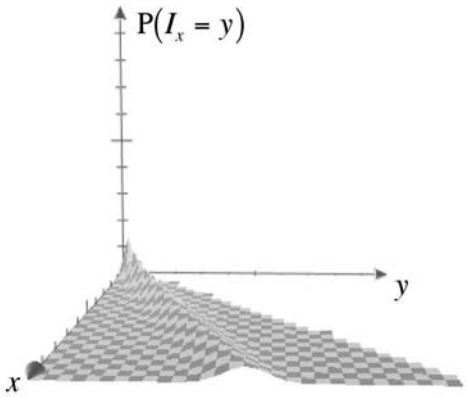


FIG. 12.3 – Distribution de probabilité du temps de réponse avec la loi équiprobable

En appliquant notre méthode avec la loi équiprobable à ce système de tâches, on obtient les probabilités de temps de réponse présentées sur la figure 12.1. On remarque que le temps de réponse d'une instance $\tau_{k,i}$ est à peu près uniformément distribué entre les valeurs $r_{k,i} + C_k$ et $r_{k,i} + D_k$, excepté pour les instances de τ_3.

La figure 12.2 indique la probabilité que l'échéance d'une instance soit respectée. Cette valeur correspond à la probabilité que le temps de réponse appartienne à l'intervalle $[r_{k,i} + C_k, r_{k,i} + D_k]$: les histogrammes de la figure 12.2 correspondent à l'intégrale des courbes de la figure 12.1. Pour la première instance de chaque tâche, la probabilité de respect de l'échéance est proche de 1, c'est normal puisque les tâches sont indépendantes. Toutefois, cette probabilité décroît très fortement pour les instances suivantes bien que les courbes aient la même topologie. Ce phénomène est dû au fait que la contrainte due au respect des échéances soit prise en compte par conjonction : la somme de la probabilité des transitions sortant d'un état peut être strictement inférieure à 1. Ce phénomène est gênant pour l'étude de gros systèmes : comment interpréter le fait que la probabilité de réussite des dernières instances des tâches soit de l'ordre de 10^{-50} ?

Un autre point remarquable concerne la première instance de la tâche τ_2. Sa probabilité de réussite est de l'ordre de 10^{-10}, alors que les probabilités de réussite de la première instance de τ_1 et de τ_3 sont proches de 1. Cette différence est due au rapport C_k/D_k. Avec la loi équiprobable, les distributions de probabilité que l'on obtient sont des gaussiennes centrées sur la droite d'équation $y = x/2$ et tronquées par la fenêtre $\{C_k, \ldots, D_k\}$ (voir figure 12.3). Donc, lorsque le rapport C_k/D_k est inférieur à $1/2$, la probabilité de réussite de la première instance de τ_k est forte ; dans le cas contraire $2C_k > D_k$, elle est faible.

Cette loi de probabilité présente donc plusieurs inconvénients pour notre approche. Premièrement, bien que les tâches soient indépendantes, la probabilité de respecter chaque échéance n'est pas 1. Le comportement temporel des tâches n'est donc pas représenté par cette loi. Deuxièmement, l'ordre de grandeur des probabilités obtenues décroît fortement entre chaque instance d'une même tâche. Pour un système de tâches complexes, on est alors amené à manipuler des quantités extrêmement petites dont l'interprétation paraît délicate.

12.4 Tâches indépendantes

Les marches aléatoires obtenues avec la loi équiprobable ne traduisent pas le comportement temporel des tâches. Pour y remédier, nous intégrons par condition les deux propriétés suivantes :

- une tâche inactive ne peut pas être exécutée,
- une tâche dont la laxité est nulle doit être exécutée.

Avec ces deux propriétés, la loi de probabilité dirige la marche aléatoire à l'intérieur des états admis par la validité temporelle : toutes les transitions qui amènent une tâche à ne pas respecter son échéance ont une probabilité nulle. Comme pour la loi équiprobable, nous considérons que les tâches sont indépendantes au sens probabiliste du terme.

Considérons un état $e = (e_1, \ldots, e_n)$ valide à l'instant t et déterminons les probabilités des transitions partant de e sachant que les deux propriétés ci-dessus sont vérifiées. Pour cela, adoptons les deux notations suivantes :

- l'ensemble A des tâches actives à l'instant t :

$$A = \{k \in \{1, \ldots, n\} | t \geq r_k \wedge e_k < C_k + i.C_k\}$$

- l'ensemble L des tâches dont la laxité est nulle à l'instant t :

$$L = \{k \in A | C_k + i.C_k - e_k = r_k + i.T_k + D_k - t\}$$

où $i = \lfloor (t - r)/T_k \rfloor$.

Pour une tâche τ_k inactive à l'instant t ($k \notin A$), la probabilité de transition est :

$$\mathbb{P}(I^k_{t+1} = e_k | I^k_t = e_k) = 1$$

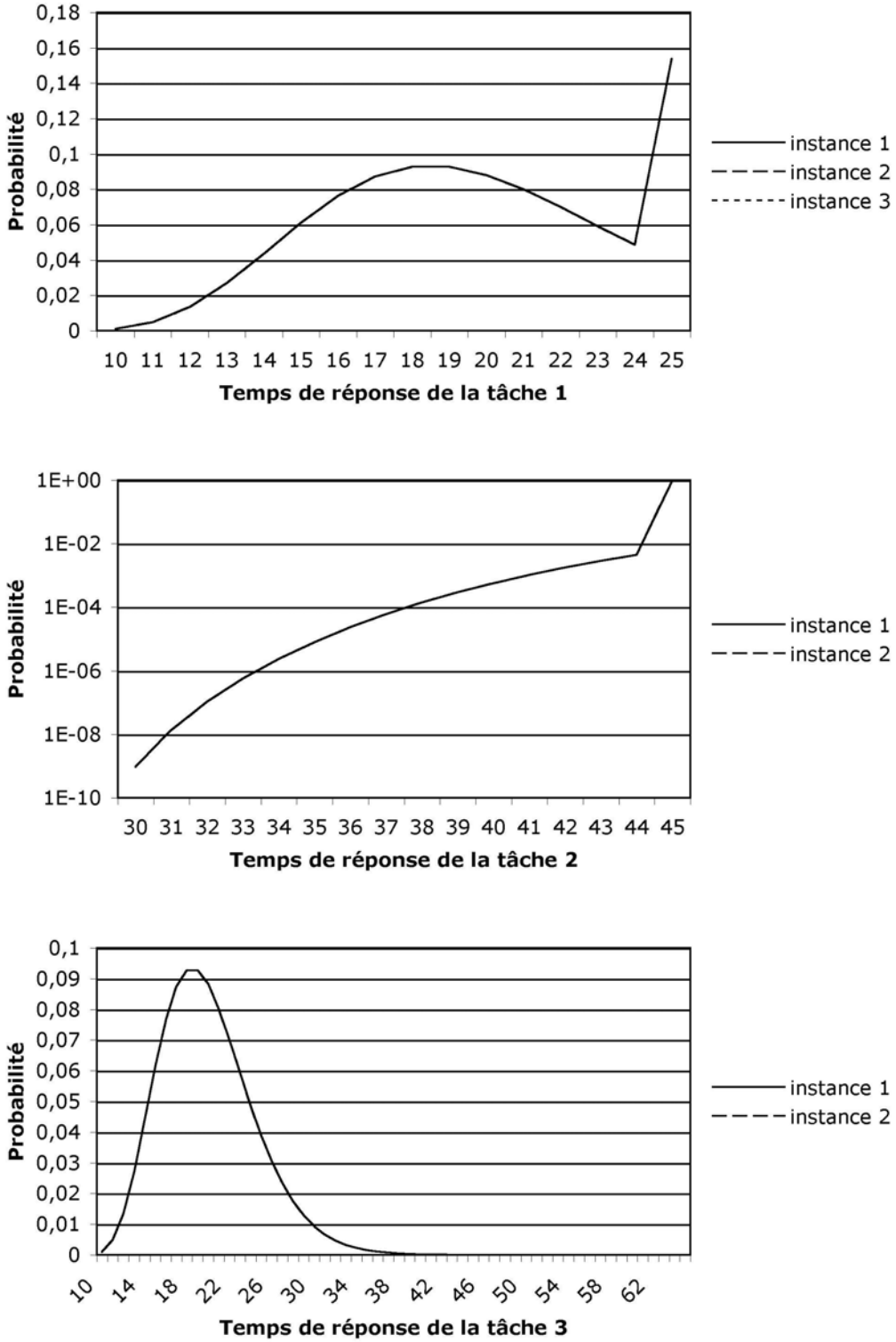


FIG. 12.4 – Distribution de probabilité du temps de réponse de chaque instance

Pour une tâche τ_k active à l'instant t et dont la laxité est strictement positive ($k \in A \setminus L$), la probabilité de transition est :

$$\mathbb{P}(I_{t+1}^k = e_k | I_t^k = e_k) = 1/2$$

$$\mathbb{P}(I_{t+1}^k = e_k + 1 | I_t^k = e_k) = 1/2$$

Et finalement, pour une tâche dont la laxité est nulle ($k \in L$), la probabilité de transition est :

$$\mathbb{P}(I_{t+1}^k = e_k + 1 | I_t^k = e_k) = 1$$

Ainsi, seules les tâches appartenant à $A \setminus L$ ont le choix entre plusieurs transitions possibles. Le nombre de transitions autorisées à partir de l'état e à l'instant t est donc :

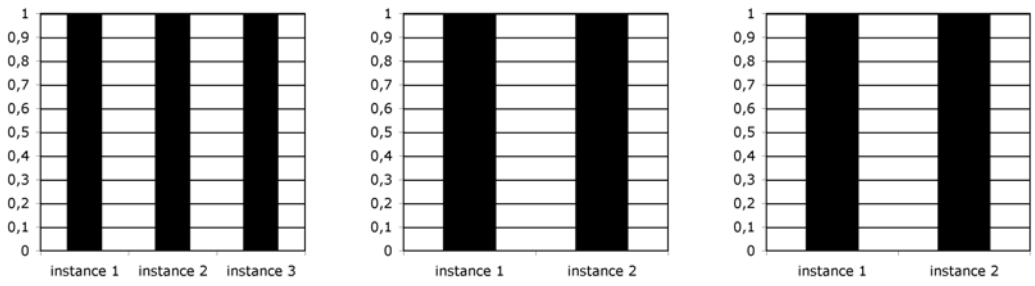
$$\sum_{k=0}^{|A|-|L|} C_{|A|-|L|}^k$$

En supposant qu'elles sont toutes équiprobables, la probabilité d'une transition valide partant de e à l'instant t est alors :

$$\frac{1}{\sum_{k=0}^{|A|-|L|} C_{|A|-|L|}^k}$$

Reprenons maintenant le système de tâches utilisé dans la section précédente pour étudier les temps de réponse obtenus par la loi équiprobable. Les figures 12.4 et 12.4 indiquent ceux que l'on obtient avec la loi que nous venons de définir. On constate que toutes les instances ont une probabilité 1 de respecter leur échéance : leurs caractéristiques temporelles sont donc bien représentées par cette loi de probabilité. De plus, toutes les instances d'une même tâche ont la même distribution de probabilité du temps de réponse. Ce point confirme le fait que cette loi considère que les tâches sont indépendantes.

Comme pour la loi équiprobable, on remarque que les distributions obtenues sont des gaussiennes tronquées par la fenêtre temporelle dans laquelle la tâche doit terminer son exécution. Par contre, un pic apparaît au moment de l'échéance des tâches, il correspond aux ordonnancements pour lesquels la tâche atteint une laxité nulle - i.e. où elle est exécutée au plus tard. En choisissant $1/2$ comme probabilité d'exécution d'une tâche τ_k active dont la laxité n'est pas nulle, on centre la gaussienne sur la valeur $2C_k$. Il peut être intéressant de choisir une autre valeur pour obtenir un autre centrage. Par exemple, en choisissant C_k/D_k, on obtient un centrage sur l'instant D. Ce rapport est intéressant puisqu'il représente des accès aux processeurs uniformément répartis dans le temps, mais il renforce encore le pic correspondant à l'échéance. Pour obtenir une répartition plus uniforme dans la fenêtre $\{C_k, \ldots, D_k\}$, on utilise une probabilité de transition qui centre la gaussienne au milieu de la fenêtre temporelle : $\frac{2C_k}{D_k+C_k}$ (voir figure 12.6). Avec cette loi aussi, la probabilité de respecter les échéances vaut 1, elle permet d'obtenir un comportement moyen pour chaque instance.



Tâche τ_1 Tâche τ_2 Tâche τ_3

FIG. 12.5 – Probabilité de respect des échéances

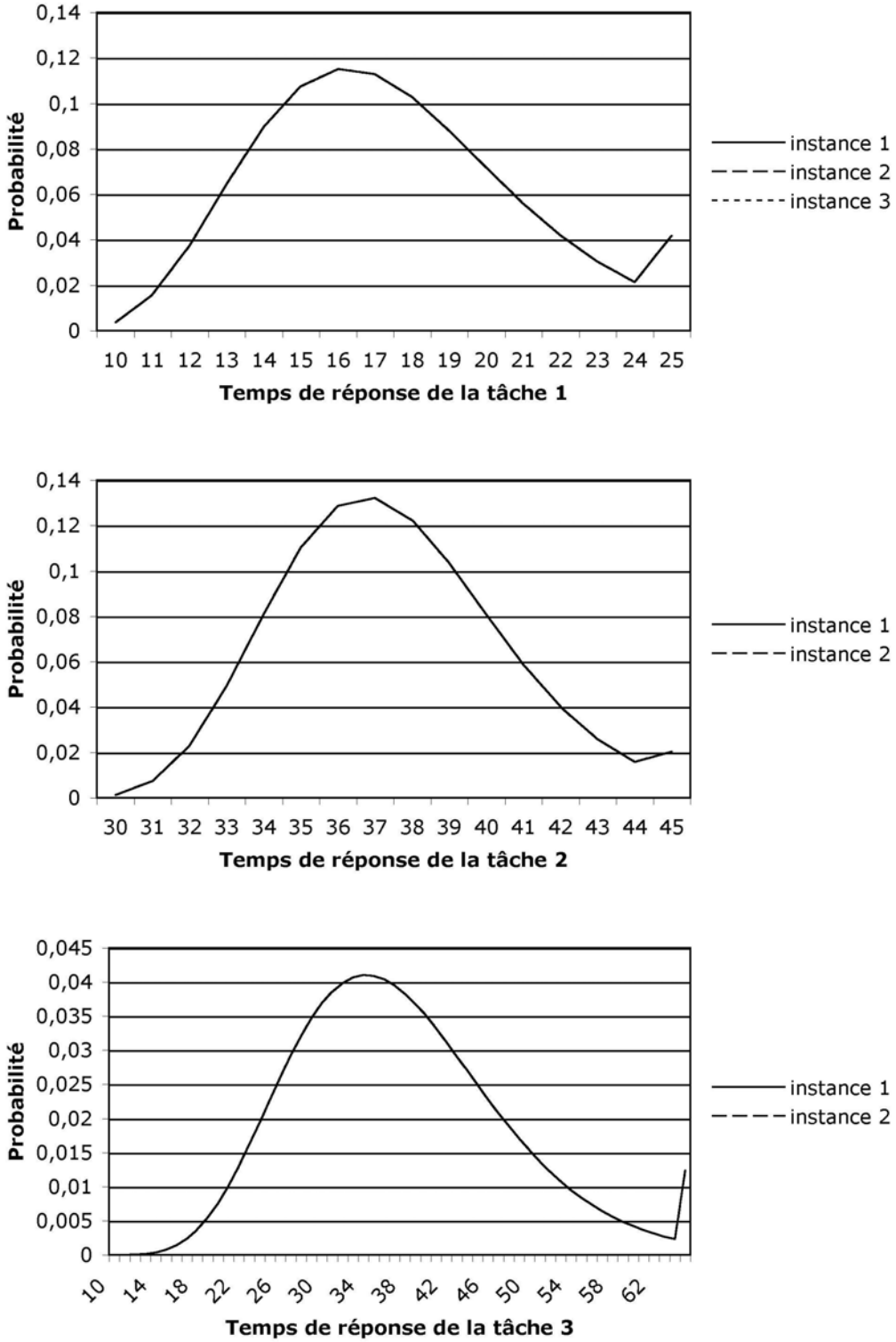


FIG. 12.6 – Distribution de probabilité du temps de réponse de chaque instance

Cette loi est donc intéressante puisqu'elle traduit bien le comportement temporel des tâches. Cependant, elle ne tient pas compte de l'architecture de l'exécutif hébergeant le système de tâches : les contraintes sous-jacentes doivent aussi être intégrées à la loi de probabilité.

12.5 Contexte préemptif et non-conservatif

Dans cette section, nous définissons une loi de probabilité adaptée aux exécutifs préemptifs avec migration totale. Dans ce contexte, la seule contrainte imposée par l'architecture est due au nombre de processeurs disponibles. Notre démarche consiste donc à intégrer par condition la propriété suivante : au plus p tâches peuvent être exécutées simultanément.

Considérons un état $e = (e_1, \ldots, e_n)$ valide à l'instant t et reprenons les notations A et L définies dans la section précédente, respectivement pour dénoter l'ensemble des tâches actives à l'instant t et l'ensemble des tâches dont la laxité est nulle à l'instant t.

Les seules tâches admettant deux transitions valides sont celles appartenant à $A \setminus L$. Le nombre de transitions valides issues de e à l'instant t est donc :

$$\sum_{k=0}^{\min\{p-|L|,|A|-|L|\}} C^k_{|A|-|L|}$$

En tenant compte des tâches actives et de celles dont la laxité est nulle, on intègre par condition le comportement temporel des tâches. En prenant en compte le nombre de processeurs disponibles, on intègre par condition les contraintes dues à l'exécutif.

Reprenons maintenant le système de tâches que nous utilisons pour étudier les lois de probabilité que nous définissons. Les figures 12.7 et 12.7 présentent les distributions de probabilité du temps de réponse que l'on obtient avec la loi que nous venons de définir.

On constate que les courbes correspondant aux instances d'une même tâche n'ont pas la même topologie. Cette nouvelle loi ne considère donc pas que les tâches sont indépendantes (au sens probabiliste) : les interactions dues au partage du processeur sont donc représentées. Certaines particularités apparaissent aussi sur ces courbes.

Par exemple, la probabilité que la première instance de τ_2 soit terminée avant l'instant 40 est nulle. Interprétons ce constat. A l'instant $t = 0$, les trois tâches sont activées. A l'instant $t = 25$, l'échéance de τ_1 arrive et elle doit donc avoir été exécutée. Entre $t = 0$ et $t = 25$, la tâche τ_2 n'a pu accéder qu'au plus 15 fois au processeur. Il lui reste donc 15 accès à réaliser, et elle ne peut donc pas se terminer avant l'instant $t = 25 + 15 = 40$.

La distribution de probabilité de la première instance de τ_3 est encore plus surprenante puisqu'elle est discontinue, expliquons ce phénomène. Entre $t = 0$ et $t = 45$, les tâches τ_1 et τ_2 utilisent 40 accès au processeur. Pour qu'elles respectent leur échéance, la tâche τ_3 ne peut donc accéder que 5 fois le processeur jusqu'à l'instant $t = 45$. Elle ne peut donc pas être terminée avant l'instant $t = 50$. Ce constat confirme la partie "droite" de la courbe de la première instance de τ_3. D'autre part, la partie "gauche" ne peut correspondre qu'à des séquences non-valides : certaines tâches manqueront leur échéance bien qu'elles ne soient pas encore de laxité négative. A l'instant $t = 0$, la laxité de τ_1 est 15, et celle de τ_3 est aussi 15. Supposons que τ_3 est exécutée dès le début ($t = 0$). A l'instant $t = 10$, la laxité de τ_1 et de τ_2 est 5. En exécutant ensuite τ_1 pendant 5 unités de temps, on arrive à l'instant $t = 15$ où la laxité de τ_1 est encore 5 et celle de τ_2 est 0. On peut alors exécuter τ_2 pendant 5 unités de temps. A l'instant $t = 21$, l'une de ces deux tâches a obligatoirement une laxité négative : les ordonnancements correspondants sont alors éliminés par la réduction à la laxité. Les temps de réponse correspondants disparaissent alors aussi, et la coupure dans la courbe des temps de réponse de τ_3 apparaît. La partie "gauche" de cette courbe apparaît seulement parce qu'aucun critère ne permet de détecter suffisamment tôt que ces temps de réponse amènent obligatoirement à un dépassement d'échéance.

Certaines discontinuités peuvent aussi apparaître naturellement. Prenons l'exemple d'une tâche τ_k pouvant ou non être active à l'instant t en fonction de la séquence d'exécution considérée. Supposons qu'à l'instant t d'autres tâches s'activent et qu'elles requièrent totalement l'accès au

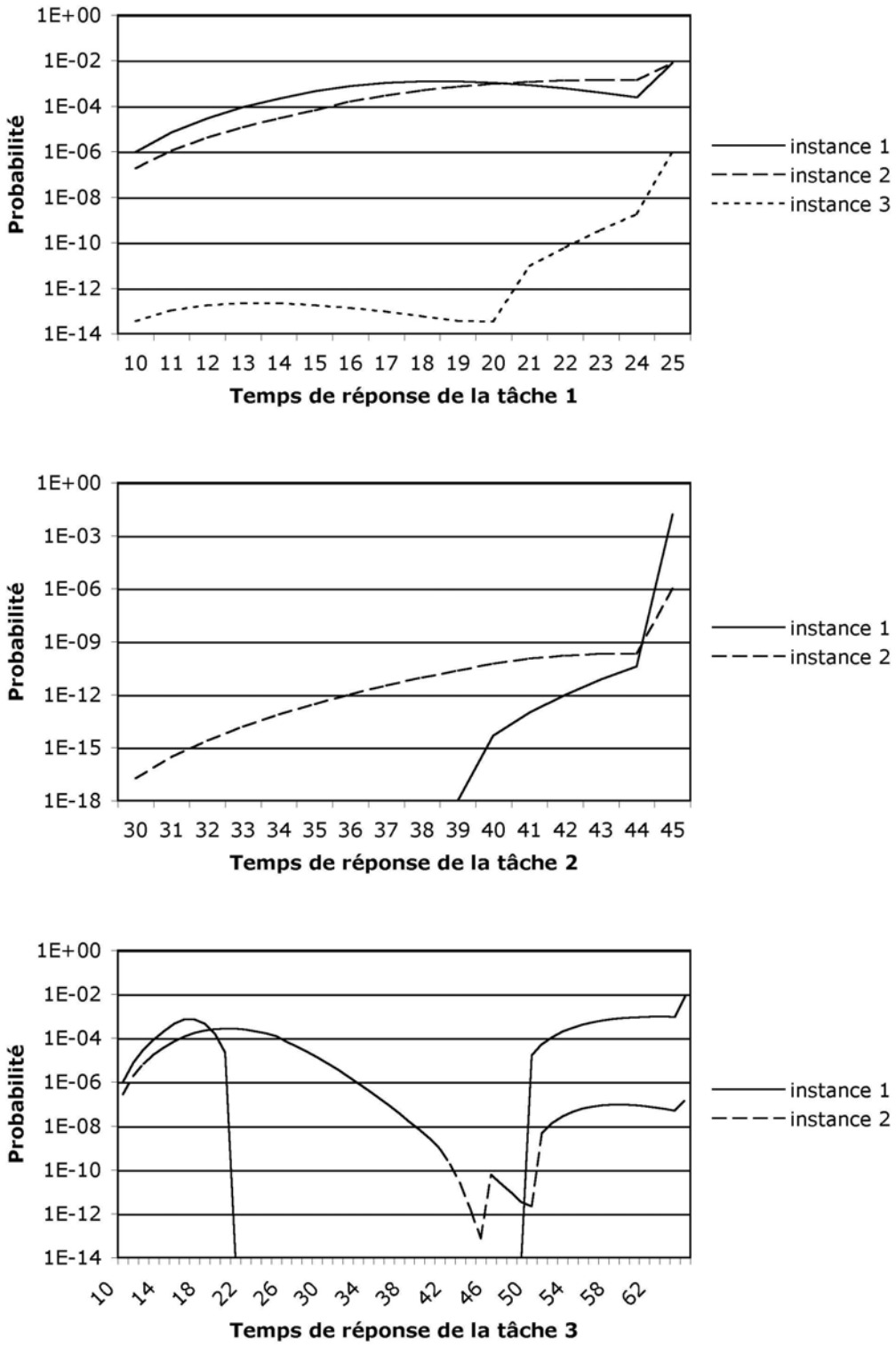


FIG. 12.7 – Distribution de probabilité du temps de réponse de chaque instance

processeur pendant un certain temps Δ pour que leurs échéances soient respectées. Pour une séquence d'exécution où τ_k est active à l'instant t, son échéance est différée de Δ unités de temps et arrive donc strictement après $t + \Delta$. Pour une séquence d'exécution où τ_k est inactive en t, son échéance est donc au plus tard en t. Cette situation fait donc apparaître une discontinuité dans la courbe du temps de réponse de τ_k.

Ces différentes courbes montrent que cette loi de probabilité permet de faire ressortir des phénomènes propres au système de tâches considéré et qui sont dus aux interactions avec l'exécutif temps-réel ciblé. Ceux-ci ne sont pas directement visibles à partir de la spécification temporelle des tâches. Cette loi permet donc d'apporter un diagnostic significatif de l'ordonnançabilité de ce système de tâches.

12.6 Contexte préemptif et conservatif

Les politiques d'ordonnancement temps-réel sont généralement conservatives. La prise en compte de cette propriété permet d'obtenir des courbes de temps de réponse plus proches de la réalité. Les exécutifs temps-réel considérés ici sont donc conservatifs et préemptifs avec migration totale. Pour intégrer cette contrainte, la loi de probabilité doit tenir compte par condition de la propriété suivante : le nombre de tâches exécutées à un instant donné est égal au minimum du nombre de processeurs et du nombre de tâches actives.

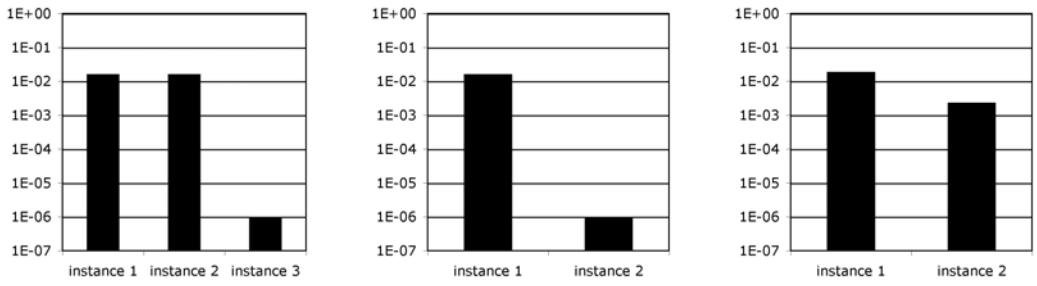
Considérons un état $e = (e_1, \ldots, e_n)$ valide à l'instant t et reprenons les notations A et L définies précédemment, respectivement pour dénoter l'ensemble des tâches actives à l'instant t et l'ensemble des tâches dont la laxité est nulle à l'instant t.

Les seules tâches admettant deux transitions valides sont celles appartenant à $A \setminus L$. Le nombre de transitions valides partant de e à l'instant t est donc :

$$C_{|A|-|L|}^{\min\{p-|L|,|A|-|L|\}}$$

Reprenons maintenant le système que nous utilisons pour étudier les lois de probabilité que nous définissons. La figure 12.9 présente les distributions de probabilité du temps de réponse que l'on obtient avec la loi que nous venons de définir. Elles sont sensiblement identiques à celles obtenues avec la loi pour les exécutifs non-conservatifs. Remarquons tout de même quelques différences. Par exemple, toutes les instances de la tâche τ_3 sont obligatoirement terminées avant leur échéance. Ceci signifie qu'à la fin de leur exécution, il n'y a aucune autre instance active, et puisque l'exécutif est conservatif, elles sont obligatoirement exécutées. On remarque aussi le même phénomène avec la deuxième instance de la tâche τ_1.

La figure 12.10 présente la probabilité que l'échéance de chaque instance soit respectée. Pour chaque instance, cette probabilité varie entre $0,2$ et $0,3$. Ainsi, cette loi de probabilité permet d'obtenir un indice sur l'ordonnançabilité d'une instance dont l'ordre de grandeur paraît plus significatif que celui de l'indice obtenu avec la loi correspondant aux exécutifs non-conservatifs.



Tâche τ_1 Tâche τ_2 Tâche τ_3

FIG. 12.8 – Probabilité de respect des échéances

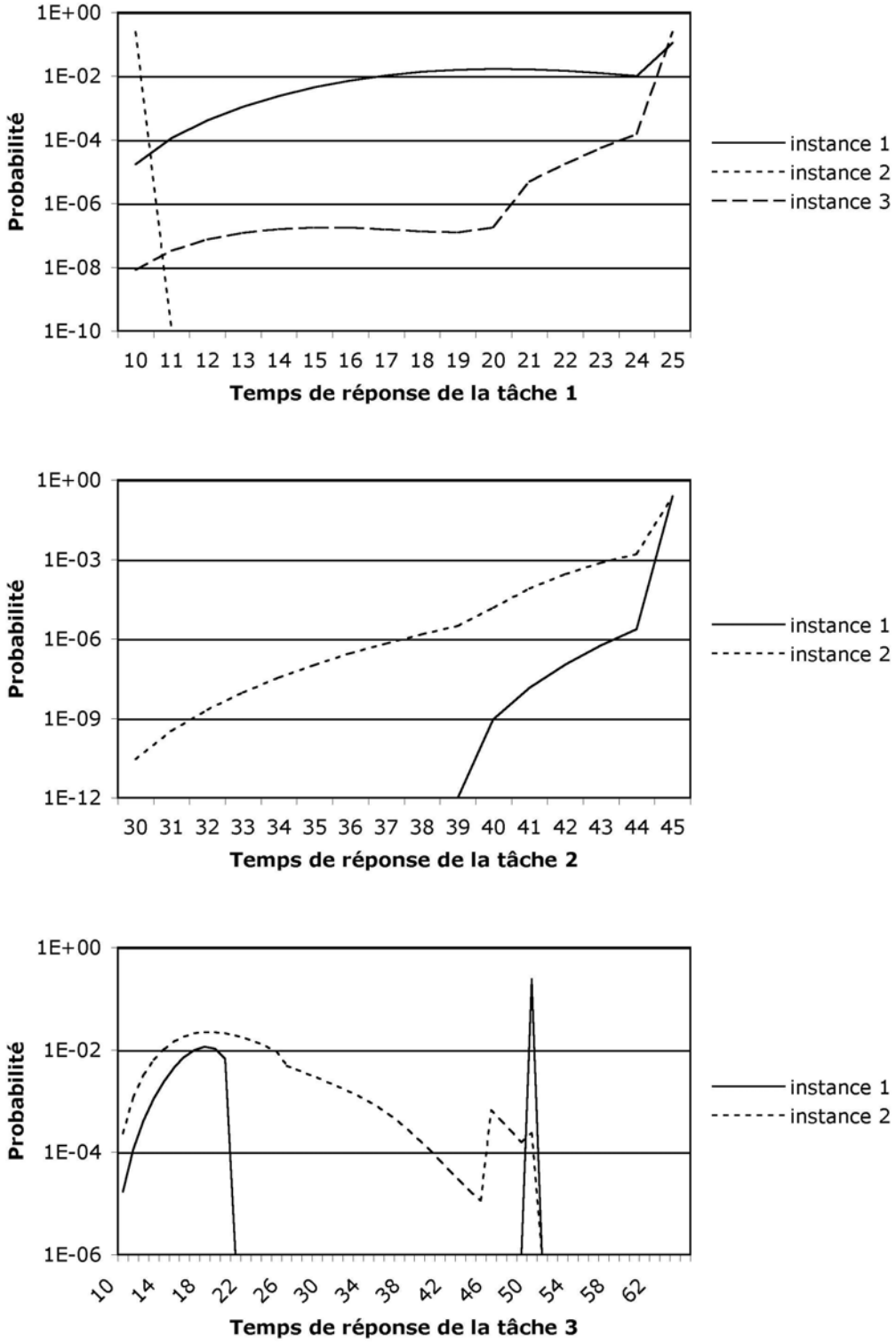


FIG. 12.9 – Distribution de probabilité du temps de réponse de chaque instance

12.7 Contexte non-préemptif et conservatif

Les exécutifs non-préemptifs ne permettent pas qu'une instance en cours d'exécution soit interrompue. Dans ce type de contexte, les ordonnancements sont généralement conservatifs, nous n'étudions donc pas le cas non-préemptif et non-conservatif.

Considérons un état $e = (e_1, \ldots, e_n)$ valide à l'instant t et adoptons les deux notations suivantes :

- A : l'ensemble des tâches actives à l'instant t,
- E : l'ensemble des tâches qui sont en cours d'exécution

$$E = \{k \in A | e_k > C_k . \lfloor (t - r)/T_k \rfloor\}$$

Lorsque $A = E$, toutes les instances actives sont en cours d'exécution, la seule transition possible consiste à continuer à exécuter les mêmes tâches. De même, lorsque $|E| = p$, tous les processeurs sont occupés, on ne peut donc que continuer à exécuter les mêmes tâches. Dans ces deux situations, la seule transition possible est :

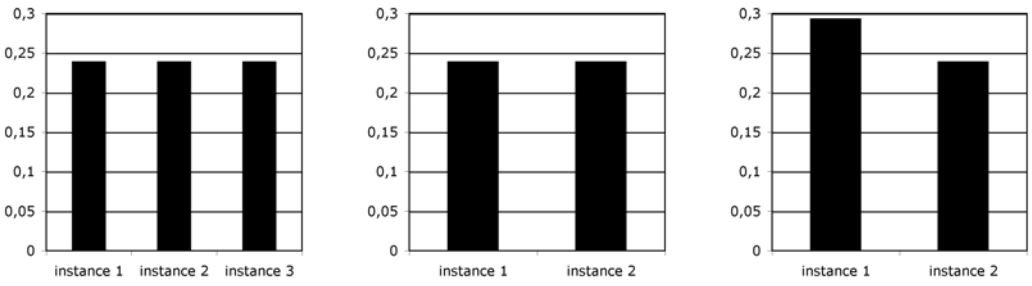
$$\mathbb{P}(I_{t+1} = e' | I_t = e) = 1 \Leftrightarrow \begin{cases} \forall k \in E, e'_k = e_k + 1 \\ \forall k \in \{1, \ldots, n\} \setminus E, e'_k = e_k \end{cases}$$

Lorsque $A \neq E$ et $|E| < p$, il existe des instances prêtes à être exécutées ainsi que des processeurs disponibles. Les transitions possibles correspondent aux combinaisons sans ordre ni remise à $\min\{p - |E|, |A| - |E|\}$ éléments choisies dans l'ensemble $A \setminus E$: les tâches exécutées étant celles appartenant à E ou à la combinaison choisie. Le nombre de ces transitions est donc :

$$C_{|A|-|E|}^{\min\{p-|E|,|A|-|E|\}}$$

En supposant qu'elles soient équiprobables, on en déduit les probabilités de transition. Pour les définir, nous n'avons pas tenu compte des tâches dont la laxité est nulle. On peut naturellement le faire, mais les différences obtenues dépendent de la combinaison à un même instant de deux situations indépendantes : il existe une tâche de laxité nulle et une tâche dont l'exécution s'achève. La combinaison de ces deux évènements étant rare, nous ne l'avons pas prise en compte.

Reprenons maintenant le système de tâches que nous utilisons pour étudier les lois de probabilité que nous définissons. La figure 12.11 présente les distributions de probabilité du temps de réponse que l'on obtient avec la loi que nous venons de définir. Les courbes obtenues ne sont plus constituées que de pics. Effectivement, les séquences étant non-préemptives, les tâches en cours d'exécution ne peuvent pas permuter avec celles qui attendent. Ainsi chaque pic correspond aux choix de l'ordre d'exécution des tâches actives réalisés lorsqu'un processeur devient oisif. D'autre part, le nombre de séquences prises en compte étant beaucoup plus faible que dans un contexte préemptif, l'ordre de grandeur des probabilités obtenues est plus raisonnable (autour de 10^{-1}). Elles fournissent donc un indicateur intéressant.



Tâche τ_1 Tâche τ_2 Tâche τ_3

FIG. 12.10 – Probabilité de respect des échéances

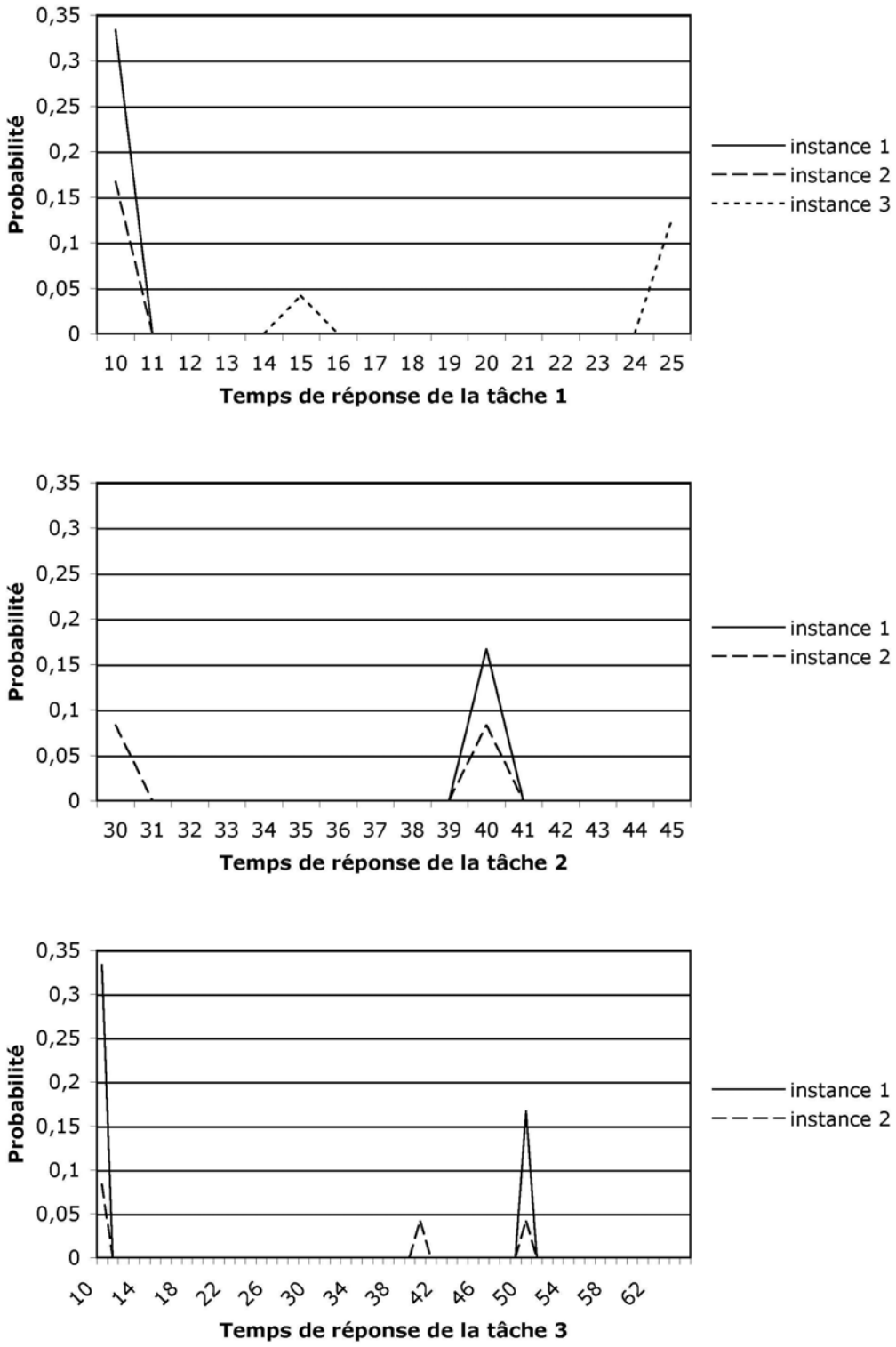


FIG. 12.11 – Distribution de probabilité du temps de réponse de chaque instance

La figure 12.12 indique la probabilité pour qu'une instance respecte son échéance. L'ordre de grandeur des probabilités obtenues (10^{-1}) rend ces informations significatives. En moyenne, chaque instance respecte son échéance avec une probabilité d'environ $0,2$. Ce résultat est indépendant de la politique d'ordonnancement choisie tant que l'exécutif est non-préemptif et conservatif. Ce type d'indicateur fournit donc une information intéressante sur les parties critiques d'un système de tâches.

En fonction de l'exécutif concerné, le nombre de transitions possibles varie beaucoup. Nous avons évalué le nombre maximum d'états atteints à un même instant en fonction de la loi de probabilité utilisée. Le tableau suivant résume nos résultats :

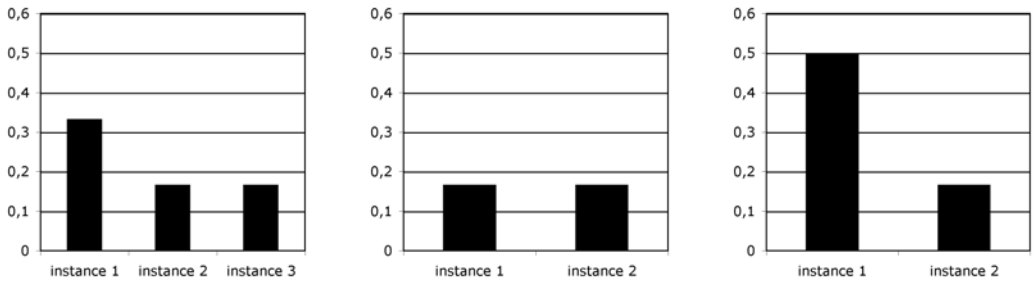
Loi de probabilité	Nombre maximum d'états atteints
équiprobable	1936
tâches indépendantes	1936
préemptif et non-conservatif	746
préemptif et conservatif	106
non-préemptif et conservatif	5

Pour ce système de tâches, la complexité spatiale au pire est 3751. Les lois n'intégrant pas les contraintes dues à l'exécutif n'offrent aucune réduction du nombre d'états atteignables. La valeur 1936 correspond à la complexité réelle, alors que 3751 est un majorant. La différence entre les deux illustre le gain fourni par la réduction à la laxité par rapport à la réduction aux premières instances. Par contre, dès que l'on prend en compte les spécificités de l'exécutif, le nombre maximum d'états atteints peut décroître fortement. Pour les exécutifs non-préemptifs, le gain est même considérable et permet d'envisager l'utilisation de notre méthode avec des systèmes de tâches complexes.

12.8 Conclusion

Nous avons développé une méthode permettant d'obtenir la distribution de probabilité du temps de réponse de chaque instance. Pour chaque type d'exécutif, nous avons élaboré une loi de probabilité tenant compte de leurs spécificités. Ainsi, pour un même système de tâches, on obtient des informations sur son comportement moyen pour chaque type d'exécutif. Ces informations sont indépendantes de la politique d'ordonnancement utilisée. En particulier, elles peuvent servir à choisir une politique adaptée à ce système de tâches, voire même à adapter une politique donnée au système de tâches. Dans le cas où un système se révèle être difficilement ordonnançable, les distributions du temps de réponse permettent d'identifier les zones où l'ordonnancement est difficile. On peut alors aussi les utiliser pour intervenir sur la spécification même du système de tâches. Elles peuvent donc aussi servir à modifier une spécification temporelle afin de la rendre plus facilement ordonnançable.

Les courbes du temps de réponse que nous avons obtenues expérimentalement révèlent des informations qui ne sont pas directement identifiables à partir de la seule spécification temporelle.



Tâche τ_1 Tâche τ_2 Tâche τ_3

FIG. 12.12 – Probabilité de respect des échéances

Elles permettent alors d'établir un diagnostic significatif et représentatif de l'exécutif ciblé. Toutefois, leur analyse, et donc leur interprétation, nécessitent encore un effort de recherche. La mise en place d'une méthode d'interprétation automatisée, ou au moins assistée, semble possible. De plus, étant donné la quantité d'informations fournies par notre méthode (dès que le système de tâches est conséquent), un tel prétraitement semble nécessaire. A terme, on peut sans doute obtenir un diagnostic élaboré et destiné aux concepteurs des systèmes temps-réel.

Pour les exécutifs préemptifs, les distributions de temps de réponse se présentent sous la forme de fonction continue par morceaux. Par contre, pour ceux non-préemptifs, elles sont sous la forme de pics successifs. Chacun de ces pics correspondant à un choix d'ordonnancement différent, on doit pouvoir les associer avec des configurations de priorité. Leur interprétation semble donc prometteuse. D'autre part, on peut sans doute établir une correspondance entre les pics des courbes obtenues dans le contexte non-préemptif avec les morceaux continus des courbes obtenues dans le contexte préemptif.

Finalement, la complexité spatiale utilisée dans le cas des exécutifs non-préemptifs est beaucoup plus faible qu'avec les exécutifs préemptifs. Elle permet d'envisager l'application de notre méthode à des systèmes de tâches de taille industrielle dès que l'exécutif ciblé est non-préemptif.

Les travaux exposés dans ce chapitre ont fait l'objet d'un article soumis à une revue internationale en 2007.

Conclusion générale

Bilan. Actuellement, la majorité des systèmes temps-réel sont ordonnancés à l'aide d'une politique d'ordonnancement à priorités. Des alternatives existent, par exemple certaines méthodes hors-ligne basées sur des modèles à états permettent de trouver des séquences valides, mais leur manque de flexibilité est souvent un frein à leur mise en exploitation. D'autres solutions plus réalistes consistent à déterminer une configuration de priorité adaptée au système à ordonnancer. Cette dernière approche permet d'utiliser toute la puissance analytique nécessaire tout en conservant la flexibilité de l'ordonnancement en-ligne. En suivant cette approche, nous avons élaboré plusieurs algorithmes permettant de fournir des configurations de priorité valides pour ordonnancer un système de tâches exécutées en contexte multiprocesseur et préemptif avec migration totale. Ces résultats prolongent au cas multiprocesseur l'approche suivie par [9] pour les ordonnancements monoprocesseurs produits par des politiques à priorités fixes. Nous avons aussi étendu le spectre des configurations de priorité recherchées à celles de la classe $\overline{\overline{\text{PFI}}}$. Nous avons étudié les propriétés de cette classe de politiques d'ordonnancement. Nous avons montré qu'elle est optimale pour ordonnancer des systèmes de tâches en monoprocesseur. Elle semble aussi être une bonne candidate pour :

- être la plus petite classe optimale en monoprocesseur,
- être la classe la plus puissante représentable à l'aide d'un nombre fini de niveaux de priorité.

La première de ces deux conjectures s'appuie sur le fait que la classe des politiques à priorités fixes n'est pas optimale en monoprocesseur, et la seconde repose sur les constats suivants :

- la séquence produite par EDF n'est pas toujours représentable à l'aide d'un nombre fini de niveaux de priorité,
- la politique EDF appartient à $\overline{\text{PFI}}$ et à DCL.

La classe $\overline{\overline{\text{PFI}}}$ semble bien adaptée aux exécutifs temps-réel actuels puisque la plupart d'entre eux reposent sur un nombre fini de niveaux de priorité attribuée statiquement aux tâches (ou aux instances). Les expérimentations que nous avons menées ont montré que la classe des politiques d'ordonnancement à priorités fixes et celle des politiques à priorités fixes par instance offrent une puissance d'ordonnancement dépassant par exemple celle de EDF.

Les propriétés offertes par PFX et $\overline{\overline{\text{PFI}}}$ nous ont incités à rechercher des méthodes permettant d'obtenir des configurations de priorité valides. Nous avons alors suivi plusieurs approches. Tout d'abord, nous avons proposé un algorithme de calcul des configurations de priorité engendrant une séquence quelconque. Cet algorithme peut être assimilé à une passerelle entre les exécutifs à priorités et les méthodes exactes ou approchées construisant des séquences d'exécution valides. Il facilite l'intégration de ces séquences dans les exécutifs temps-réel. Ensuite, nous avons proposé un algorithme calculant toutes les configurations de priorité fixe. Cet algorithme permet au concepteur d'intégrer certaines contraintes sur les priorités des tâches. Son temps de calcul moyen pour les systèmes contenant une dizaine de tâches est de l'ordre de la seconde. Plusieurs optimisations sont possibles, celles-ci devraient lui permettre de traiter des systèmes contenant plusieurs dizaines de tâches. Nous avons étendu nos algorithmes aux configurations de priorité appartenant à $\overline{\overline{\text{PFI}}}$. Toutefois, la complexité de celui déterminant toutes les configurations valides devient importante. Finalement, nous avons proposé une hybridation entre PFX et $\overline{\overline{\text{PFI}}}$ permettant d'attribuer des priorités différentes à deux instances d'une même tâche seulement lorsque cela est nécessaire. Nous en attendons un gain important au niveau du temps de calcul.

Parallèlement à ces travaux, nous avons appliqué certaines méthodes d'approximation au problème de l'ordonnancement multiprocesseur d'un système de tâches liées par des relations de précédence. Ce type d'approche permet à l'aide d'heuristiques d'obtenir en temps polynomial une solution approchée d'un problème donné. Leur complexité est donc meilleure que celle des méthodes basées sur des modèles à états qui fournissent des solutions exactes. A partir des algorithmes génétiques développés par [52, 23], nous en avons élaboré un nouveau améliorant la qualité des solutions fournies par [52] tout en utilisant un temps de calcul inférieur à celui de [23]. Nous avons ensuite proposé un algorithme basé sur la méthode taboue obtenant des solutions de meilleure qualité que ceux proposés par [23, 52] en utilisant un temps de calcul généralement inférieur. La qualité des solutions obtenues par la méthode taboue est généralement très bonne (proche de 90% de l'optimal théorique), cette méthode nous paraît donc satisfaisante.

Nos recherches ont aussi porté sur l'étude des propriétés théoriques de l'ordonnancement des systèmes temps-réel en contexte multiprocesseur. Nous avons établi un intervalle de faisabilité dont la validité dépend seulement de deux hypothèses portant sur le temps de réponse des instances (voir section 4.2) :

- le temps de réponse des instances ne peut pas diminuer,
- le temps de réponse des instances croît seulement par effet domino.

Nous avons montré que les séquences multiprocesseurs produites par les politiques de $\overline{\text{PFI}}$ en contexte préemptif avec migration totale satisfont ces deux hypothèses. Ce résultat englobe par exemple les séquences d'exécution produites par les politiques RM, DM et EDF. Pour les séquences d'exécution conservatives et satisfaisant les deux hypothèses données ci-dessus, nous avons aussi montré que les temps de réponse des instances se stabilisent toujours dès que le système de tâches n'est pas surchargé. Ce résultat permet d'obtenir un intervalle d'étude valable pour les séquences d'exécution non-valides, grâce à lui on peut alors établir un diagnostic complet sur les problèmes d'ordonnancement propres à une séquence d'exécution. Toutefois, ce résultat nécessite de connaître un majorant du temps de réponse de toutes les instances. Nous avons seulement montré qu'il en existe lorsque le système de tâches n'est pas surchargé ; toutefois, la valeur $\max\{T_1, \ldots, T_n\}$ nous paraît être une bonne candidate.

Afin d'obtenir un intervalle de faisabilité plus précis que celui fourni par nos premiers travaux, nous avons restreint notre contexte d'étude en nous consacrant aux politiques à priorités fixes. Nous avons alors établi une formule indiquant la durée exacte de montée en charge d'un système de tâches. La complexité de cette formule étant exponentielle, nous en avons alors proposé plusieurs majorations que nous avons ensuite comparées avec celle fournie par [24]. Nos majorants se sont révélés être plus précis. Nous avons aussi évalué expérimentalement la complexité du calcul de la durée exacte. En moyenne, celui-ci nous paraît être de complexité polynomiale. Excepté pour les systèmes contenant au moins une centaine de tâches, ce calcul nous paraît donc réalisable. Après avoir étudié les séquences produites par les politiques à priorités fixes, nous avons étendu notre approche à la classe $\overline{\text{PFI}}$. En appliquant une transformation au système de tâches étudié, on peut alors appliquer aux configurations $\overline{\text{PFI}}$ les résultats que nous avons obtenus pour les politiques à priorités fixes. Toutefois, la durée de montée en charge du système transformé est un majorant de celle du système initial. Nos résultats fournissent donc une majoration de la durée de montée en charge des séquences produites par les politiques de $\overline{\text{PFI}}$. En étudiant plus précisément les propriétés de la transformation que l'on applique, on doit pouvoir déterminer la durée exacte de leur montée en charge.

Lors de l'élaboration de notre méthode permettant de diagnostiquer un système temps-réel, nous avons eu besoin de déterminer un intervalle permettant d'étudier son comportement lorsque l'on considère simultanément toutes les séquences d'exécution. Nous avons montré pour le contexte multiprocesseur que les intervalles de faisabilité propres à une séquence d'exécution ne pouvaient pas être utilisés pour définir l'intervalle d'étude recherché. Nous avons alors élaboré une nouvelle approche de ce problème en considérant la cyclicité des états atteignables au lieu de celle des séquences d'exécution considérées. Nous avons établi qu'un intervalle d'étude existe et donné un critère permettant de détecter le début du régime cyclique des états atteignables. Nous avons

ensuite évalué expérimentalement la longueur de l'intervalle d'étude. Nous avons constaté pour certains systèmes de tâches qu'il est plus petit que l'intervalle de faisabilité de certaines séquences d'exécution. Plus précisément, nous avons donné un exemple où les états atteignables sont cycliques alors que la séquence produite par EDF est encore en phase de montée en charge.

Pour étudier la qualité de service offerte par un système temps-réel, nous avons conçu une méthode fournissant un diagnostic de son comportement en fonction de l'exécutif ciblé et indépendamment de la politique d'ordonnancement utilisée. Pour cela, nous avons modélisé les ordonnancements possibles par des marches aléatoires produites par des chaînes de Markov. Ensuite, nous avons intégré les contraintes dues aux interactions entre les tâches, par exemple : relations de précédence, exclusion mutuelle. Puis, nous avons intégré celles dues aux spécificités de l'exécutif temps-réel ciblé, par exemple : préemptivité, conservativité, migration totale ou partielle. Cette approche nous permet de déterminer la probabilité que les contraintes dues aux interactions entre les tâches soient respectées sachant que les contraintes dues à l'exécutif ciblé sont respectées. Les résultats que nous obtenons sont alors indépendants de toute politique d'ordonnancement.

Pour évaluer les marches aléatoires engendrées par notre modèle, nous avons élaboré un algorithme spécifique. Celui-ci permet de tirer parti de certaines contraintes afin de diminuer la complexité de calcul. Nous avons analysé sa complexité moyenne : cet indicateur permet d'approximer la durée de calcul utilisée par notre algorithme pour traiter un système de tâches. La complexité étant exponentielle, il peut se révéler très utile. En outre, nous avons constaté expérimentalement que le temps de calcul est très variable en fonction du type de l'exécutif ciblé. En particulier pour les exécutifs non-préemptifs, les temps de calcul obtenus expérimentalement permettent d'envisager l'utilisation de notre méthode avec des systèmes de tâches de taille industrielle.

En utilisant cet algorithme, nous avons pu calculer les distributions de probabilité du temps de réponse de chaque instance. Cette information permet de réaliser une analyse de la qualité de service offerte par un système de tâches. Elle permet aussi d'identifier les zones à risque d'un ordonnancement. Le diagnostic correspondant devrait aider le concepteur du système temps-réel à cerner les problèmes et donc le guider dans les modifications à apporter. Toutefois, nous avons seulement donné quelques exemples d'interprétation des courbes du temps de réponse des instances. Une étude approfondie est nécessaire pour exploiter réellement les informations fournies par notre méthode.

Perspectives. Lors de notre étude des politiques à priorités fixes et de celles à priorités fixes par instance, nous avons proposé plusieurs algorithmes fournissant des configurations de priorité valides. Ils ne doivent pas être considérés comme aboutis, ils ouvrent au contraire une voie vers un nouvel axe de recherche. L'étude que nous avons menée offre plusieurs perspectives possibles. Tout d'abord, les algorithmes que nous avons énoncés peuvent être optimisés de plusieurs manières :

- le traitement effectué à certains instants ne modifie pas les relations de priorité en cours de calcul, il peut donc être évité,
- le traitement de chaque relation de priorité est indépendant des autres, on peut donc paralléliser leur traitement,
- les calculs effectués sur les clôtures transitives des relations de priorité sont coûteux, il semble donc intéressant de développer une structure de données optimisée.

Une fois ces optimisations réalisées, on devrait pouvoir déterminer les configurations de priorité fixe valides pour des systèmes de tâches contenant plusieurs dizaines de tâches. Notre méthode pourrait alors être appliquée à des systèmes de taille industrielle. Pour calculer les configurations de priorité valides appartenant à $\overline{\overline{\mathrm{PFI}}}$, nous avons proposé une méthode itérative permettant de diminuer la complexité. La maîtrise de cette méthode paraît être un enjeu important puisque l'on pourrait alors utiliser la puissance d'ordonnancement fournie par $\overline{\overline{\mathrm{PFI}}}$ pour ordonnancer des systèmes contenant plusieurs dizaines de tâches.

Ensuite, nous avons considéré dans cette étude que les tâches sont indépendantes. En introduisant la notion de tâche bloquée dans notre méthode, il semble possible d'intégrer les contraintes de précédence et l'exclusion mutuelle. La combinaison de notre approche avec les protocoles de gestion

de ressources est alors envisageable. En particulier, on doit pouvoir déterminer les configurations de priorité valides en utilisant un protocole de gestion de ressources donné, par exemple PIP ou PCP. La prise en compte des exécutifs non-préemptifs semble aussi envisageable puisqu'il suffit alors de considérer que les relations de priorité engendrées à un instant sont significatives si et seulement si un choix d'ordonnancement est effectué à ce moment-là - i.e. une tâche est nouvellement attribuée à un processeur.

Ainsi, la méthode que nous avons établie ouvre de nombreuses perspectives de recherche. A terme, on peut espérer obtenir un outil généraliste permettant d'ordonnancer des systèmes de tâches réels en utilisant des configurations de priorité fixe : par tâches, ou par instances si nécessaire.

Etant donné la complexité théorique des problèmes posés par l'ordonnancement temps-réel, l'utilisation des méthodes d'approximation semble appropriée. Pour l'ordonnancement d'un système de tâches liées par des relations de précédence, l'algorithme basé sur la méthode taboue que nous avons proposé fournit des solutions de bonne qualité. L'utilisation des processus de décision markoviens devrait permettre d'obtenir des solutions de même qualité en un temps plus court, étant donné qu'ils ont déjà rendu ce type de service pour d'autres problèmes, par exemple [65, 53, 54]. Il semble donc intéressant d'explorer cette voie afin de diminuer la complexité.

L'enrichissement du modèle de tâches que nous avons considéré pour cette étude, et notamment l'introduction d'échéances temporelles, permettrait d'appliquer nos travaux aux systèmes temps-réel. Effectivement, ce type d'approche est récent et les modèles pris en compte ne permettent pas encore de représenter entièrement un système temps-réel. Beaucoup reste à faire dans ce domaine, mais l'application des méthodes d'approximation au problème de l'ordonnancement temps-réel paraît prometteuse.

Par ailleurs, ce type de méthode peut aussi être utilisé pour déterminer des configurations de priorité valides, plutôt que des séquences d'exécution valides dont l'exploitation est ensuite généralement délicate. Leur application, par exemple à la classe $\overline{\overline{\text{PFI}}}$, semble intéressante et fournirait un outil complémentaire de ceux que nous avons déjà élaborés pour calculer des configurations de priorité valides.

Les hypothèses que nous avons formulées dans le chapitre 4 ont permis d'obtenir un intervalle de faisabilité pour une classe de politiques d'ordonnancement englobant par exemple RM, DM et EDF. Nous avons montré que les séquences produites par les politiques appartenant à $\overline{\text{PFI}}$ vérifient nos deux hypothèses. Toutefois, d'autres séquences d'exécution produites par exemple par LLF vérifient aussi nos hypothèses. Afin de poursuivre cette étude, plusieurs axes de recherche peuvent être considérés.

Tout d'abord, on peut chercher à affiner l'intervalle de faisabilité que nous avons proposé, soit dans le cas général, soit en se consacrant à certaines classes d'ordonnanceurs comme nous l'avons fait dans le chapitre 5 pour les séquences produites par les politiques à priorités fixes (et par extension celles de $\overline{\overline{\text{PFI}}}$).

Ensuite, on peut chercher à déterminer le spectre exact de nos deux hypothèses, et notamment répondre à la question suivante : est-ce que les séquences produites par les politiques de DCL vérifient nos deux hypothèses ? En outre, il semble possible d'appliquer notre approche aux séquences d'exécution produites par différents types d'exécutifs temps-réel, par exemple : non-préemptif, préemptif avec migration totale. En étudiant la validité de nos deux hypothèses dans ces contextes, on pourrait étendre la portée de notre intervalle de faisabilité et de notre résultat sur la cyclicité des séquences d'exécution. L'étude de l'influence des interactions entre les tâches sur la validité de nos hypothèses pourrait permettre d'étendre le spectre de nos résultats au-delà du contexte des tâches indépendantes.

Finalement, notre analyse est aussi valable pour les séquences d'exécution qui ne respectent pas les échéances des tâches, et nous avons pu fournir une borne à la durée maximale pour que les temps de réponse des instances se stabilisent. Toutefois, cette borne nécessite de connaître un majorant des temps de réponse. Pour les séquences conservatives, nous avons suggéré que $\max\{T_1, \ldots, T_n\}$ pouvait être l'un d'entre eux. Résoudre cette question fournirait une durée d'étude maximale

pour les séquences non valides. On pourrait alors effectuer un diagnostic complet des problèmes d'ordonnancement propres à une séquence d'exécution donnée.

Les chaînes de Markov se sont révélées être un outil efficace pour modéliser le comportement des systèmes temps-réel. Nous avons notamment pu intégrer les interactions entre les tâches et les propriétés propres à l'exécutif temps-réel ciblé. Pour les exécutifs non-préemptifs, la complexité de notre méthode laisse envisager son application à des systèmes de tâches de taille industrielle. Pour véritablement exploiter notre approche, il est nécessaire d'établir une méthode permettant d'interpréter les résultats fournis. Pour les systèmes de tâches contenant beaucoup de tâches, la quantité d'informations produites par notre méthode peut être considérable. Il paraît alors nécessaire "d'automatiser" leur étude, ou tout au moins "d'assister" le concepteur du système dans son analyse. Ce point constitue sans doute le principal enjeu avant d'envisager l'exploitation réelle de notre approche.

Publications

Publications dans les journaux d'audience nationale avec comité de lecture :

B. Chauvière and D. Geniet. *Une approche markovienne pour l'étude de systèmes temps-réel à contraintes strictes.* Technique et Science Informatiques, 34 pages, parution en décembre 2007.

Publications dans les actes de conférence d'audience internationale avec comité de lecture :

B. Chauvière, D. Geniet and R. Schott. *Contributions to the multiprocessor scheduling problem.* In proceedings of IASTED Computational Intelligence, pp. 55–60, 2007.

Publications dans les actes de conférence d'audience nationale avec comité de lecture :

G. Largeteau, B. Chauvière and D. Geniet. *Une approche géométrique de la validation temps-réel.* In proceedings of Real-Time Systems, pp. 289–302, Teknea, 2005.

B. Chauvière and D. Geniet. *Quantification de l'ordonnançabilité des systèmes temps-réel à contraintes strictes par l'analyse markovienne.* In proceedings of RJCITR, pp. 41–44, 2005.

Articles soumis à des journaux avec comité de lecture :

B. Chauvière, D. Geniet and R. Schott. *A New Method for Determining Fixed Priority Configurations for Real-Time Systems on Multiprocessor Targets.* 33 pages.

B. Chauvière and D. Geniet. *Multiprocessor Real-Time Scheduling : Contributions to the Cyclicity Problems.* 32 pages.

B. Chauvière. *Durée exacte de montée en charge des systèmes temps-réel périodiques ordonnancés en multiprocesseur par une politique à priorités fixes.* 28 pages.

B. Chauvière and D. Geniet. *Diagnostic de l'ordonnançabilité des systèmes temps-réel basé sur une analyse markovienne des temps de réponse.* 24 pages.

Notations

Notation	**Définition**
τ	Ensemble de tâches
n, p	Nombre de tâches appartenant à τ, et le nombre de processeurs
$\overline{\tau}$	Ensemble des instances engendrées par les tâches de τ
τ_k, $\tau_{k,i}$	$k^{\text{è}}$ tâche et sa $i^{\text{è}}$ instance
$\equiv_m$	Equivalence entre deux instances d'une même tâche (voir page 11)
r_k, C_k, D_k, T_k	Caractéristiques temporelles de la tâche τ_k (voir page 12)
$r_{k,i}$, $d_{k,i}$	Date d'activation et échéance absolue de l'instance $\tau_{k,i}$
P, r, C, D, T	Caractéristiques d'un système de tâches périodiques (voir page 11)
δ_k	Nombre d'instances générées par la tâche τ_k sur chaque métapériode (voir page 11)
s	Séquence d'exécution (voir page 18)
s_t	Ensemble des instances exécutées par la séquence s dans l'intervalle $[t, t+1[$
$s_t^\star$	Ensemble des instances actives dans la séquence s à l'instant t
$e_s(\tau_{k,i})$	Date de terminaison de l'instance $\tau_{k,i}$ dans la séquence s
$TR_s(\tau_{k,i})$	Temps de réponse de l'instance $\tau_{k,i}$ dans la séquence s
$CE_s(\tau_{k,i})$	Charge de l'instance $\tau_{k,i}$ absorbée dans l'intervalle $[0, t[$ dans la séquence s
$LX_s(\tau_{k,i})$	Laxité de l'instance $\tau_{k,i}$ à l'instant t dans la séquence s
U	Charge du système de tâches τ (voir page 19)
$U_s(t)$, $\bar{U}_s(t)$, $\dot{U}_s(t)$	Respectivement, charge injectée, absorbée et restante
$\tau_i \succ \tau_j$	Relation de précédence entre τ_i et τ_j (voir page 24)
PFX	Ordonnancement en priorités fixes (voir page 17)
PFI	Ordonnancement en priorités fixes par instance
DCL	Ordonnancement déterministe basé sur la laxité et l'état d'avancement des instances actives
DYN	Ordonnancement en priorités dynamiques
$\overline{\text{PFI}}$	Ordonnancement en priorités fixes par instance où les relations de priorités entre les instances équivalentes sont préservées entre deux métapériodes (voir page 38)
$\overline{\overline{\text{PFI}}}$	Ordonnancement en priorités fixes par instance où deux instances équivalentes ont la même priorité (voir page 38)

Bibliographie

[1] I. Ahmad and M.K. Dhodhi. Multiprocessor Scheduling in a Genetic Paradigm. Parallel Computing, 22 :395–406, 1996.

[2] R. Alur and D. Dill. A theory of timed automata. Theoretical Computer Science, 126 :182–235, 1994.

[3] K. Altisen, G. Gössler and J. Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. Real-Time Systems, 23(1-2) :55–84, 2002.

[4] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson and W. Yi. TIMES : a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. International Workshop on Formal Modeling and Analysis of Timed Systems, LNCS, 2791 :60–72, 2003.

[5] B. Andersson, S. Baruah and J. Jonsson. Static priority scheduling on multiprocessors. Real-Time Systems Symposium, pp. 193–202, 2001.

[6] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non partitioned multiprocessor scheduling. Real-Time Systems Symposium, Work-In-Progress Session, pp. 53–56, 2000.

[7] B. Andersson and J. Jonsson. Preemptive Multiprocessor Scheduling Anomalies. International Parallel and Distributed Processing Symposium, pp. 12–19, 2002.

[8] N.C. Audsley, A. Burns, M.F. Richardson, K.W. Tindell and A.J. Wellings. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. Software Engineering Journal, 8(5) :284–292, 1993.

[9] N.C. Audsley, K. Tindell, and A. Burns. The End of the Road for Static Cyclic Scheduling ? Euromicro Workshop on Real-Time Systems, pp. 36–41, 1993.

[10] T.P. Baker. Stack-based schedulinf of real-time processes. Real-Time Systems, 3 :67–99, 1991.

[11] T.P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. Real-Time Systems, 32(1-2) :49–71, 2006.

[12] S.K. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. IEEE Transactions on Computers, 52(7) :966-970, 2003.

[13] S.K. Baruah, R.R. Howel and L.E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. Real-Time Systems, 2 :301–324, 1990.

[14] G. Berry. The foundations of Esterel Proof, Language and Interaction : Essays in Honour of R. Miller, G. Plotkin, C. Stirling and M. Tofte, MIT press, 2000.

[15] G. Berry, and G. Gonthier. The Esterel synchronous programming language : design, semantics, implementation. Science of Computer Programming, 19(2) :83–152, 1992.

[16] J. Blazewicz. Scheduling dependant tasks with different arrival times to meet deadlines. Modeling and performance evaluation of computer systems, pp. 57–65, 1976.

[17] C. Braun and L. Cucu. Negative Results on Idle Intervals and Periodicity for Multiprocessor Scheduling under EDF. Junior Researcher Workshop on Real-Time and Network Systems, 2007.

[18] G. Bucci, A. Fedeli, L. Sassoli and E. Vicario. Modeling Flexible Real Time Systems with Preemptive Time Petri Nets. Euromicro Conference on Real-Time Systems, pp. 279–286, 2003.

[19] G.C. Buttazzo. Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications. Kluwer Academics Publishers, 381 pages, 1997.

[20] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms. Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Computer and Information Science Series, pp. 30–60, Chapman Hall/ CRC Press. 2004.

[21] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offset. Theoretical Computer Science, 310(1-3) :117–134, 2004.

[22] A. Choquet-Geniet. Un premier pas vers l'étude de la cyclicité en environnement multi-processeur. Proceedings of Real-Time Systems, pp. 289–302, 2005.

[23] R.C. Corrêa, A. Ferreira and P. Rebreyend. Scheduling Multiprocessor Tasks with Genetic Algorithms. IEEE Transactions on Parallel and Distributed Systems, 10(8) :825–837, 1999.

[24] L. Cucu and J. Goossens. Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors. International Conference on Emerging Technologies and Factory Automation (ETFA'06), pp. 397–405, 2006.

[25] L. Cucu and J. Goossens. Feasibility Intervals for Multiprocessor Fixed-Priority Scheduling of Arbitrary Deadline Periodic Systems. Design Automation and Test in Europe (DATE'06), 2006. To appear (6 pages).

[26] B. Chauvière, D. Geniet. Une approche markovienne pour l'étude de systèmes temps-réel à contraintes strictes. Technique et Science Informatique, in press.

[27] M. Chen and K. Lin. Dynamic priority ceilings : a concurrency protocol for real-time systems. Real-Time Systems, 2(4) :325–346, 1990.

[28] H. Chetto, M. Silly and T. Bouchentouf. Dynamic scheduling of real-time systems under precedence constraints. Real-Time Systems, 2 :181–194, 1990.

[29] Groupe de Réflexion Temps-Réel du CNRS. Le temps-réel. Technique et Science Informatique, 7(5) :493–500, 1988.

[30] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. Real-Time Systems, 18(2) :249–274, 2000.

[31] F. Comets, F. Delarue and R. Schott. Distributed algorithms in an ergodic markovian environment. Random Structures and Algorithms, 30(1-2) :131–167, 2007.

[32] R.W. Conway, W.L. Maxwell and L.W. Miller. Theory of Scheduling Addison-Wesley, 1967.

[33] M.L. Dertouzos. Control robotics : the procedural control of physical processors. Proceedings of IFIP Congress, pp. 807–813, 1974.

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

[35] S. Faucou, A.M. Déplanche and J.P. Beauvais. Heuristic techniques for allocating ans scheduling communicating periodic tasks in distributed real-time systems. IEEE International Workshop on Factory Communication Systems (WFCS), pp. 257–266, 2000.

[36] W. Feller. Introduction to probability theory and its applications. vol. I, Wiley, 1971.

[37] W. Feller. Introduction to probability theory and its applications. vol. II, Wiley, 1971.

[38] P. Flajolet. The evolution of two stacks in bounded space and random walks in a triangle. Proceedings of FCT'86, Lecture Notes in Computer Science 233, pp. 325–340, 1986.

[39] J. Françon. A quantitative approach of mutual exclusion. RAIRO, 20 :275–289, 1986.

[40] M.R. Garey and D.S. Johnson. Computers and Intractability : A Guide to the Theory of NP-Completeness. New York, W.H. Freeman, 1979.

[41] D.E. Goldberg. Genetic Algorithms in Search, Oprtimization, and Machine Learning. Addison-Wesley, 1989.

[42] J. Goossens and R. Devillers. The non-optimality of the monotonic priority assignments for hard real-time offset free systems. Journal of Real-Time Systems, 13(2) :107–126, 1997.

[43] J. Goossens and R. Devillers. Feasibility intervals for the deadline driven scheduler with arbitrary deadlines. International Conference on Real-time Computing Systems and Applications, pp. 54–61, 1999.

[44] J. Goossens, S. Funk and S. Baruah. EDF scheduling on multiprocessors : some (perhaps) counterintuitive observations. International Conference on Real-Time Computing Systems and Applications, pp. 321–330, 2002

[45] E. Grolleau and A. Choquet-Geniet. Off-line computation of real-time schedules by means of Petri nets. Journal of Discrete Event and Dynamic Systems, 12 :107–126, 2002.

[46] P. Le Guernic, M. Le Borgne, T. Gauthier and C. Le Maire. Programming real-time applications with Signal. Proceedings of the IEEE, 79(9) :1321–1336, 1991.

[47] N. Guillotin-Plantard and R. Schott. Distributed algorithms with dynamical random transitions. Random Structures and Algorithms, 21(3-4) :371–396, 2002.

[48] N. Guillotin-Plantard and R. Schott. Dynamic Random Walks, Theory and Applications. Elsevier, 280 pages, 2006.

[49] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous dataflow programming language Lustre. Proceedings of the IEEE, 79(9) :1305–1320, 1991.

[50] P.K. Harter. Response Times in Level Structured Systems. ACM Transactions on Computer Systems, 5(3) :232–248, 1987.

[51] J.F. Hermant, L. Leboucher and N. Rivierre. Real-time fixed and dynamic priority driven scheduling algorithms : theory and experience. Rapport de recherche INRIA n°3081, 141 pages, 1996.

[52] E.S. Hou, N. Ansari and H. Ren. A Genetic Algorithm for Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems, 5(2) :113–120, 1994.

[53] L. Idoumghar, M. Alabau and R. Schott. New Hybrid Genetic Algorithms for the Frequency Assignment. IEEE Transactions on Broadcasting, 48(1) :27–34, 2002.

[54] L. Idoumghar, J.Y. Greff and R. Schott. Application of Markov Decision Processes to the Frequency Assignment Problem. Journal on Applied Artificial Intelligence, 18(8) :761–773, 2004.

[55] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Research Report UCLA, Management Sciences Research Project, 1955.

[56] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. The Computer Journal (British Computer Society), 29(5) :390–395, 1986.

[57] M.H. Klein, T. Ralya, B. Pollak, R. Obenza and M.G. Harbour. A practitioner's handbook for real-time analysis : guide to rate monotonic analysis for real-time systems. Kluwer Academics Publishers, 712 pages, 1994.

[58] D.E. Knuth. The art of computer programming, vol. 1, 1973.

[59] Y. Kwok, and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys, 31(4) :406–471, 1999.

[60] Y. Kwok, and I. Ahmad. Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm. Journal of Parallel and Distributed Computing, 47(1) :58–77, 1997.

[61] J. Labetoulle. Un algorithme optimal pour la gestion des processus en temps-réel. Revue française d'Automatique, Informatique et Recherche Opérationnelle, pp. 11–17, Février 1974.

[62] G. Largeteau, B. Chauvière and D. Geniet. Une approche géométrique de la validation temps-réel. In proceedings of Real-Time Systems, pp. 289–302, Teknea, 2005.

[63] G. Largeteau, D. Geniet. Term Validation of Distributed Hard Real-Time Application. Conference on Implementation and Application of Automata, 2002.

[64] G. Largeteau, D. Geniet and E. Andrès. Discrete geometry applied in hard real-time system validation. DGCI, Lecture Notes in Computer Science, Vol. 3429, pp. 23-33, 2005.

[65] P. Laroche. GraphMDP : A New Decomposition Tool for Solving Markov Decision Processes. International Journal on Artificial Intelligence Tools, 10(3) :325–344, 2001.

[66] J.P. Lehoczky, L. Sha and J.K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In Real-Time Systems Symposium, pp. 261–270, 1987.

[67] J.P. Lehoczky, L. Sha and Y. Ding. The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case Behavior. Real-Time Systems Symposium, pp. 166–171, 1989.

[68] J.P. Lehoczky. Fixed priority scheduling of periodic tasks sets with arbitrary deadlines. Real-Time Systems Symposium, pp. 201–209, 1990.

[69] J.P. Lehoczky, L. Sha, J.K. Strosnider and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In *Foundations of Real-Time Computing : Scheduling and resource management*, Kluwer Academics Publishers, pp. 1–30, 1991.

[70] J.Y.T Leung and M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. Information Processing Letters, 11(3) :115–118, 1980.

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

[72] D. Lime, O.H. Roux. A Translation Based Method for the Timed Analysis of Scheduling Extended Time Petri Nets. Real-Time Systems Symposium, pp. 187–196, 2004.

[73] F. Lindemann. Sur le rapport de la circonférence au diamètre et sur les logarithmes népériens des nombres commensurables ou des irrationnelles algébriques. Comptes Rendus de l'Académie des Sciences, Paris, tome 95, 1882, pp. 72–74.

[74] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM, 20(1) :46–61, 1973.

[75] D. Liu, X.S. Hu, M.D. Lemmon and Q. Ling. Scheduling Tasks with Markov-Chain Based Constraints. Euromicro Conference on Real-Time Systems, pp. 157–166, 2005.

[76] J.M. Lopez, J.L. Diaz and D.F. Garcia. Minimum and maximum utilization bounds for multiprocesseur RM scheduling. Euromicro Conference on Real-Time Systems, pp. 67–75, 2001.

[77] G. Louchard. Some distributed algorithms revisited. Communications in Statistic and Stochastic Models, 11(4) :563–586, 1995.

[78] G. Louchard and R. Schott. Probabilistic analysis of some distributed algorithms. Random Structures and Algorithms, 2 :151–186, 1991.

[79] G. Louchard, R. Schott, M. Tolley and P. Zimmermann. Random walks, heat equations and distributed algorithms. Journal of Computational and Applied Mathematics, 53 :243–274, 1994.

[80] OSEK Group. OSEK/VDX Operating System Specification. http ://www.osek-vdx.org

[81] R. McNaughton. Scheduling with deadline and loss functions. Management Science, 12(1) :1–12, 1959.

[82] J.F. Mari and R. Schott. Probabilistic and statistical methods in computer science. Kluwer Academics Publishers, 252 pages, 2001.

[83] R.S. Maier. Colliding stacks : a large deviations analysis. Random Structures and Algorithms, 2 :379–420, 1991.

[84] R.S. Maier and R. Schott. Exhaustion of shared memory : stochastic results. Proceedings of WADS'93, Lecture Notes in Computer Science 709, pp. 494–505, 1993.

[85] H. Mitra and P. Ramanathan. A Genetic Approach For Scheduling Non-preemptive Tasks With Precedence and Deadline Constraints. Proceedings of the 26th Hawaii International Conference on Systems Sciences, 11 :556–564, 1993.

[86] A.K. Mok. Fundamental design problems for the hard real-time environments. PhD Thesis, MIT, 1983.

[87] Y. Monnier, J.P. Beauvais and A.M. Déplanche. A Genetic Algorithm for Scheduling Tasks in a Real-Time Ditributed System. Proceedings of the 24th Euromicro Conference, 2 :708–714, 1998.

[88] M.A. Al-Mouhamed. Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs. IEEE Transactions on Software Engineering, 16(12) :1390–1401, 1990.

[89] M.L. Puterman. Markov Decision Processes. Wiley, 1994.

[90] D.I. Oh and T.P. Baker. Utilization bounds for N-processor rate monotone scheduling with stable processor assignment. Real-Time Systems, 15(2) :183–193, 1998.

[91] P. Puschner and A. Burns. Guest editorial : A review of worst case execution time analysis. Real-Time Systems, 18 :115–128, 2000.

[92] P. Richard and F. Cottet. Ordonnancement temps-réel monoprocesseur avec contraintes de précédence simple et généralisé. Rapport de Recherche LISI n°9902, 1999.

[93] I. Ripoll, A. Crespo and A.K. Mok. Improvement in feasability testing for real-time tasks. Real-Time Systems, 11(1) :19–39, 1996.

[94] RealTime Application Interface for Linux. http ://www.rtai.org

[95] RTLinux. http ://www.rtlinux-gpl.org

[96] O. Serlin. Scheduling of time critical processes. Proceedings of Spring Joint Computers Conference, pp. 925–932,1972.

[97] L. Sha, R. Rajkumar and J.P. Lehoczky. Priority inheritance protocols : an approach to real-time synchronization. Technical Report CMU-CS-87-181, Carnegie-Mellon, 23 pages, 1987.

[98] L. Sha, R. Rajkumar and J.P. Lehoczky. Priority inheritance protocols : an approach to real-time synchronization. IEEE Transactions on Computers, 39(9) :1175–1185, 1990.

[99] B. Sprunt and L. Sha. Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System. Technical Report CMU/SEI-89-TR-11 ADA211344, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1989.

[100] B. Sprunt, L. Sha and J.P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. Real-Time Systems, 1(1) :27–60,1989.

[101] M. Spuri and J.A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. IEEE Transactions on Computers, 43(12) :1407–1412, 1994.

[102] M. Spuri and G.C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. Real-Time Systems, 10(2) :179–210, 1996.

[103] J.A. Stankovic. Misconceptions about real-time computing. IEEE Transactions on Computers, 21(10) :10–19, 1988.

[104] J.A. Stankovic, M. Spuri, K. Ramamritham and G.C. Buttazzo. Deadline Scheduling for Real-Time Systems : EDF and Related Algorithms. Kluwer Academic Publishers, 1998.

[105] Y. Trinquet. Les systèmes d'exploitation temps-réel. Ecole d'été temps-réel (ETR'05), pp. 123–134, 2005.

[106] VxWorks. http ://www.windriver.com/vxworks

[107] R. White, F. Mueller, C. Healy, D. Whalley and M. Harmon. Timing analysis for data caches and set-associative caches. IEEE Real-Time Technology and Applications Symposium, pp. 192–202, 1997.

[108] A.S. Wu, H. Yu, S. Jin, K.C. Lin and G. Schiavone. An Incremental Genetic Algorithmic Approach to Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems, 15(9) :824–834, 2004.

[109] A. Yao. An analysis of a memory allocation scheme for implementing stacks. SIAM Journal on Computing, pp. 398–403, 1981.

[110] N. Zhang, A. Burns and M. Nicholson. Pipelined processors and worst case execution times. Real-Time Systems, 5(4) :319–343, 1993.

[111] Q. Zheng and K.G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched network. IEEE Transactions on Communications, 42(2,3,4), 1994.

[112] A.Y. Zomaya, C. Ward and B. Macey. Genetic Scheduling for Parallel Processor Systems : Comparative Studies and Performance Issues. IEEE Transactions on Parallel and Distributed Systems, 10(8) :795–812, 1999.

Résumé

Nos recherches portent sur l'étude des systèmes temps-réel composés de tâches périodiques et ordonnancés en environnement multiprocesseur. Nos travaux s'organisent autour de deux thèmes de recherche : la production de solutions d'ordonnancement et l'analyse quantitative de l'ordonnançabilité. Pour mener à bien ces recherches, nous avons été amenés à étudier certaines propriétés théoriques comme la cyclicité des séquences d'exécution. Sous certaines hypothèses, englobant par exemple RM, DM et EDF, nous montrons que les séquences d'exécution multiprocesseur sont cycliques, et nous proposons un intervalle de faisabilité.

Nous apportons deux contributions au problème de l'ordonnancement. D'une part, nous proposons une méthode pour déterminer toutes les configurations de priorités fixes permettant d'ordonnancer un système de tâches. Nous étendons cette approche aux configurations de priorités fixes par instance et proposons l'étude d'une nouvelle classe de politiques d'ordonnancement. D'autre part, nous appliquons les méthodes d'approximation (méthode taboue, algorithmes génétiques, etc) au problème de l'ordonnancement des systèmes de tâches liées par des contraintes de précédence, afin d'obtenir des séquences d'exécution optimisant certains critères comme la durée totale d'exécution par exemple.

Lorsque aucune politique connue n'est utilisable, le concepteur est généralement amené à modifier la conception du système. Peu de travaux permettent de le guider. Nous proposons une méthode basée sur les chaînes de Markov qui renseigne sur le comportement d'un système de tâches en fonction du support matériel utilisé. Les informations apportées permettent de comprendre les problèmes qui sont à la source des difficultés et donc d'aider à modifier la conception. Elles renseignent aussi sur la qualité de service offerte par le système.

Mots-clés : Temps-réel, Ordonnancement, Multiprocesseur, Processus de Markov, Qualité de service.

Abstract

We deal with real-time systems composed of periodic tasks designed to run on multiprocessor. We address two objectives : producing scheduling solutions and analyzing the feasibility of the systems we study in a quantitative way. Producing useful results leads us to study theoretical properties, e.g. the cyclicity of scheduling sequences. Assuming properties including RM, DM and EDF sequences, we show that multiprocessor scheduling sequences are cyclic, and we produce a feasibility interval.

Two contributions concern the scheduling problem. Firstly, we produce all static-priority configurations which schedule a specific task system, and we extend this technique to job-level static-priority policies. This study leads us to propose a new class of scheduling policies. Secondly, we use known approximation methods (e.g. tabu search, genetic algorithms, etc.) to solve the scheduling problem according precedence constraints and addressing the optimization of specific criteria (e.g. the total execution time).

When no known policy schedules the task system, the classically-used solution consists in modifying the system. Few works deal with this problem of real-time conception-aid-design. In this thesis, we propose a Markov chain-based technique which informs on the behaviours of the tasks on a hardware-dependency way. Informations produced concern the quality of service of the different possible scheduling solutions. When the system can not be scheduled, they are useful to identify the origines of the failure. Therefore, they help the conceiver to modify the timing specification.

Keywords : Real-time, Scheduling, Multiprocessor, Markov processes, Quality of service.