proc. AGIS, Automatique Génie informatique Image Signal, pp.239-246, Angers,9-11/12/1997

Ordonnancement Optimal des Systèmes de Tâches Temps Réel à l'Aide de Réseaux de Petri

Emmanuel GROLLEAU, Annie CHOQUET−GENIET, Francis COTTET
LISI−ENSMA
Site du FUTUROSCOPE BP 109
86960 FUTUROSCOPE Cedex
E−mail grolleau@alienor.univ−poitiers.fr, ageniet@diane.univ−poitiers.fr, cottet@ensma.univ−poitiers.fr

Résumé : Nous présentons l'application d'une méthodologie de détermination de l'ensemble des séquences optimales d'ordonnancement de systèmes de tâches temps−réel partageant des ressources et se synchronisant. Les critères d'optimalité proposés sont la minimisation du temps de réponse, du taux de réaction ou la maximisation de la latence des tâches, ainsi que l'importance d'un ensemble de tâches par rapport à un autre. La méthodologie utilisée se base sur une représentation du système de tâches par des réseaux de Petri colorés à contraintes de successeur avec ensemble terminal.

Mots clés : Application temps−réel, réseau de Petri, ordonnancement optimal

1. Introduction

L'informatisation de nombreux domaines industriels est incontournable de nos jours, en particulier en ce qui concerne le contrôle de procédé. Qu'elles soient dédiées à l'assistance d'un opérateur accomplissant une tâche délicate, telle que le pilotage d'un avion, ou bien au contrôle du fonctionnement d'une chaîne de montage, les applications temps réel sont caractérisées par la nécessité absolue de respecter des contraintes temporelles strictes. En effet, une chaîne de montage automatique qui s'emballe fait subir de lourdes pertes financières à une entreprise, et un contrôle d'altitude défaillant, même uniquement au sens temporel (i.e. intervenant trop tard), peut provoquer un accident coûteux en vies humaines.

Une application temps réel peut être vue comme un système de tâches se synchronisant et partageant des ressources telles que le processeur dans les systèmes monoprocesseurs, les processeurs dans les systèmes répartis, ou la sortie vers un terminal de contrôle [Stankovic88]. Ces tâches sont communément classées en deux catégories: les tâches périodiques dédiées le plus souvent au contrôle, qui peuvent par exemple scruter l'altitude d'un avion, et les tâches sporadiques, dont le déclenchement n'est pas prévisible, qui peuvent être activées par des tâches périodiques dans certains cas d'urgence. Par exemple une tâche qui redressera un avion pourra être activée par une tâche de contrôle d'altitude. Ces tâches doivent non seulement être correctes algorithmiquement, mais aussi temporellement : chaque tâche doit s'exécuter intégralement dans un intervalle de temps de taille fixe, qui dépend de la périodicité de la tâche, des contraintes imposées par le concepteur, des caractéristiques physiques du système, etc... L'échéance est la date à laquelle la tâche doit avoir terminé son exécution, et le délai critique est le temps imparti à la tâche pour s'exécuter après chacune de ses activations. Celui-ci est toujours de durée inférieure ou égale à la période dans le cas d'une tâche périodique. Si le délai critique est égal à la période, on parle d'échéance sur requête. La correction temporelle d'une application temps réel est étroitement liée, une fois la correction algorithmique des tâches établie, à la politique d'ordonnancement choisie. En effet la manière dont on attribue le ou les processeurs aux différentes tâches à un instant donné est déterminante pour la correction du système. C'est pour cette raison qu'en informatique temps réel, beaucoup de travaux proposent des politiques d'ordonnancement de systèmes de tâches temps réel. Ces politiques d'ordonnancement ont toutes le même objectif : permettre aux tâches de respecter leur échéance.

Les tâches temps réel que nous considérons sont périodiques, ont un délai critique, et une date de première activation pouvant être non nulle.

Une tâche temps réel est donc caractérisée temporellement [Liu-Layland73] par trois valeurs numériques données par le concepteur de l'application : la période, le délai critique, et la date de première activation. La charge induite, ou durée d'exécution, est déduite du code composant la tâche :

- r_i : date de première activation (de réveil)
- T_i : période
- R_i : délai critique
- C_i : charge induite (durée d'exécution)

Il est clair que Ci dépend du matériel utilisé, donc la durée de chaque instruction doit être paramétrable [Cottet-Babau94] [Pushner-Koza89]. Dans la suite nous considérons le paramètre C_i connu et nous dénotons un système de tâches temps réel composé de n tâches τ_i (i=1..n) par $\{ \tau_1 <r_1, C_1, R_1, T_1>, \tau_2 <r_2, C_2, R_2, T_2>, ..., \tau_n <r_n, C_n, R_n, T_n>\}$ ou plus brièvement par $\{ \tau_i <r_i, C_i, R_i, T_i> \}_{i = 1..n}$. L'échéance d d'une tâche au temps t est la prochaine date à laquelle celle−ci doit avoir terminé son exécution. La latence d'une tâche est à tout instant t le temps d−t qu'il lui reste pour s'exécuter avant sa prochaine échéance d. La laxité d'une tâche est donnée par sa latence moins sa durée d'exécution restante.

Dans une première partie nous rappelons quelques résultats d'ordonnancement, puis nous présentons la modélisation par réseau de Petri utilisée, et enfin nous décrivons l'étude du graphe d'accessibilité obtenu pour extraire les ordonnancements optimaux au vu de certains critères.

2. Résultats fondamentaux en matière d'ordonnancement de systèmes temps réel

Beaucoup de politiques d'ordonnancement sont basées sur la notion de priorité. Les algorithmes se différencient

par leur politique d'attribution des priorités, statique ou dynamique, et par le fait qu'ils sont préemptifs ou non. La plupart d'entre eux sont en ligne : de faible complexité ils sont implémentés dans un système d'exploitation temps réel et ordonnancent les tâches d'un système. Nous verrons que la méthode proposée ici calcule la ou les meilleures séquences d'ordonnancement avant de les implanter dans un séquenceur : c'est de l'ordonnancement hors ligne.

L'algorithme d'ordonnancement à priorités dynamiques le plus répandu est ED *(Earliest Deadline)* dans lequel la priorité d'une tâche est inversement proportionnelle à sa latence. Dans sa version préemptive, celui-ci est optimal [Liu-Layland73] dans la classe des algorithmes à priorités dynamiques pour les systèmes de tâches périodiques indépendantes. Rappelons qu'un algorithme est optimal dans sa classe d'algorithmes pour une famille de tâches donnée si, pour tout système de tâches de la famille et pour tout algorithme de la classe donnée, le système n'est pas ordonnançable, ou bien il est ordonnançable par un ensemble d'algorithmes, dont l'algorithme optimal fait partie.

L'algorithme à priorités statiques le plus répandu est RM *(Rate Monotonic)*, dans lequel chaque tâche se voit assigner une priorité inversement proportionnelle à sa période (ou à son délai critique pour *Deadline Monotonic*). RM, dans sa version préemptive, a été prouvé optimal dans [Liu-Layland73] dans la classe des algorithmes à priorité statique préemptifs pour des systèmes de tâches périodiques indépendantes avec échéance sur requête. De plus il minimise le taux de réaction du système [Peng-Shin93]. Ce résultat n'est cependant valide que pour les systèmes de tâches indépendantes.

Il existe des algorithmes classiques d'ordonnancement optimaux pour les systèmes de tâches indépendantes, mais comme tout système multitâche, un système temps réel doit permettre aux tâches d'accéder à des ressources partagées. Les problèmes liés au partage de ressources, tels que l'inversion de priorité ou les interblocages invalident les résultats d'optimalité des algorithmes en ligne classiques. Les protocoles de gestion de priorité couplés aux algorithmes d'ordonnancement classiques (protocole à priorité plafond, protocole à priorité héritée [Sha-Rajkumar-Lehoczky90]), bien que fournissant de bons résultats, ne sont pas optimaux.

Il n'existe pas, à ce jour, d'algorithme polynomial permettant d'ordonnancer des systèmes de tâches partageant des ressources et le problème d'ordonnancement de tâches temps réel partageant des ressources est NP difficile [Leung-Merrill80].

En utilisant un modèle de représentation des systèmes de tâches temps réel basé sur les réseaux de Petri et proposé dans [Choquet_Geniet-Geniet-Cottet96], nous proposons un algorithme capable de calculer toutes les séquences d'ordonnancement possibles pour un système de tâches temps réel périodiques en environnement monoprocesseur, qui partagent des ressources et qui communiquent par message. Dans le cadre de cet article, nous nous limitons aux systèmes de tâches synchrones au démarrage (i.e. $r_i=0\ \forall\tau_i$). L'extension de l'algorithme aux systèmes de tâches non synchrones au démarrage est à l'étude.

Cet algorithme est bien sûr NP, mais nous utilisons des techniques permettant de réduire le temps de construction du graphe d'accessibilité du réseau, ainsi l'algorithme est utilisable en pratique. Puis nous présentons un algorithme permettant d'extraire l'ensemble des séquences optimales au vu de certains critères en un temps linéaire en fonction de la taille du graphe d'accessibilité du réseau.

3. Modélisation de systèmes de tâches temps réel à l'aide des réseaux de Petri

Dans ce paragraphe nous présentons la modélisation de systèmes de tâches temps réel à l'aide des réseaux de Petri colorés avec marquages terminaux proposée dans [Choquet_Geniet-Geniet-Cottet96]. Pour une définition des réseaux de Petri, le lecteur peut se reporter à [Choquet_Geniet-Vidal_Naquet93]. P est l'ensemble des places du réseau, T l'ensemble des transitions, et M : $P\rightarrow N\cup\{a\}\cup\{b\}$ la fonction de marquage qui donne les jetons contenus dans une place où N est l'ensemble des entiers naturels, et a et b sont des couleurs de jetons.

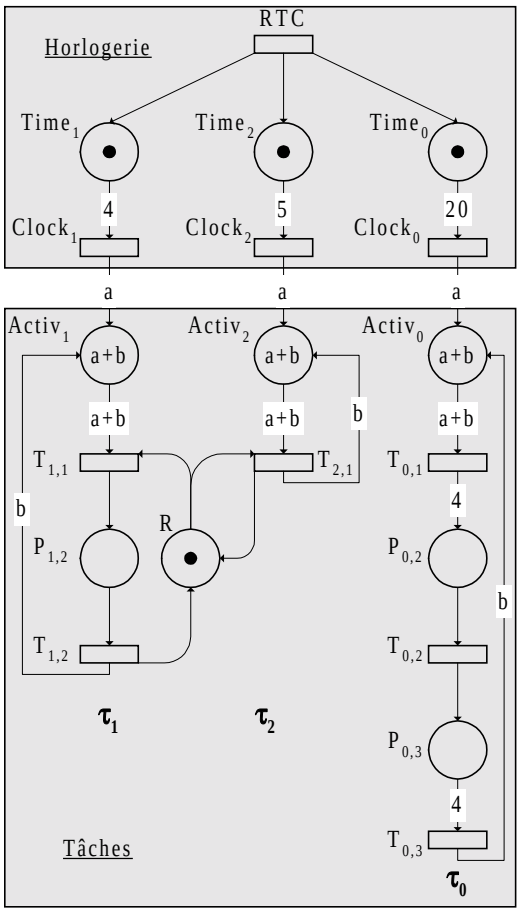

Figure 1: Modélisation d'un système de 3 tâches à l'aide d'un réseau de Petri

Nous donnons sur la figure 1 la modélisation d'un système de tâches temps réel :

$S=\{ \tau_1 <0,2,4,4>, \tau_2 <0,1,1,5>, \tau_0 <0,6,20,20>\}$ où τ_1 et τ_2 partagent la ressource R pendant toute leur exécution.
Le modèle se décompose en deux parties:

- La structure temporelle qui contient

- l'horloge globale notée RTC (*Real Time Clock*) qui tire un jeton à chaque unité de temps dans chacune des places composant les horloges locales.

- les horloges locales des tâches qui permettent de réactiver les tâches à chaque fin de période de celles-ci. L'horloge locale d'une tâche τ_i est composée d'une place accumulatrice de temps $Time_i$, qui reçoit à chaque unité de temps un jeton de l'horloge globale RTC. Lorsqu'elle contient T_i jetons (toutes les périodes), elle permet à la transition Clk_i de tirer un jeton *a* (pour activation) dans la première place de la partie système de tâches de la tâche τ_i.

- Le système de tâches périodiques, mises en parallèle, qui sont représentées de manière classique. La première place de chaque tâche τ_i, nommée $Activ_i$, peut contenir deux couleurs de jetons : *a* pour tâche activée, et *b* pour tâche qui a fini son exécution lors de sa dernière activation. La notation a+b signifie « un jeton de couleur a et un jeton de couleur b ». Lorsqu'une tâche τ_i reçoit un jeton *a*, elle est réactivée. A ce moment-là elle doit avoir fini son exécution lors de sa dernière activation, ce qui signifie que soit elle reçoit un jeton *b* (la tâche a eu une latence de 0 pendant sa dernière période), soit elle en contient déjà un (sa latence était plus grande que 0). Si la tâche n'est pas à échéance sur requête, lorsque l'horloge locale de la tâche dépasse le délai critique R_i, la tâche doit avoir terminé son exécution, ce qui veut dire que $M(Time_i)>R_i \Rightarrow M(Activ_i)\geq b$. Si la tâche est à échéance sur requête, T_i-1 unités de temps après (ré)activation de τ_i, $M(Time_i)=T_i$, et $M(Activ_i)\leq b$. Une unité de temps plus tard, $M(Time_i)=1$ et $Activ_i$ doit contenir un jeton *a* et un jeton *b*. Donc, comme $M(Time_i)=1$ uniquement lorsque la tâche τ_i vient d'être (ré)activée, il est nécessaire, pour que la tâche respecte son échéance, que $M(Time_i)=1 \Rightarrow M(Activ_i)=a+b$. Les échéances des tâches, qu'elles soient à échéance sur requête ou non, nécessitent donc que

$(M(Time_i)>R_i \Rightarrow M(Activ_i)=b)$ (1)
et
$(M(Time_i)=1 \Rightarrow M(Activ_i)=a+b)$ (2)

pour un marquage donné M.

Cette contrainte est utilisée pour définir le marquage terminal du modèle. Un marquage n'est valide que si il fait partie de l'ensemble terminal. Il doit donc respecter les contraintes (1) et (2).

Les transitions du système de tâches sont mises en exclusion mutuelle à l'aide d'une place processeur qui n'est pas représentée graphiquement par raison de lisibilité. Cela impose qu'une seule tâche peut avancer à un instant donné puisque nous faisons fonctionner le réseau en parallèle à vitesse maximale (on tire le plus de transitions possibles à chaque unité de temps).

Avec un tel fonctionnement, nous n'obtenons que des ordonnancements au plus tôt (i.e. les temps creux n'interviennent que lorsqu'aucune tâche n'est active). Pour obtenir tous les ordonnancements possibles, nous introduisons en plus une tâche oisive τ_0, qui modélise l'occurrence de temps creux dans les ordonnancements. Cette tâche a une période et une échéance égales au plus petit commun multiple (PPCM) des périodes des tâches du système, une durée d'exécution égale à
$PPCM(T_i)_{i=1..n} \times (1-U)$, U étant la charge totale du système de tâches : $U=\sum_{i=1}^{n} \frac{Ci}{Ti}$. Nous notons $H= PPCM(T_i)_{i=1..n}$ l'hyperpériode du système de tâches. La tâche oisive τ_0 est donc caractérisée par $<0,H\times(1-U), H, H>$.

Sur la figure 1, τ_0 est la tâche oisive associée au système $S'=\{ \tau_1 <0,2,4,4>, \tau_2 <0,1,1,5>\}$. Sans l'adjonction de cette tâche, le système S' n'est pas ordonnançable par le réseau de Petri car S' n'est pas ordonnançable au plus tôt. En effet pour les 54 ordonnancements possibles de ce système, lors de son premier réveil, τ_1 doit être prioritaire sur τ_0 alors qu'à son second réveil, c'est τ_0 qui doit être prioritaire sur elle : la tâche τ_1 doit être mise en attente bien qu'elle soit la seule tâche active pour permettre à la tâche τ_2 de respecter son échéance au temps 6 : en effet l'utilisation de la ressource R rend ces tâches non préemptives l'une par l'autre. Ceci est en contradiction avec le fonctionnement au plus tôt des algorithmes classiques. S' n'est donc ordonnançable par aucun algorithme classique (ED, RM, DM, avec ou sans utilisation du protocole à priorité plafond ou à priorité héritée [Sha-Rajkumar-Lehoczky90], assignement statique des priorités,...).

Remarque : Si on étiquette les transitions $T_{i,j}$ des systèmes de tâches par la lettre τ_i et les autres par le mot vide, le langage du réseau de Petri (ensemble des mots pouvant être formés par tout comportement valide du réseau de Petri, les mots étant donnés par les lettres étiquetant les transitions mises bout à bout) est exactement l'ensemble des ordonnancements possibles du système de tâches modélisé.

4. Ordonnancement d'un système

On peut donc, en construisant le graphe des marquages du réseau de Petri, dont les arcs sont étiquetés par l'alphabet $\{\tau_0, \tau_1, .., \tau_n\}$, obtenir tout ordonnancement valide du système. Il convient de borner la profondeur du graphe d'accessibilité (i.e. le temps de simulation) : au temps 0, toutes les tâches sont prêtes et viennent d'être activées, $H=PPCM(T_i)_{i=1..n}$ unités de temps plus tard, le système se retrouve dans la même configuration. Il suffit donc d'étudier les ordonnancements sur une période de H unités de temps, puisque l'état du système boucle sur cette période.

La taille du graphe d'accessibilité du réseau de Petri est exponentielle en nombre de noeuds, et sa construction entraîne du *backtracking*, retour arrière dû au fait que l'on doit pousser la construction de ce graphe soit jusqu'à la profondeur H, dans ce cas, la séquence est valide, soit jusqu'à un blocage du réseau (interblocage du système dû aux ressources ou interdépendance de tâches), soit jusqu'à ce qu'on s'aperçoive qu'une tâche ne peut pas respecter sa

prochaine échéance. C'est pour cela que nous utilisons des heuristiques permettant de réduire la taille du graphe et de réduire le *backtracking*.

4.1 Calcul de l'ensemble des ordonnancements

La construction du graphe des marquages se fait en profondeur d'abord jusqu'à la profondeur H. L'algorithme de construction du graphe des marquages d'un réseau de Petri avec ensemble terminal (noté Ω) est composé d'appels récursifs à un algorithme d'ajout d'un noeud dans le graphe des marquages .

```
Fonction Ajouter_Noeud (
        G: IN OUT Noeud,
        M: IN Marquage,
        P: IN Profondeur,
        RdP: IN Réseau_de_Petri)
        retourne booléen
-- retourne vrai si et seulement si le marquage M fait
-- partie d'une séquence valide
Début
  Si P = PPCM(Ti)
    Alors retourner M∈Ω
  Finsi
  NoeudViable: Booléen ← Faux
  Pour toute tâche τ avançable au sens usuel faire
    M'←Avancer(RdP,M,τ)
    Si M'∈ Ω alors
      Si M' n'existe pas déjà à la profondeur P alors
        G'← Créer_Noeud(G,M',τ)
        Si Ajouter_noeud(G',M',P+1,RdP)=Vrai alors
          NoeudViable←Vrai
          --NoeudViable est vrai ssi M fait partie d'une
          --séquence valide
        FinSi
      FinSi
    FinSi
  Fait
  Si NoeudViable alors retourner Vrai
  Sinon Détruire(G,G')
    retourner Faux
  FinSi
Fin

Appel de la fonction:
G: Noeud ← M0
Si Ajouter_Noeud(G, M0,0,RdP)=Faux
Alors G←∅
```

Une tâche τ est « avançable au sens usuel » si la transition correspondante dans la partie système de tâches du réseau de Petri est franchissable au sens habituel.

La fonction *Avancer* prend en paramètres un réseau de Petri, un marquage M, et la tâche que l'on veut faire avancer. Cette fonction tire toutes les transitions de la partie temporelle du réseau, et la transition franchissable de la tâche τ de la partie système de tâches.

La fonction *Créer_Noeud* prend en entrée un noeud *G*, un marquage *M'* et le nom d'une tâche, crée dynamiquement un noeud contenant le marquage *M'* et le relie à *G* par un arc étiqueté par le nom de la tâche.

La fonction *Détruire* détruit l'arc qui relie les noeuds *G* et *G'* et libère la mémoire occupée par *G'*.

Si le système est ordonnançable, au retour de la fonction *Ajouter_Noeud*, le noeud passé en paramètre est l'entrée d'un graphe de profondeur H dont les étiquettes des arcs donnent tous les ordonnancements valides du système. Ce graphe possède un noeud de hauteur 0 et un noeud de hauteur H, qui contiennent le même marquage. Le graphe n'a qu'une entrée G_0 et qu'une sortie G_P, et tout chemin de G_0 à G_P est étiqueté par un ordonnancement valide du système de tâches.

4.2 Optimisations

L'algorithme de construction du graphe des marquages développe chaque séquence d'ordonnancement possible jusqu'à ce que:

(i) arrivé à la fin de la simulation (i.e. au temps H) il valide la séquence

ou

(ii) arrivé à un marquage non terminal (non valide) M' (dépassement d'une échéance) il déclare la séquence non valide. Dans ce cas la fonction retourne faux à la fonction appelante contenant le marquage M précédant M'. Si M ne fait partie d'aucune séquence valide, il est enlevé du graphe, et la fonction retourne faux à la fonction appelante, etc...

ou

(iii) blocage du réseau de Petri. Même façon de procéder qu'en *(ii)*.

4.2.1 Diminution du backtracking

Pour limiter le nombre de cas *(ii)*, nous avons optimisé l'algorithme de la manière suivante :

La validité d'un marquage n'est pas seulement donnée par le respect des échéances au temps de simulation considéré. Nous dirons qu'un marquage est valide si il respecte les échéances au temps considéré, et si il peut respecter les échéances à venir des tâches.

Pour cela nous utilisons une table de prévision du futur contenant pour chaque tâche (y compris la tâche oisive) le travail qu'il lui reste à effectuer avant sa prochaine échéance (donc en un temps égal à sa latence), et sa latence. Nous maintenons cette table triée dans l'ordre croissant des latences des tâches. Cette table est décrite sur le tableau 1.

$\tau_{t,0}$	$\tau_{t,1}$	...	$\tau_{t,n}$
Travail $\tau_{t,0}$	Travail $\tau_{t,1}$	...	Travail $\tau_{t,n}$
Latence $\tau_{t,0}$	Latence $\tau_{t,1}$	...	Latence $\tau_{t,n}$

tableau 1: Table de prévision

Le principe de cette table est le suivant : lorsque l'échéance d'une tâche $\tau_{t,i}$ expirera, les échéances des tâches $\tau_{t,j}$ telles que $j \leq i$ auront également expiré. Donc les tâches

$\tau_{t,0}$, $\tau_{t,1}$, ..., $\tau_{t,i}$ doivent pouvoir terminer leur exécution avant que l'échéance de $\tau_{t,i}$ ne tombe. Donc si il existe i tel que :

$$\sum_{j=0}^{i} \text{Travail } \tau_{t,j} \geq \text{Latence } \tau_{t,i}$$

alors une des tâches de $\{\tau_{t,0}, \tau_{t,1}, ..., \tau_{t,i}\}$ ne pourra pas respecter sa prochaine échéance. Ce qui est intéressant, c'est que ce test, ainsi que la mise à jour de la table ont une complexité linéaire en fonction du nombre de tâches.

Voici comment nous procédons pour mettre à jour la table de prévision à chaque unité de temps :

```
Précondition: la table T est ordonnée.
Procédure MaJ_de_table_de_prévision (
        T : IN OUT Table_de_prévision
        τ : Tâche
        )
Début
  Modification_d_ordre ← Faux
  Temporaire : Colonne_de_table_de_prévision
  Temporaire.tâche ← τ
  --Temporaire sert à stocker les paramètres de τ au cas
  --où il y a modification d'ordre de la table
  Pour i de 0 à n faire
    T[i].Latence ← T[i].Latence-1
    -- La tâche voit son échéance se rapprocher
      Si T[i].tâche = τ alors
        Si T[i].Travail = 1 alors
          -- La tâche qui progresse se termine
          Modification_d_ordre ← Vrai
          Temporaire.Travail ← τ.C_i
          --Elle devra s'exécuter totalement à sa
          --prochaine activation
          Temporaire.Latence ← T[i].Latence+τ.T_i
      --Et sa prochaine échéance est dans une
          --période
        Sinon T[i].Travail ← T[i].Travail –1
          -- La tâche progresse
        FinSi
      Sinon
        --La tâche ne progresse pas
      Si Modification_d_ordre
      et T[i].Latence< Temporaire.Latence alors
        -- La tâche sera avant τ dans la table
        T[i-1] ← T[i]
      SinonSi Modification_d_ordre alors
        --La tâche en ième position est la tâche qui
        --va être placée juste derrière τ dans la table
        T[i-1] ← Temporaire
        Modification_d_ordre ← Faux
        -- La place des tâches suivantes ne change pas
      FinSi
    FinSi
  Fait
  Si Modification_d_ordre alors
    -- τ a l'échéance la plus éloignée
    T[n] ← Temporaire
  FinSi
Fin
```

Cet algorithme utilise la propriété suivante: toutes les tâches, à l'exception de celle qui avance, voient leur échéance se rapprocher d'une unité de temps. Elles restent donc dans le même ordre les unes par rapport aux autres. Seule la tâche qui progresse peut voir le numéro de sa place modifié dans le cas où elle a terminé (*Modification_d_ordre* mis à vrai). Dans ce cas elle voit son échéance s'éloigner (ou rester constante dans le cas peu réaliste où sa période est de 1). Elle peut donc être repoussée plus loin dans la table.

Remarque : on peut utiliser les fonctions de gestion d'une table de prévision proposées ici dans un ordonnanceur ED préemptif. Dans le cas de systèmes de tâches indépendantes, il suffit à l'ordonnanceur de toujours faire progresser la première tâche de la table, ou de déclarer le système non ordonnançable si une tâche ne peut pas respecter sa prochaine échéance. L'algorithme ED ainsi obtenu utilise à chaque pas un temps en $\theta(n)$.

Il est difficile d'évaluer théoriquement le gain de temps apporté par cette optimisation. Mais il est peu coûteux en temps pour chaque noeud: $\theta(2n)$ par noeud par rapport au calcul d'un noeud qui est en plus de $\theta(n^2)$, de plus, sur l'exemple suivant, nous voyons qu'il peut faire gagner plus de 50% en temps.

Soient deux tâches indépendantes τ_1<0,10,10,20> et τ_2<0,10,20,20>. Le graphe des ordonnancements est donné en figure 2, sur lequel les arcs en gras représentent le graphe des ordonnancements valides, les arcs en trait fin représentent les noeuds calculés par l'algorithme optimisé, et les arcs en traits pointillés représentent les noeuds calculés par l'algorithme dans sa version non optimisée.

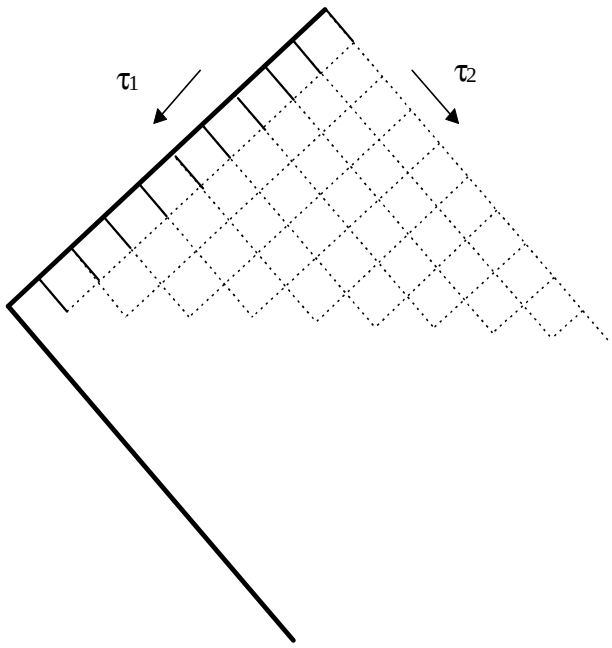

Figure 2 : Gain apporté par l'optimisation

4.2.2 Diminution de la taille du graphe

Pour diminuer la taille du graphe, nous utilisons des contraintes de successeurs sur le réseau de Petri lors de la construction du graphe d'accessibilité [Choquet_Geniet-Geniet-Cottet96]. Ces contraintes traduisent l'interdiction à toute tâche d'interrompre la tâche travaillant sauf dans les cas suivants :

- Une tâche qui vient d'être (ré)activée peut préempter la tâche travaillant
- Une tâche qui vient d'être débloquée par la réception d'un message ou la libération d'une ressource peut préempter la tâche travaillant
- Si la tâche travaillant demande son entrée en section critique, toute tâche peut la préempter.

4.3 Quelques indications sur la complexité

Nous calculons ici le nombre maximal de sommets du graphe des marquages d'un réseau. Nous définissons un hyperpavé de dimension d comme un pavé maillé de dimension d au lieu de 3 :

Définition : Un hyperpavé de dimension d et de longueurs L_1, L_2, ...,L_d est défini de la manière suivante sur d :

d=1 : suite de L_1+1 sommets s_0, s_1, ..., s_{L1} tels que pour tout $i \in 0..L_1-1$, un arc relie s_i à s_{i+1}.

d=n : L_n+1 hyperpavés h_0, ..., h_{Ln} de dimension n−1, de longueurs L_1, L_2, ..., L_{n-1}, tels que pour un étiquetage identique des sommets des hyperpavés h_0, ..., h_{Ln}, pour tout $i \in [0..L_n-1]$, le sommet s_i de h_i est relié par un arc au sommet s_i de h_{i+1}.

Le nombre de sommets d'un hyperpavé de dimension d et de longueurs L_1, L_2, ...,L_d est donné par $\prod_{i=1}^{d}(L_i+1)$ ♦

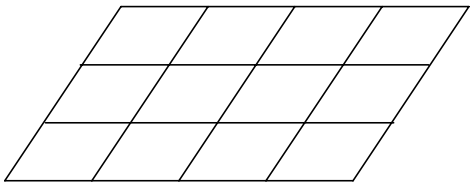

Figure 3 : Hyperpavé de dimension 2, de longueurs 4 et 3

Considérons un système $S=\{\tau_i <0, C_i, R_i, T_i>\}_{i = 1..n}$ et R le réseau de Petri le modélisant. $H=PPCM(T_i)_{i=1..n}$ est le temps de simulation de S. Chaque tâche τ_i est exécutée $\frac{H}{T_i}$ fois. Le corps de τ_i étant constitué de C_i transitions, sur le temps de simulation, $\frac{H \times C_i}{T_i}$ transitions de τ_i sont franchies. Au pire τ_i peut être interrompue à chaque instant par n'importe quelle transition d'une tâche τ_j, y compris par la tâche oisive τ_0. Le graphe d'accessibilité de R est donc contenu dans un hyperpavé de dimension n+1 et de longueurs respectives le nombre de transitions à franchir pour chaque tâche, y compris pour la tâche oisive, pendant le temps de la simulation. Il contient donc au plus

$$((1-U)H+1)\times\left(\frac{H\times C_1}{T_1}+1\right)\times \ \dots \ \times\left(\frac{H\times Cn}{Tn}+1\right)$$

sommets. Dans le pire des cas les périodes sont premières entre elles et $H=\prod_{i=1}^{n} T_i$. De plus, comme $C_i \le T_i$, le nombre de sommets du graphe est borné par $\left(1+\prod_{i=1}^{n} Ti\right)^{n+1}$. Il y a, évidemment, une borne très large et exponentielle aux graphes des marquages. Cependant, à cause des synchronisations et des exclusions mutuelles, la taille des graphes diminue. De plus, les échéances et les périodicités des tâches font que le graphe forme souvent une *grappe* autour de la diagonale de l'hyperpavé, allant du noeud d'entrée au noeud de sortie.

Pour exemple nous donnons en figure 4 (resp. 5) le graphe des ordonnancements du système de tâches S={t1<0,6,14,14>, τ_2<0,4,7,7>} (resp. {t1<0,9,21,21>, τ_2<0,4,7,7>}). Bien que très artificiel, pour entraîner une bonne lisibilité du graphe, cet exemple est typique des ordonnancements.

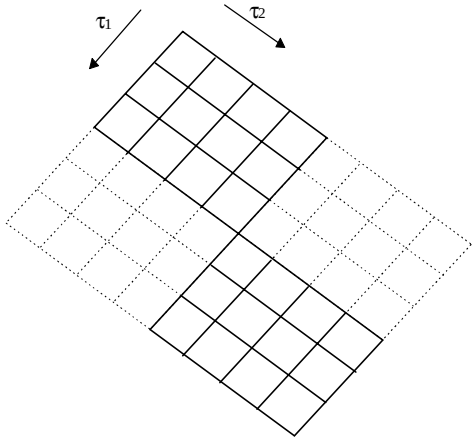

Figure 4 : Un graphe d'ordonnancement, la moitié de l'hyperpavé est utilisé

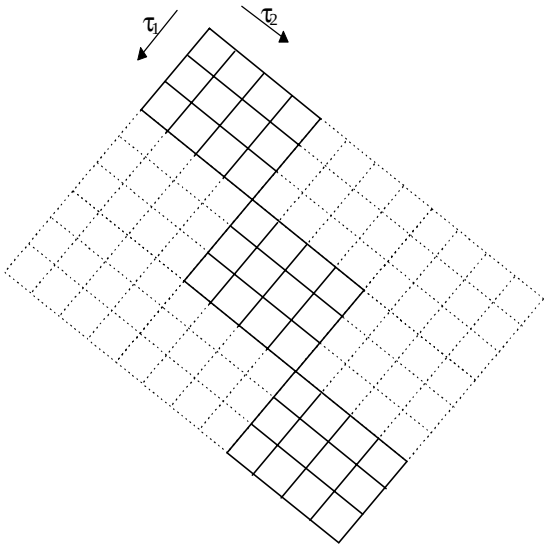

Figure 5 : Exemple de graphe d'ordonnancement, 1/3 de l'hyperpavé est utilisé

5. Choix des séquences optimales

Etant donné un graphe d'ordonnancement de taille N, et de profondeur P, le nombre de séquences possibles peut être exponentiel en N. Nous montrons ici une méthode d'extraction d'une séquence d'ordonnancement en fonction de différents critères (importance, minimisation du temps de réponse, maximisation du temps de retard admissible, etc... de certaines tâches ou ensembles de tâches). La séquence ϕ extraite par cette méthode est la meilleure séquence d'ordonnancement pour les critères donnés. Comme le système est dans le même état au début qu'à la fin de la séquence ϕ, ϕ répété à l'infini est le meilleur ordonnancement pour les critères donnés.

5.1 Critères d'optimisation

Nous présentons ici la liste des critères que nous avons étudiés pour un ensemble de tâches E d'un système de tâches S :

- Maximisation de l'importance : les tâches de E sont exécutées le plus tôt possible.
- Minimisation du temps de réponse maximal (ou moyen) : le temps maximal (ou moyen) entre l'activation et la terminaison des tâches de E est minimisé.
- Maximisation du temps de retard admissible minimal (ou moyen) : le temps minimal (ou moyen) entre la terminaison et l'échéance des tâches de E est maximisé.
- Minimisation du taux de réaction [Martineau-Silly94] maximal (ou moyen) : le temps maximal (ou moyen) du temps de réponse divisé par l'échéance des tâches de E est minimisé.

5.2 Algorithmes de choix de séquence

Nous présentons l'algorithme de base utilisé pour déterminer les séquences d'ordonnancement optimales permettant l'exécution des tâches d'un ensemble de tâches le plus tôt possible (maximisation de l'importance). Puis nous montrons quelles modifications lui apporter pour qu'il détermine les séquences optimales pour les autres critères donnés en 5.1.

Les arcs du graphe sont valués implicitement de la façon suivante : le coût d'un arc situé entre la profondeur n-1 et la profondeur n est n si il est étiqueté par une tâche importante et 0 sinon. Les ordonnancements optimaux sont donnés par l'ensemble des plus courts chemins allant du noeud initial au noeud terminal. Considérons pour un ordonnancement donné qu'une tâche τ de durée 3, que l'on veut exécuter le plus tôt possible, est exécutée aux instants 1,2 et 4. Le coût associé sur le graphe à cette tâche est 1+2+4=7. Si une autre branche fait que τ est exécutée aux instants 1,3 et 4, le coût associé est 8 et c'est le premier chemin qui permet l'exécution de τ le plus tôt possible.

L'algorithme proposé est un algorithme de pondération des noeuds du graphe tel que la pondération d'un noeud est donnée par le coût minimal des chemins allant de ce noeud au noeud terminal.

```
Procédure Pondérer (G: IN OUT Graphe_Marquages)
Début
  Pour temps décroissant de PPCM(Ti) à 0 faire
    Pour tout noeud N de G de hauteur temps faire
      Si temps = PPCM(Ti) alors
        Coût(N) ← 0
        -- Le coût de la sortie est nul
      Sinon
        Coût(N) ← Min(CoûtPriorité(G,N,Ni,temps)) ∀fils Ni de N
      FinSi
    Fait
  Fait
Fin
```

La fonction calculant le coût d'un noeud en fonction de ses fils est donnée ci-après:

```
Fonction Coût_Priorité (
        G: IN Graphe_des_Marquages,
        N, Ni: IN Noeud,
        Profondeur: IN Entier,
        ) retourne Entier
Début
-- Renvoie le coût du plus court chemin passant
-- par N et Ni et allant au noeud terminal
  CoûtN : Entier ← Coût(Ni)
  Si Etiquette_de_l_arc(G,N,Ni) ∈ E alors
  --E est l'ensemble des tâches à exécuter
  --le plus tôt possible
    CoûtN ← CoûtN + Profondeur +1
  FinSi
  retourner CoûtN
Fin
```

Une fois les noeuds pondérés, l'ensemble des plus courts chemins est donné par un parcours du graphe. Ce parcours part du noeud initial N_0, et ne choisit dans ses fils que les noeuds N_i de pondération minimale. Il procède récursivement de la même façon sur chacun des noeuds N_i choisis.

La complexité de l'algorithme est en θ(Nombre de noeuds + Nombre d'arcs), car il parcourt chaque arc et chaque noeud une fois et une seule pour l'assignation des coûts des noeuds, et en θ(Nombre de noeuds + nombre d'arcs) pour le parcours topologique permettant de supprimer les noeuds ne faisant partie d'aucun chemin optimal. Comme on peut borner le nombre de fils de chaque noeud par le nombre de tâches n, la complexité de l'algorithme est en $\theta((n+1)\times$Nombre de noeuds).

Sur la figure 6 nous donnons un exemple d'application de l'algorithme de pondération sur le système de tâches $S=\{\tau_1 <0,3,7,7>, \tau_2 <0,4,7,7>\}$ pour lequel on veut donner la priorité à la tâche 1. Il y a 35 séquences d'ordonnancement possibles, sans contraintes de

successeur, du système S, mais une seule est optimale pour minimiser les dates d'exécutions de τ_1.

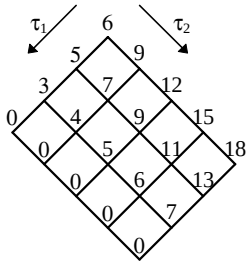

Figure 6 :Pondération d'un graphe d'ordonnancement

Le parcours du graphe ne laisse qu'une seule séquence d'ordonnancement $\tau_1,\tau_1,\tau_1,\tau_2,\tau_2\ \tau_2,\tau_2$.

Nous donnons une brève description des autres algorithmes de choix de séquence, qui sont de même complexité :

- Minimisation du temps moyen de réponse : le coût d'un arc allant d'une hauteur h-1 à h est nul sauf quand il est étiqueté par la dernière transition d'une tâche τ_i de E, dans ce dernier cas il coûte $TR = (h - \left\lfloor \frac{h}{T_i} \right\rfloor \times T_i)$ où $\lfloor\ \rfloor$ est la partie entière inférieure. Le coût d'un noeud N est donné par $coût(N)=min_{Ni\in Fils(N)}(coût(N_i)+ coût\_arc(N,N_i))$.
- Maximisation du temps moyen de retard admissible : le coût d'un arc allant d'une hauteur h-1 à h est nul sauf quand il est étiqueté par la dernière transition d'une tâche τ_i de E, dans ce dernier cas il coûte $TRA=R_i-TR$. Le coût d'un noeud N est donné par $coût(N)=max_{Ni\in Fils(N)}(coût(N_i)+ coût\_arc(N,N_i))$.
- Minimisation du taux moyen de réaction : dans ce cas, les coûts sont des réels, et le coût d'un arc allant d'une hauteur h-1 à h est nul sauf quand il est étiqueté par la dernière transition d'une tâche τ_i de E, dans ce dernier cas il coûte $\frac{TR}{R_i}$. Le coût d'un noeud N est donné par $coût(N)=min_{Ni\in Fils(N)}(coût(N_i)+ coût\_arc(N,N_i))$.
- Minimisation du temps maximal de réponse : Identique que pour sa version en moyenne, mais le coût d'un noeud N est donné par $coût(N)=min_{Ni\in Fils(N)}(max(coût(N_i), coût\_arc(N,N_i)))$.
- Maximisation du temps minimal de retard admissible : Idem mais le coût de N est donné par $coût(N)=max_{Ni\in Fils(N)}(min(coût(N_i), coût\_arc(N,N_i)))$.
- Minimisation du taux maximal de réaction : Idem mais le coût d'un noeud N est donné par $coût(N)=min_{Ni\in Fils(N)}(max(coût(N_i), coût\_arc(N,N_i)))$.

6. Conclusion

Nous avons proposé un algorithme NP d'ordonnancement optimal d'un système de tâches temps–réel à partir d'un modèle basé sur les réseaux de Petri. Dans une première phase, le graphe d'accessibilité est construit en temps et en espace NP, en utilisant des techniques permettant de limiter sa taille ainsi que le *backtracking* inhérent à sa construction. Puis nous avons montré qu'un algorithme du type recherche de plus court chemin pouvait être mis en place sur ce graphe pour extraire les séquences optimales au vu de certains critères en un temps linéaire du nombre de noeuds du graphe d'accessibilité.

Il en résulte que l'ordonnancement optimal au vu de certains critères de systèmes de tâches à l'aide du modèle basé sur les réseaux de Petri décrit dans [Choquet_Geniet–Geniet–Cottet96] a une complexité NP. La structure de graphe, réduit, et construit en utilisant une table de prévision diminue de manière exponentielle la complexité en temps atteinte par les approches utilisant la logique de l'ordonnancement qui eux, développent une approche arborescente des ordonnancements.

Les critères que nous avons choisi d'optimiser sont la minimisation du temps de réponse, la minimisation du taux de réaction, la maximisation de la latence en moyenne ou au pire d'un ensemble de tâches, et l'importance d'un ensemble de tâches par rapport aux autres.

Bibliographie

A. Choquet–Geniet, D. Geniet, F. Cottet, « Exhaustive Computation of the Scheduled Task Execution Sequences of a Real–Time Application », actes du 4th International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, Uppsala, Suède, 11–13/09/96.

A. Choquet-Geniet, G. Vidal-Naquet, « Réseaux de Petri et systèmes Parallèles », Editions Armand Colin, 1993.

F. Cottet, J.P. Babau, « Off-Line Temporal Analysis of Hard Real-Time Applications », second IEEE Workshop on Real-time Applications, Washington DC 1994.

J.Y.T. Leung, M.L. Merill, « A Note on Preemptive Scheduling of Periodic Real–Time Tasks », Information Processing Letters 11 (3), pp 115–118, 1980.

C.L. Liu, J.W. Layland, « Scheduling Algorithms for Multiprogramming in a Hard Real–Time Environment », journal of the ACM, 20 (1), pp 46–61, 1973.

P. Martineau, M. Silly, « Evaluation d'Algorithmes d'Ordonnancement Temps réel en Présence de ressources Partagées », Rapport interne LAN 94,4 Ecole Centrale de Nantes, 1994.

D.T. Peng, K.G. Shin, « A New Performance Measure for Scheduling Independant Real-Time Tasks », Journal of Parallel and distributed Computing 19, pp 11-26, 1993.

P. Pushner, C. Koza, « Calculating the Maximum Execution Time of Real-Time Programs », Jounal Real-Time Systems (1), 1989, pp 159-176.

L. Sha, R. Rajkumar, J.P. Lehoczky, « Priority Inheritance Protocols : An Approach to Real–Time Synchronisation », IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175–1185, September 1990.

J.A. Stankovic, « Misconception about Real–Time Computing », IEEE Computer Magazine, 21 (10), 1pp 0–19, 1988.