Ordonnancement temps réel dur multiprocesseur tolérant aux fautes appliqué à la robotique mobile

Mohamed Marouf

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 :
Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Mohamed MAROUF

le 01 juin 2012

Ordonnancement temps réel dur multiprocesseur tolérant aux fautes appliqué à la robotique mobile

~ ~ ~

Fault-tolerant multiprocessor hard real-time scheduling for mobile robotics

Directeur de thèse : **Yves SOREL**
Co-encadrement de la thèse : **Brigitte D'ANDREA-NOVEL**

Jury

Mme. Maryline CHETTO, Professeur,Université de Nantes	Rapporteur
M. Pascal RICHARD, Professeur, ENSMA	Rapporteur
Mme. Brigitte D'ANDREA-NOVEL, Professeur, MINES ParisTech	Examinateur
M. Fawzi NASHASHIBI, Directeur de recherche, INRIA Rocquencourt	Examinateur
M. Patrick RIVES, Directeur de recherche, INRIA Sophia Antipolis	Examinateur
M. Yves SOREL, Directeur de recherche, INRIA Rocquencourt	Examinateur

T H È S E

MINES ParisTech
Centre de robotique (CAOR)
60, boulevard Saint-Michel 75272 Paris cedex 06

Remerciements

Je tiens à remercier en premier lieu mon directeur de thèse Yves Sorel pour son encadrement excellent. Il a toujours été attentif et disponible malgré ses nombreuses charges, et il a su m'apprendre, grâce à ses compétences scientifiques, la rigueur et l'autonomie qui resteront pour moi des moteurs de mon travail de chercheur.

Je tiens aussi à remercier Maryline Chetto et Pascal Richard qui ont eu la lourde tâche d'être rapporteurs de ma thèse et qui m'ont fait des commentaires constructifs ayant permis l'amélioration de mon manuscrit. De même, je remercie Brigitte d'Andréa-Novel, Fawzi Nashashibi et Patrick Rives pour avoir accepté de faire partie de mon jury de thèse.

J'exprime aussi mes vifs remerciements à Michel Parent pour m'avoir accueilli comme stagiaire au sein de son équipe et ensuite m'avoir donné l'opportunité d'effectuer cette thèse. Je remercie également Fawzi Nashashibi qui a toujours été à mon écoute et a veillé sur le bon déroulement de ma thèse. Je remercie aussi Brigitte d'Andréa-Novel pour sa rigueur et ses orientations pertinentes.

Je tiens également à remercier Laurent George avec qui j'ai eu le plaisir de travailler sur certains aspects de ma thèse.

Je remercie aussi tous mes collègues des équipes projets AOSTE et IMARA pour les bons moments passés ensemble à l'INRIA-Rocquencourt. De même, je remercie tous mes amis qui m'ont soutenu tout au long de ma thèse.

À mes parents, à mes frères et sœurs,
à mon églantine.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Contexte

Les applications de robotique mobile sont de plus en plus complexes car d'une part elles font intervenir des algorithmes de traitement du signal et des images ainsi que des algorithmes de commande tous les deux multipériodes, et d'autre part elles sont soumises à des contraintes temps réel dures. Ces applications doivent, pour des raisons de modularité, de puissance de calcul et de tolérance aux fautes, être implantées sur des architectures distribuées hétérogènes formées de plusieurs processeurs et média de communication, tous les deux éventuellement de type différents. Il existe des solutions théoriques et un outil logiciel SynDEx pour résoudre ce problème de conception et d'implantation dans le cas monopériode sur une architecture distribuée. Cependant le code généré par SynDEx monopériode tolérant aux fautes, n'a jamais été testé sur une véritable architecture distribuée. Par ailleurs cette thèse a été réalisée dans le cadre d'une collaboration entre les équipes-projets AOSTE et IMARA, cette dernière ayant développé une architecture matérielle distribuée à base de microcontrôleurs MPC555 et de bus de communication CAN pour leur véhicule électrique appelé CyCab. Une application de suivi de CyCab a été développée et testée avec une version de SynDEx multipériode non tolérante aux fautes.

Objectif

Il existait deux versions de SynDEx : l'une monopériode tolérante aux fautes mais qui n'avait jamais été testée sur une véritable architecture, et l'autre multipériode non tolérante aux fautes qui avait été testée sur le CyCab, nous avons choisi de fonder nos travaux sur la version multipériode non tolérante aux fautes pour la rendre tolérante aux fautes. Cependant SynDEx multipériode ordonnance les tâches non préemptives à périodes strictes selon une condition d'ordonnançabilité très restrictive, i.e. elle donne un faible taux de succès d'ordonnancement. Donc le premier objectif visé a été de proposer des nouvelles conditions d'ordonnançabilité donnant de meilleurs taux de succès. Le deuxième objectif a été d'étudier séparément les tâches harmoniques ayant toutes des périodes multiples les unes

des autres, et les tâches non harmoniques. En effet les tâches harmoniques ont de meilleurs taux de succès d'ordonnancement que les tâches non harmoniques. Le troisième objectif a été de garder certaines tâches non préemptives périodiques strictes et d'y ajouter des tâches préemptives périodiques non strictes afin d'améliorer le taux de succès d'ordonnancement. Le quatrième objectif a été d'étendre les résultats d'ordonnancement pour rendre ces systèmes tolérants aux fautes matérielles des processeurs et des média de communication. L'objectif final consiste à intégrer tous ces travaux dans le logiciel SynDEx et de les tester sur une sur une application de contrôle/commande robotique fonctionnant sur le CyCab.

Plan général de la thèse

Ce manuscrit est constitué de quatre parties chacune d'elles comportant deux chapitres.

La première partie est consacrée à l'état de l'art. Nous proposons dans le premier chapitre un état de l'art sur l'ordonnancement temps réel embarqué et dans le deuxième chapitre un état de l'art sur la tolérance aux fautes.

La deuxième partie est consacrée à l'ordonnancement temps réel monopériode. Nous proposons dans le troisième chapitre une analyse d'ordonnançabilité de tâches non-préemptives périodiques strictes dans le cas des tâches harmoniques et des tâches non harmoniques. Cette étude est ensuite étendue dans le quatrième chapitre pour prendre en compte des mélanges de tâches non préemptives périodiques strictes et de tâches préemptives périodiques strictes.

La troisième partie est consacrée à l'ordonnancement multiprocesseur. Comme nous utilisons une approche d'ordonnancement partitionnée qui transforme le problème d'ordonnancement multiprocesseur en plusieurs problèmes d'ordonnancement monoprocesseur, nous exploitons, dans le cinquième chapitre les résultats d'ordonnancement monoprocesseur obtenus dans la partie précédentes pour proposer des algorithmes d'ordonnancement multiprocesseurs. Ces résultats sont ensuite étendus dans le septième chapitre pour tolérer les fautes de processeurs et de média de communications.

La dernière partie est consacrée aux développements logiciels et à l'application de suivi automatique de véhicules électriques CyCabs. Nous présentons dans le septième chapitre les améliorations apportées au logiciel SynDEx et dans le huitième chapitre l'applications de suivi automatique de CyCabs tolérante aux fautes.

Il y a aussi trois annexes : la première donne les caractéristiques du CyCab en particulier celles des processeurs et du bus de communication qu'il utilise, la deuxième présente le code source généré par SynDEx pour les microntrôleurs dsPICs, et la dernière donne les programme sources tolérants aux fautes.

Première partie

État de l’art

Chapitre 1

Ordonnancement des systèmes temps réel embarqués

1.1 Systèmes temps réel embarqués

Les applications qui nous intéressent dans cette thèse sont temps réel et aussi embarquées. Pour celà, nous devons prendre en considération les caractéristiques de ces systèmes dans les différents travaux que nous menons.

1.1.1 Caractéristiques

Un système embarqué ("embedded system" en anglais) est un système intégré dans un système plus large avec lequel il est interfacé, et pour lequel il réalise des fonctions particulières (contrôle/commande de procédés, surveillance, communication) spécifiant l'application concernée. Beaucoup de systèmes embarqués sont temps réel car d'une part ils doivent interagir infiniment avec leur environnement (le monde physique), et d'autre part ils doivent prendre des décisions et/ou effectuer des calculs en respectant des contraintes temporelles.

1.1.2 Architecture des systèmes embarqués

Un système embarqué comporte une partie matérielle formée d'un ensemble d'éléments physiques : processeur(s), mémoire(s) et entrées/sorties, une partie logicielle qui consiste en des programmes, et une source d'énergie. Les systèmes embarqués nécessitent parfois d'utiliser plusieurs processeurs pouvant être de types différents. Un premier classement des architectures pour système embarqué dépend du nombre de processeurs : architecture monoprocesseur ou multiprocesseur. Il existe différents classements pour les architectures multiprocesseurs :

- **homogène/hétérogène** selon la *nature des processeurs* de l'architecture :
  - identiques : les processeurs sont interchangeables et ont la même puissance de calcul ;
  - uniformes : chaque processeur est caractérisé par sa puissance de calcul avec l'interprétation suivante : lorsqu'un travail s'exécute sur un processeur de puissance de calcul s pendant t unités de temps, il réalise $s \times t$ unités de travail ;
  - indépendants : un taux d'exécution $r_{i,j}$ est associé à chaque couple travail-processeur (J_i, P_j), avec l'interprétation suivante : le travail ; le travail J_i réalise $(r_{i,j} \times t)$ unités de travail lorsqu'il s'exécute sur le processeur P_j pendant t unités de temps. Dès lors la vitesse de progression sur un même processeur varie éventuellement d'un travail à l'autre ;
- **homogène/hétérogène** selon la *nature des communications* entre processeurs :
  - homogène : les coûts de communication entre chaque paire de processeurs de l'architecture sont toujours les mêmes ;
  - hétérogène : les coût de communication entre processeurs varient d'une paire de processeurs à une autre ;
- **parallèle/distribuée** selon le type de *mémoire* de l'architecture :
  - parallèle : les processeurs communiquent par mémoire partagée ;
  - distribuée : les processeurs ne partagent pas de mémoire et ont leur propre mémoire qui est ainsi dite distribuée. Ils communiquent par envoi/réception de messages.

1.1.3 contraintes des systèmes temps réel embarqués

Nous distinguons deux types de contraintes : les contraintes temporelles et les contraintes matérielles.

1.1.3.1 contraintes temporelles

Les systèmes temps réel peuvent avoir deux types de contraintes temporelles : *contraintes strictes (dures)* et *contraintes souples*. Un système temps réel est dit à contraintes *strictes* quand une faute temporelle (non respect d'une échéance, arrivée d'un message après les délais, irrégularité d'une période d'échantillonnage, dispersion temporelle trop grande dans un lot de mesures simultanées) peut avoir des conséquences catastrophiques du point de vue humain, économique ou écologique. Un système temps réel est à *contraintes souples* lorsque le non respect de contraintes temporelles est acceptable. On admet alors de ne pas respecter certain pourcentage d'échéances par exemple. On parle alors de qualité de service (QoS).

Dans les applications de contrôle/commande temps réel critiques, les traitements de capteurs/actionneurs et les traitements de commande de procédés ne doivent pas avoir de gigue sur les entrées issues des capteurs et sur les sorties fournies aux actionneurs. Dans un tel système, une faute temporelle peut avoir des conséquences catastrophiques autant ou plus qu'une faute de calcul.

1.1.3.2 contraintes matérielles

Ces contraintes sont dues aux restrictions de ressources dans les systèmes embarqués. La puissance de calcul et la mémoire de ces systèmes sont limitées. Cela est dû aux limitations de poids, de volume (avions), de consommation d'énergie (véhicules), ou de prix. Parmi ces différentes contraintes, la gestion de la mémoire et la consommations sont les principales contraintes étudiées dans la littérature.

L'accès aux mémoires reste toujours plus lent comparé à la vitesse des processeurs qui ne cesse de croître. Bien que la capacité mémoire dans les systèmes embarqués augmente, elle reste toujours insuffisante pour des applications qui deviennent de plus en plus complexes. Les premières techniques de gestion de la mémoire étaient manuelles comme la réservation de mémoire (instruction "malloc" en langage C) et comme la restitution de mémoire (instruction "free" en langage C). Des techniques automatiques où l'intervention de l'utilisateur est (quasi) nulle ont été introduites plus tard. Ces techniques restent plus complexes à mettre en œuvre.

La complexité des systèmes embarqués ne cesse d'augmenter, ce qui engendre des consommations d'énergie de plus en plus élevées. La consommation d'énergie est un paramètre essentiel dans le développement des applications embarquées. L'utilisation de processeurs puissants cadencés à des fréquences élevées génère des consommations d'énergie élevées [Koc00]. Une technique importante de minimisation de la consommation d'énergie est basée sur la minimisation du "makespan" qui correspond au temps total d'exécution des tâches [LL08], afin d'utiliser des processeurs moins puissants pour avoir les mêmes résultats obtenus avec des processeurs puissants et qui consomment plus d'énergie [Ven06, Par02, Gar05, Koc00].

1.2 Ordonnancement de tâches temps réel

Dans la suite on présente les modèles de tâches qui nous intéressent et les contraintes temporelles qu'on peut y appliquer, puis on présente les différents algorithmes monoprocesseurs et multiprocesseurs capables d'ordonnancer ces tâches.

1.2.1 Tâches temps réel

Une tâche temps réel est une séquence d'instructions constituant l'unité de base d'un système temps réel. Les tâches réalisent des entrées/sorties et des calculs permettant de commander des procédés via un ensemble de capteurs et d'actionneurs, éventuellement tout ou rien, par exemple ensemble de tâches réalisant le contrôleur de vitesse d'une voiture ou le pilotage automatique d'un avion. Elles sont soumises à des contraintes temporelles qui doivent être respectées. Par exemple il faut désactiver avant un certain temps le contrôleur de vitesse d'une voiture lorsqu'on actionne les freins.

On suppose que les caractéristiques temporelles sont des entiers non négatifs multiples d'unité de temps en général égale à la période de l'horloge du processeur.

Les tâches *périodiques* se répètent indéfiniment et leurs instances sont séparées par une période constante. Une tâche périodique est activée par le déclenchement d'une interruption produite par un timer externe, à une date appelée date d'activation. Chaque instance s'exécute pendant une durée d'exécution maximale et doit terminer son exécution avant une échéance relative à sa date d'activation. Une tâche périodique est représentée par quatre paramètres : une date d'activation, un temps d'exécution maximum (pire cas), une échéance relative et une période.

Les tâches *apériodiques* doivent s'exécuter au moins une fois et ne se répètent pas nécessairement indéfiniment. Une tâche apériodique est représentée par trois paramètres : une date d'activation, un temps d'exécution maximum et une échéance relative à sa date d'activation. Différents procédés d'ordonnancement de tâches apériodiques d'un système temps réel sont exposés dans [SSL89].

Les tâches *sporadiques* sont un cas particulier de tâches apériodiques. Ce sont des tâches qui se répètent indéfiniment avec une période minimum. En d'autres termes, deux instances peuvent être séparées d'une valeur plus grande que cette période minimum. Une tâche sporadique est représentée par quatre paramètres : une date d'activation, un temps d'exécution maximum, une échéance relative à sa date d'activation et une durée minimum entre deux activations successives de la tâche.

1.2.1.1 Modèles de tâches

Comme par la suite nous nous intéresserons aux systèmes de tâches temps réel périodiques, nous introduisons d'abord le modèle classique habituellement cité dans la littérature et nous introduisons par la suite le modèle de tâches non préemptives périodiques strictes que nous utilisons dans nos travaux.

Modèle classique de tâches préemptives périodiques

Un modèle canonique de tâches temps réel préemptives périodiques a été proposé par Liu & Layland [LL73]. Ce modèle regroupe les principaux paramètres temporels qui permettent de caractériser une tâches temps réel et son ordonnancement. Ces paramètres sont scindés en deux groupes : paramètres statiques et paramètres dynamiques.

Afin d'alléger la lecture, les tâches préemptives périodiques (non strictes) sont notées tâches *PP* dans la suite du manuscrit.

Les paramètres de base d'une tâche $\tau_i(r_i^0, C_i, D_i, T_i)$ sont :

- r_i^0 (*first release* ou *offset*) : première date de réveil ou d'activation, qui représente le moment de déclenchement de la première requête d'exécution,
- C_i (*computing time*) : durée d'exécution maximale d'une tâche, souvent considéré comme le pire temps d'exécution WCET (Worst Case Execution Time), qui est une borne supérieure du temps d'exécution d'une tâche sur un processeur donné,
- D_i (*relative deadline*) : échéance relative, date au plus tard ou délai critique, ce paramètre représente le délai, relativement à la date d'activation, avant laquelle l'exécution de la tâche doit être terminée,
- T_i (*period*) : période d'exécution d'une tâche périodique ou écart minimum entre deux activations successives d'une tâche apériodique.

Notez que pour la k^{me} instance τ_i^k, la date d'activation (release time, request time, ready time) est donnée par $r_i^k = r_i^0 + k \cdot T_i$.

Lorsque la période d'une tâche est égale à son échéance, on dit que la tâche est à échéance sur requête.

D'autres paramètres sont dérivés des paramètres de base :

- $U_i = \frac{C_i}{T_i}$ (*utilization factor*) : facteur d'utilisation d'une tâche ; on a $U_i \leq 1$,
- $CH_i = \frac{C_i}{D_i}$ (*density*) : densité d'une tâche ; on a $CH_i \leq 1$.

Les paramètres dynamiques servent à suivre le comportement de l'exécution d'une tâche :

- s_i^k (*start time*) : date de début d'exécution de la k^{me} instance τ_i^k,
- e_i^k (*end time*) : date de la fin d'exécution de la k^{me} instance τ_i^k,
- d_i^k (*absolute deadline*) : échéance absolue de la k^{me} instance τ_i^k donnée par $d_i^k = r_i^k + D_i = r_i^0 + k \cdot T_i + D_i$,
- R_i^k (*response time*) : temps de réponse de la k^{me} instance τ_i^k donné par $e_i^k - r_i^k$. On a $C_i \leq R_i^k \leq D_i$,
- L_i (*laxity*) : laxité nominale d'une tâche qui représente le retard maximum pour son début d'exécution s_i^k lorsque cette tâche est exécutée seule.

La figure 1.1 représente un modèle canonique d'une tâche périodique où l'on peut voir deux instances dont la deuxième est préemptée deux fois.

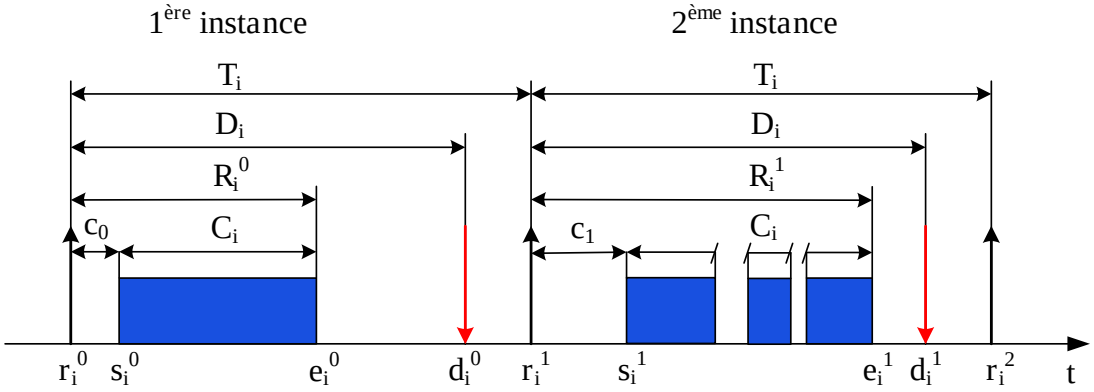


FIGURE 1.1 – Modèle classique d'une tâche PP $\tau_i(r_i^0, C_i, D_i, T_i)$

Modèle de tâches non préemptives périodiques strictes

Dans les applications de contrôle/commande temps réel critiques, les tâches réalisant les entrées/sorties ne doivent pas avoir de gigue. Donc pour éviter la gigue sur les entrées issues des capteurs les tâches les gérant doivent être périodiques strictes, et pour éviter la gigue sur les sorties fournies aux actionneurs les tâches les gérant doivent être non préemptives.

Le modèle de tâches à non préemptives périodiques strictes est le modèle naturel utilisé pour faire de l'ordonnancement sans préemption qui a existé bien avant celui préemptif dit de Liu & Layland décrit précédemment. Dans ce dernier la différence entre la date d'activation et la date de début d'exécution peut varier alors qu'elle est constante dans le cas de tâches non préemptives périodiques strictes. Ce qui revient à dire qu'il n'y a pas de gigue [CS03]. De plus il a le gros avantage d'être déterministe car il ne dépend pas de *l'approximation du coût de l'OS* définie par Liu & Layland [LL73], nécessaire pour gérer la préemption. Cet avantage est utile dans le cas des applications temps réel critiques citées plus haut.

Afin d'alléger la lecture, les tâches non préemptives périodiques strictes sont notées tâches *NPPS* dans la suite du manuscrit.

Ce modèle a les caractéristiques suivantes :

- la différence entre la date d'activation et la date de début d'exécution d'une instance τ_i^k est égale à une constante : $s_i^k - r_i^k = d$,
- le temps de réponse est constant : $R_i = C_i + d$,

– la différence entre deux dates de début d'exécution successives est égale à la période : $s_i^{k+1} - s_i^k = T_i$,
– l'échéance est égale à la période : $T_i = D_i$,

La figure 1.2 représente un modèle de tâches NPPS dans le cas où $d = 0$. Dans la suite du manuscrit et sans perte de généralité nous ferons cette hypothèse.

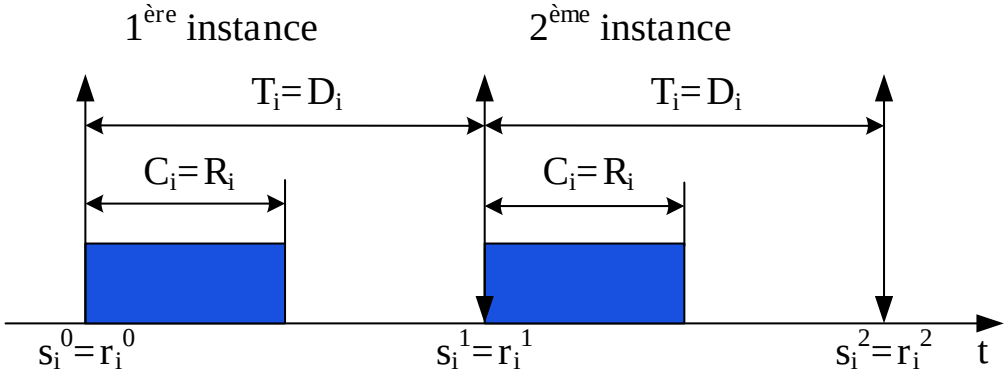


FIGURE 1.2 – Modèle d'une tâche NPPS $\tau_i(C_i, T_i)$

1.2.1.2 contraintes temps réel

Parmi les contraintes temporelles traitées dans cette thèse on retrouve les contraintes suivantes :

Echéance

La contrainte d'échéance exprime une condition sur la date de fin au plus tard de chacune des tâches du système. On distingue trois types d'échéances :

1. $D_i = T_i$: ce cas représente les tâches à échéance sur requête, où chaque activation d'une tâche doit être exécutée avant la prochaine requête de la même tâche.
2. $D_i \leq T_i$: chaque activation d'une tâche doit être exécutée avant une date inférieure ou égale à la prochaine date d'activation de la même tâche.
3. $D_i \leq T_i$ ou $D_i \geq T_i$: la date de fin d'exécution d'une tâche peut être inférieure, égale ou supérieure à sa période.

Précédence et dépendance

On parle de précédence et de dépendance quand il y a une interaction entre les tâches. Une précédence désigne un ordre partiel d'exécution des tâches [MBFSV07,

Cuc04] sans qu'il y ait transfert de données entre ces tâches. Deux tâches sont dépendantes si une tâche produit des données et l'autre tâche les consomme. La tâche réceptrice, appelée successeur, ne peut s'exécuter avant la réception de données de la tâche émettrice appelée prédécesseur. Une dépendance impose une précédence entre la tâche qui produit une donnée et la tâche qui la consomme. Les tâches sont dites indépendantes lorsqu'elles ne sont définies que par leurs paramètres temporels.

Dans la suite du manuscrit quand on parle d'ordonnancement on sous-entend ordonnancement temps réel.

Gigue

L'exécution de requêtes périodiques est parfois soumise à des contraintes de régularité pour le début de leurs exécutions ou pour leurs temps de réponse. La variation entre le début d'exécution de deux instances successives est appelée gigue de début d'exécution (gigue sur les entrées). La variation entre le temps de réponse de deux instances successives est appelée gigue de temps de réponse (gigue sur les sorties).

1.2.2 Typologie des algorithmes d'ordonnancement

Étant donné un algorithme d'ordonnancement, une *condition d'ordonnançabilité* permet de déterminer s'il existe un ordonnancement qui satisfait toutes les contraintes des tâches. En revanche une *condition de faisabilité* permet de déterminer s'il existe un ordonnancement qui satisfait toutes les contraintes des tâches. Donc un ensemble de tâches est faisable s'il existe une condition de faisabilité qui est satisfaite par cet ensemble.

Un algorithme d'ordonnancement est dit *hors ligne* s'il construit la séquence d'ordonnancement complète à partir des paramètres temporels de l'ensemble des tâches avant l'exécution de l'application. Cela convient bien aux systèmes de tâches périodiques. Il est dit *en ligne* s'il construit la séquence d'ordonnancement à partir des paramètres temporels de l'ensemble des tâches pendant l'exécution de l'application. Les algorithmes en ligne sont plus robustes vis-à-vis des dépassements des WCETs. Cela convient bien aux systèmes de tâches sporadiques et apériodiques.

Un algorithme d'ordonnancement est dit *optimal* si tout ensemble de tâches faisable, est ordonnançable avec cet algorithme. En d'autres termes un algorithme d'ordonnancement est optimal si lorsqu'un ensemble de tâches est ordonnançable alors l'algorithme optimal conclut qu'il est ordonnançable.

Un algorithme d'ordonnancement est dit *monoprocesseur* s'il n'ordonnance l'ensemble des tâches que sur un seul processeur. Il est dit *multiprocesseur* s'il

ordonnance l'ensemble des tâches sur plusieurs processeurs.

Un algorithme d'ordonnancement est dit *préemptif* s'il exploite la préemption des tâches, i.e. s'il permet qu'une tâche s'exécutant soit interrompue par une autre tâche plus prioritaire. Il est dit *non préemptif* s'il n'utilise pas la préemption des tâches.

1.2.3 Algorithmes d'ordonnancement monoprocesseur

Nous présentons dans cette partie les différents types d'algorithmes d'ordonnancement monoprocesseur que l'on trouve dans la littérature. Nous commençons d'abord par présenter les algorithmes d'ordonnancement pour des tâches PP, puis nous nous focaliserons sur les algorithmes d'ordonnancement des tâches NPPS.

1.2.3.1 Préemptifs périodiques , apériodiques et sporadiques

Dans cette catégorie d'algorithmes d'ordonnancement, nous distinguons les algorithmes à priorité statique (fixe) et ceux à priorité dynamique (variable).

Priorité statique (fixe)

Pour ce type d'algorithmes, la priorité d'une tâche est fixée avant son exécution et elle est la même pour toutes ses activations. Les algorithmes RM et DM décrits ci-dessous sont de ce type.

RM (Rate Monotonic)

L'algorithme à priorité fixe Rate Monotonic a été introduit par liu et layland en 1973 [LL73]. Il s'agit d'un ordonnancement préemptif à priorité statique qui s'applique à un ensemble de tâches périodiques indépendantes, et à échéance sur requête. Les priorités sont inversement proportionnelles aux périodes des tâches, i.e. la tâche la plus prioritaire est celle ayant la plus petite période.

Propriété 1.1 *L'algorithme* RM *est optimal pour l'ordonnancement de tâches préemptives, périodiques, indépendantes, à premières dates d'activation simultanés et à échéance sur requête.*

Remarque 1.1 *On appelle souvent* scénario synchrone *le cas particulier où les première dates d'activation des tâches sont simultanées. Ces tâches sont dites* concrètes*. Les tâches qui n'ont pas cette particularité sont dites* non concrètes*.*

Pour les tâches apériodiques et sporadiques, l'algorithme RM n'est plus optimal.

La condition suffisante d'ordonnançabilité avec l'algorithme RM pour un système de n tâches est donnée par [LL73] :

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

Joseph et Pandya ont proposé par la suite une condition de faisabilité nécessaire et suffisante basée sur le calcul du temps de réponse $RT(\tau_i)$ (Response Time) [Jos85]. Elle montre que si un ensemble de tâches périodiques triées par priorités décroissantes est ordonnancé par RM et dont le temps de réponse $RT(\tau_i)$ d'une tâche τ_i est borné supérieurement par la solution de l'équation

$$RT(\tau_i)^{q+1} = \sum_{j=1}^{i-1} \lceil \frac{RT(\tau_i)^q}{T_j} \rceil . C_j + C_i$$

alors l'ordonnancement de cet ensemble est faisable si et seulement si : $\forall i = 1..n,\ RT(\tau_i) \leq T_i$. Cette équation peut être résolue récursivement où $\lceil x \rceil$ est la partie entière supérieure de x.

DM (Deadline Monotonic)

Cet algorithme a été introduit par Leung et Whitehead en 1982 [LW82]. Contrairement à l'algorithmes RM qui est basé sur les périodes des tâches, DM est basé les échéances relatives des tâches. Les priorités des tâches sont inversement proportionnelles aux échéances relatives des tâches où $T(\tau_i) \geq D_i \geq C_i$. La tâche la plus prioritaire est celle ayant la plus petite échéance relative.

Propriété 1.2 *L'algorithme* DM *est optimal pour l'ordonnancement de tâches préemptives, périodiques, indépendantes, à premières dates d'activation simultanés et à échéances contraintes* $D_i \leq T_i$.

Pour les tâches apériodiques et sporadiques, l'algorithme DM n'est plus optimal. La condition suffisante d'ordonnançabilité de n tâches périodiques (triées par priorités décroissantes) $\{\tau_1, ..., \tau_i, ..., \tau_n\}$ avec DM basée sur la densité est :

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

De la même manière que pour RM, il existe une condition nécessaire et suffisante pour l'algorithme DM. Si un ensemble de tâches périodiques triées par priorités décroissantes est ordonnancé par DM et dont le temps de réponse $RT(\tau_i)$ d'une tâche τ_i est borné supérieurement par la solution de

$$RT(\tau_i)^{q+1} = (\sum_{j=1}^{i-1} \lceil \frac{RT(\tau_i)^q}{T_j} \rceil . C_j) + C_i$$

alors l'ordonnancement de cet ensemble est faisable si et seulement si :
$\forall i_{(1 \leq i \leq n)}\ RT(\tau_i) \leq D_i$.

À noter que si toutes les tâches sont à échéance sur requête, RM et DM sont équivalents.

Priorité dynamique (variable)

Pour cette catégorie d'algorithmes d'ordonnancement, la priorité d'une tâche est définie pour chaque activation de cette tâche.

EDF (Earliest Deadline First)

Cet algorithme a été présenté par Jackson en 1955 [Jac55] puis par Liu et Layland en 1973 [LL73]. C'est un ordonnancement qui peut être préemptif ou non préemptif et est à priorité dynamique. Il s'applique à des tâches périodiques indépendantes et à échéance sur requête. Le principe d'allocation de priorités dans cet algorithme est que la tâche dont l'échéance absolue arrive le plus tôt aura la priorité la plus élevée. Les priorités sont réévaluées à chaque fois que l'ordonnanceur doit déterminer la prochaine tâche à exécuter. Par exemple lorsqu'une nouvelle tâche est activée et que son échéance est la plus proche comparée à celles des autres tâches prêtes à être exécutées.

Propriété 1.3 *L'algorithme* EDF *est optimal pour l'ordonnancement de tâches préemptives et indépendantes [Der74].*

La condition nécessaire et suffisante d'ordonnançabilité dans le cas préemptif $\forall i : D_i = T_i$ (échéance sur requête) est :

$$\sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq 1$$

LLF (Least-Laxity First)

Cet algorithme se base sur la laxité. LLF a été introduit par Mok et Dertouzos [Mok83b, MD78]. À chaque invocation, LLF élit la tâche dont la laxité est la plus faible. L'ouvrage [CDKM00] montre que les conditions d'ordonnançabilité pour l'algorithme LLF sont les mêmes que pour EDF, c'est-à-dire que la condition

nécessaire et suffisante d'ordonnançabilité dans le cas préemptif si $\forall i : D_i = T_i$ est :

$$\sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq 1$$

Propriété 1.4 *L'algorithme* LLF *est optimal pour l'ordonnancement de tâches préemptives, indépendantes et avec des échéances relatives inférieures ou égales aux périodes [DM89, Mok83a].*

L'algorithme LLF présente l'inconvénient, lorsque plusieurs tâches possèdent la même laxité, d'engendrer un grand nombre de préemptions donc de changements de contexte ce qui explique qu'il soit aussi peu utilisé dans le cas monoprocesseur [Bim07].

1.2.3.2 Non préemptifs périodiques stricts

Les problèmes d'ordonnancement non préemptifs ont reçu moins d'attention dans le domaine de l'ordonnancement temps réel. En effet, comme vu précédemment il existe beaucoup de conditions d'ordonnançabilité dans le cas préemptif. Malheureusement elles deviennent, dans le meilleur des cas, des conditions nécessaires (non suffisantes) [GRS96] dans le cas non préemptif.

Les problèmes d'ordonnancement monoprocesseur non préemptifs sont NP-difficiles au sens fort, comme Jeffay, Stanat et Martel l'ont démontré [JSM91]. Baruah a proposé dans [Bar98] un modèle de tâches récurrentes, qui est plus général que le modèle de tâches périodiques et non périodiques. Baruah et Chakraborty ont proposé dans [BC06] une condition d'ordonnançabilité nécessaire et suffisante de ce type de tâches, et ont montré qu'il existe des algorithmes en temps polynomial pour les ordonnancements préemptifs et non préemptifs.

Une analyse d'ordonnançabilité des tâches non préemptives périodiques non strictes a été effectuée par George, Rivierre, and Spuri [GRS96]. Ils ont prouvé dans le scénario synchrone que pour n'importe quel ensemble de tâches non préemptives $\{\tau_1, ..., \tau_i, ..., \tau_n\}$ tel que $\sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq 1$ est ordonnançable en utilisant EDF si et seulement si ρ est un instant donné dans le scénario synchrone de l'exécution tel que $\nexists i : D_i > \rho$ alors :

$$\forall \rho \in \mathbb{S}, \quad \rho \geq h(\rho)$$

Avec :
- $\mathbb{S} = \bigcup_{i=1}^{n} \{kT_i + D_i, k \in \mathbb{N}\} \cap [0, L[$,
- h la demande processeur qui représente la durée cumulée d'exécution des tâches qui arrivent et ont leurs l'échéances dans l'intervalle $[0, t[$,

– $\mathbb{L}$ la plus longue période d'activité du processeur obtenue dans le scénario synchrone (toutes les tâches démarrent en 0).

La principale différence entre les travaux précédents et ceux que nous proposons dans cette thèse réside dans la définition des périodes qui pour nous sont strictes.

Nous rappelons que dans le cas d'une période stricte la différence entre deux dates d'activation successives est constante et la différence entre deux dates de début d'exécution est aussi constante, et la différence entre une date d'activation et une date de début d'exécution est constante. En revanche dans le modèle classique de tâches la période est telle que la différence entre deux dates d'activation successives est constante, mais celle entre deux dates de début d'exécution peut varier.

Il existe des travaux sur l'ordonnancement monoprocesseur de tâches NPPS. Cucu et al. [CS03] ont étendu les travaux de George et al. [GRS96] dans deux directions : la première lorsque des tâches NPPS ont des précédences, et la deuxième lorsque plusieurs paires de tâches ont des contraintes de latence [CPS07]. Ces multiples contraintes de latence aussi appelées contraintes de bout-en-bout (end-to-end). Les deux types de contraintes peuvent être mélangées. Korst et al. ont proposé dans [KALW91] une condition d'ordonnançabilité nécessaire et suffisante par deux tâches NPPS. Cette condition devient une condition suffisante dans le cas de plus de deux tâches, comme l'avait prouvé Kermia et al. dans [KS08]. Cependant, comme nous l'avons mentionné dans [MS10], cette condition est très restrictive (pessimiste). Eisenbrand et al. ont proposé dans [EHN$^+$10b] un algorithme d'ordonnancement des tâches harmoniques et non harmoniques, sans donner de conditions d'ordonnançabilités.

Il existe des travaux sur l'ordonnancement multiprocesseur de tâches NPPS. Kermia et al. ont donné dans [KS07] des solutions approchées avec des heuristiques pour l'ordonnancement de tâches NPPS pour des architectures multiprocesseurs hétérogènes. Al Sheikh et al. ont donné dans [ASBH10] des solutions exactes avec la programmation linéaire en nombres entiers pour l'ordonnancement multiprocesseur de tâches NPPS pour des plates-formes IMA (Integrated Modular Avionics).

1.2.4 Algorithmes d'ordonnancement multiprocesseur

Nous avons à ordonnancer un ensemble Γ de n tâches périodiques (ou sporadiques) sur une architecture multiprocesseur composée de m processeurs. Ce problème a été formulé pour la première fois par Liu en 1969 [Liu69].

On distingue trois approches d'ordonnancement multiprocesseur : globale, partitionné et semi-partitionné.

1.2.4.1 Approche globale

Il s'agit d'appliquer sur l'ensemble des processeurs une stratégie d'ordonnancement globalement. Ceci revient à utiliser un seul ordonnanceur pour l'ensemble des processeurs. Si le nombre de tâches n est supérieur ou égal au nombre de processeurs, les tâches les plus prioritaires sont attribuées aux m processeurs, sinon des processeurs sont laissés oisifs. Dans cette approche, outre la préemption des tâches, la migration de ces dernières est aussi autorisée. Une tâche peut donc commencer son exécution sur un processeur P_i, puis être préemptée par une tâche plus prioritaire, pour reprendre son exécution sur un autre processeur P_j, avec $i \neq j$. Ce phénomène est appelé la migration de tâche et est une caractéristique des approches globales.

Hong et Leung ont démontré dans [HL92a] qu'il n'existe aucun algorithme d'ordonnancement en-ligne optimal pour des systèmes de tâches avec au moins deux échéances distinctes. Ce résultat n'est plus valable dans le cas où toutes les tâches ont une échéance commune [HL92b].

Phillips, Stein, Torng and Wein [PSTW97] ont étudié la manière dont un algorithme en ligne peut se comporter si l'on augmente la vitesse des processeurs. Ainsi un système de tâches qui est ordonnançable sur m processeurs identiques, est ordonnançable avec EDF sur m processeurs identiques mais ayant une vitesse $(2 - \frac{1}{m})$ fois plus grande que la vitesse initiale.

Contrairement aux algorithmes d'ordonnancement monoprocesseurs, les algorithmes multiprocesseurs présentent des anomalies. En effet certaines modifications des paramètres des tâches qui paraissent intuitivement positives, peuvent rendre non ordonnançables un système de tâches ordonnançables. Par exemple pour un système de tâches ordonnançables, en diminuant le facteur d'utilisation d'une tâche en augmentant sa période ou en diminuant sa durée d'exécution, on s'attendrait à avoir encore un système de tâches ordonnançables alors que parfois ce n'est pas vrai.

Par ailleurs un système de tâches périodiques ordonnançables en monoprocesseur reste ordonnançable si ces tâches sont maintenant sporadiques. Cette propriété n'est pas valable en multiprocesseur.

Parmi les algorithmes d'ordonnancement global, PFAIR est le plus connu. Baruah et al. ont établi dans [BCPV96] les fondements théoriques pour les algorithmes PFAIR et plus particulièrement la notion de "proportionate fairness". Ces algorithmes diffèrent des algorithmes classiques d'ordonnancement dans la mesure où le taux d'exécution d'une tâche est quasi constant. Lorsque l'on considère de grands intervalles de temps, le taux d'exécution d'une tâche sur un intervalle de temps est approximativement égal à son facteur d'utilisation $U(\tau_i)$. Cependant, sur de petits intervalles de temps, le taux d'exécution d'une tâche peut varier d'une façon très importante. Dans les algorithmes PFAIR, chaque tâche est exé-

cutée approximativement à un taux constant et ce en divisant la tâche en série de sous-tâches exécutées dans des intervalles identiques appelés fenêtres. Dans le cas particulier des architectures multiprocesseurs homogènes et des tâches périodiques à échéance sur requête, PFAIR est optimal. Il existe trois variantes de PFAIR : PF [BCPV96], PD [BGP95], PD^2 [And00]. La stratégie utilisée par ces algorithmes dans la manière où ils donnent des priorités aux sous-tâches est proche d'EDF.

À noter que certains algorithmes d'ordonnancement monoprocesseur comme EDF peuvent s'appliquer en ordonnancement multiprocesseur global [DP06], mais ils ne sont pas optimaux et peuvent se révéler très inefficaces (cf. effet de Dhall).

Baruah a étudié l'ordonnancement de tâches non préemptives périodiques non strictes et a proposé des conditions d'ordonnançabilité suffisantes avec l'algorithme d'ordonnancement multiprocesseur EDF_{np} selon l'approche globale [Bar06].

1.2.4.2 Approche partitionnée

Il s'agit d'appliquer sur chaque processeur une stratégie d'ordonnancement qui peut être différente des autres. Cela revient à utiliser un ordonnanceur par processeur. L'ensemble des n tâches est partitionné en m sous-ensembles $\Gamma_1, \Gamma_2, ..., \Gamma_m$, avec $\bigcup \Gamma_i = \Gamma$, et chaque sous-ensemble Γ_i est ordonnancé sur le processeur P_i. Dans cette approche, les tâches ne sont pas autorisées à migrer d'un processeur à un autre [DB10].

Le problème qui consiste à trouver un partitionnement optimal, est équivalent à un problème de remplissage de boîtes avec des objets de tailles différentes (bin-packing) [CJG$^+$98] en cherchant à placer le plus grand nombre d'objets possibles dans chaque boîte et en cherchant à minimiser le nombre de boîtes. Les tâches correspondent aux objets et les processeurs aux boîtes. Ce problème est NP-difficile au sens fort [GJ90]. Des algorithmes de complexité polynomiale, comme *First Fit*, *Next Fit*, *Best Fit* ou *Worst Fit* sont proposés pour le résoudre de façon approchée [BLOS95, OS95, SVC98]. Ils effectuent à la fois le choix des processeurs et la vérification d'une condition d'ordonnançabilité sur chaque processeur.

À noter que certains algorithmes d'ordonnancement monoprocesseur comme EDF peuvent s'appliquer en ordonnancement multiprocesseur partitionné [LDG01, LGDG00].

1.2.4.3 Approche semi-partitionnée

L'approche semi-partitionnée est dérivée de l'approche partitionnée. Dans cette approche certaines instances d'une tâche peuvent être exécutée sur des processeurs différents, ce qui entraine des migrations moins nombreuses que dans le cas global. Certaines tâches ne sont pas autorisées à migrer.

Andersson et Tovar ont introduit dans [AT06] une approche semi-partitionnée d'ordonnancement de tâches périodiques où certaines instances de tâches peuvent être exécutées sur des processeurs différents avec une utilisation bornée. Ensuite Anderson et al. ont proposé dans [ABB08] une approche où certaines instances de tâches sporadiques peuvent être exécutées sur des processeurs différents. Dans cette approche chaque processeur P_i exécute au plus les instances de deux tâches, l'une exécutée sur le processeur $P_i - 1$ et l'autre sur le processeur $P_i + 1$.

Kato et Yamasaki ont introduit plusieurs algorithmes Ehd2-SIP [KY07], EDDP [KY08a, KY08b] tous fondés sur l'algorithme à priorité dynamique EDF. Ils ont aussi proposé un algorithme DM-MP pour des tâches sporadiques fondé sur un algorithme à priorité fixe.

Lakshmanan et al. ont proposé l'algorithme PDMS-HPTS [LRL09] pour des tâches sporadiques fondé sur un algorithme à priorité fixe.

1.3 Graphes d'algorithme et d'architecture

Les systèmes distribués embarqués peuvent être décrits par deux graphes : un graphe d'algorithme représentant les fonctionnalités à implanter ainsi que leurs dépendances sous la forme d'un graphe et un graphe d'architecture matérielle représentant les processeurs inter-connectés qui implantent ces fonctionnalités. On présente donc dans ce qui suit le graphe d'algorithme et le graphe d'architecture matérielle.

1.3.1 Graphe d'algorithme

Dans la mesure où nous visons des systèmes distribués (parallèles, multi-cœur), nous avons choisi de modéliser l'algorithme par un graphe flot de données répété indéfiniment [GLS99, CKS02] bien adapté à la spécification de parallélisme potentiel pour les applications temps réel réactives. En effet l'algorithme d'un système réactif interagit de manière répétée indéfiniment avec son environnement. Une interaction est composée d'une acquisition, d'un calcul et d'une réaction (commande). Dans un graphe flot de données les sommets représentent les opérations et les arcs représentent les dépendances de données entre opérations.

Définition 1.1 *Une dépendance de données $(o_i \triangleright o_j)$ représente la relation de précédence et un échange de données entre l'opération o_i appelée prédécesseur de o_j, et l'opération o_j appelée successeur de o_i. Elle est appelée dépendance de donnée de diffusion si elle possède une seule opération productrice et plusieurs opérations consommatrices, et elle est notée $(o_i \triangleright ..., o_j, ...)$.*

Définition 1.2 *Une opération est une séquence finie de code.*

Les opérations dans un graphe d'algorithme peuvent être des opérations de calcul ou des opérations d'entrée/sortie (opération sans prédécesseur/opération sans successeur) qui peuvent être associées à des capteurs/actionneurs.

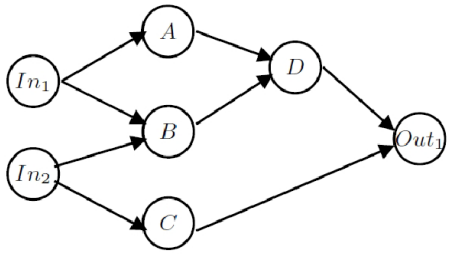


FIGURE 1.3 – Exemple d'un graphe d'algorithme.

La figure 1.3 représente un graphe d'algorithme composé de deux opérations d'entrées In_1 et In_2, de quatre opérations de calcul A, B, C, D , d'une opération de sortie Out_1, de six dépendances de données $(In_2 \triangleright B)$, $(In_2 \triangleright C)$, $(A \triangleright D)$, $(B \triangleright D)$, $(C \triangleright Out_1)$ et $(D \triangleright Out_1)$, et d'une dépendance de diffusion $(In_1 \triangleright \{A, B\})$.

La figure 1.4 montre le graphe flot de données (motif) de la figure 1.3 répété indéfiniment pour lui permettre d'interagir avec l'environnement (procédé) qu'il commande par l'intermédiaire des opérations d'entrée/sortie.

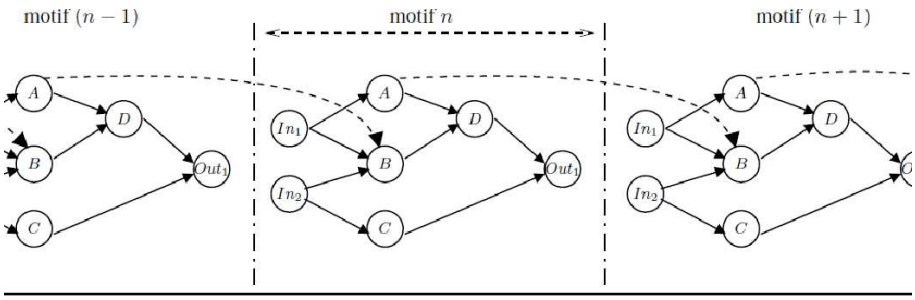


FIGURE 1.4 – Exemple d'un graphe d'algorithme infiniment répété.

Définition 1.3 *Une opération délai se charge de mémoriser la donnée produite par une opération lors de son exécution précédente à la répétition $n-1$. Cette donnée sera consommée par son successeur à la répétition n. Cette opération est nécessaire pour pouvoir factoriser le graphe flot de données répété indéfiniment afin d'obtenir un graphe factorisé.*

Définition 1.4 *Une opération constante conserve sa valeur durant l'exécution du système. Elle est exécutée une fois durant la première répétition et sa durée est considérée nulle.*

Définition 1.5 *Un algorithme est modélisé par un graphe flot de données répété indéfiniment factorisé, appelé graphe d'algorithme. Il est représenté par le couple* (O, E) *où* O *est l'ensemble des* n *opérations et* E *est l'ensemble des* m *dépendances de données.*

1.3.2 Graphe d'architecture

Une architecture distribuée peut être constituée de composants programmables (processeurs) et de composants non programmables (ASIC : Application Specific Integrated Circuit, FPGA : Field Programmable Gate Array), inter-connectés tous ensemble. Un processeur est formé d'une machine séquentielle exécutant des instructions et d'autant de machines séquentielles de communication que de connexions possibles avec d'autres processeurs. Chacune de ces machines séquentielles de communication exécute des communications inter-processeur. Nous faisons l'hypothèse que toutes les machines séquentielles d'un processeur communiquent par une mémoire partagée interne.

Ainsi le graphe d'architecture est constitué de trois types de sommets : processeur, mémoire partagée et multiplexeur-démultiplexeur, et d'arcs qui relient ces trois types de sommets. On ne peut pas connecter deux sommets du même type. Les communications inter-processeur peuvent s'effectuer soit à travers une mémoire partagée à laquelle accèdent les machines séquentielles de communication, soit par passage de messages à travers un multiplexeur-démultiplexeur auquel accèdent les machines séquentielles de communication qui possèdent chacune une partie de la mémoire distribuée [GS03]. Dans le cas de communication par passage de messages (mémoire distribuée) on distingue deux types de multiplexeurs-démultiplexeurs : point-a-point et multipoint (bus).

La figure 1.5a (resp. figure 1.5b) représente le graphe d'une architecture point-à-point (resp. à bus).

Remarque 1.2 *Sur les graphes d'architecture le sommet de type processeur représente la machine séquentielle exécutant des instructions. Les machines séquentielles exécutant les communications inter-processeurs sont représentées par les points d'accroche des arcs de connexion. Dans le cas d'une communication par passage de message chaque point d'accroche représente une machine séquentielle de communication et une mémoire partagée et dans le cas d'une communication par mémoire partagée chaque point d'accroche représente une machine séquentielle de communication seulement.*

Les opérations et les dépendances du graphe d'algorithme devront être distribuées et ordonnancées sur les processeurs et les machines séquentielles de communication de l'architecture matérielle. Pour faire cela les sommets du graphe

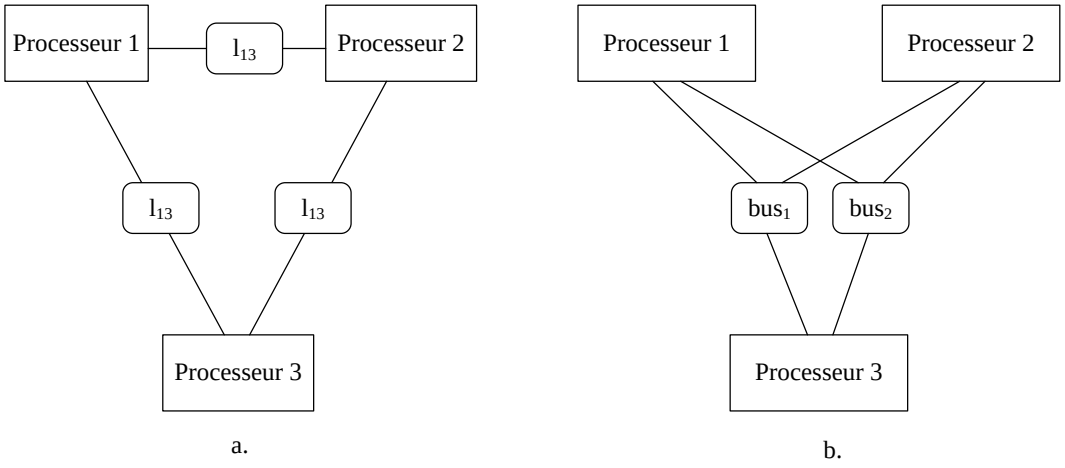


FIGURE 1.5 – Modèles d'architecture

d'algorithme doivent être étiquetés par des caractéristiques temporelles indépendantes de l'architecture comme la période, la période minimale, l'échéance, et des caractéristiques temporelles dépendantes de l'architecture comme le WCET, la quantité de mémoire, etc. De la même manière les dépendances de données doivent être étiquetées par des caractéristiques temporelles dépendantes de l'architecture WCCT, quantité de mémoire, etc. Les périodes des communications sont déduites de celles des opérations dépendantes. Quand les sommets et les dépendances sont étiquetés le graphe d'algorithme est un graphe de tâches temps réel.

1.4 Conclusion

Dans ce chapitre nous avons présenté un état de l'art sur les systèmes temps réel embarqués. Nous avons commencé par donner des définitions concernant les architectures des systèmes embarqués ainsi que les contraintes temporelles et matérielles, puis nous avons présenté le graphe de tâches temps réels classique et celui de tâches NPPS. Ensuite nous avons présenté les différentes conditions d'ordonnançabilité et les algorithmes d'ordonnancement associés, monoprocesseur et multiprocesseur. Enfin nous avons présenté le graphe d'algorithme correspondant à la spécification de systèmes de tâches temps réel et le graphe d'architecture distribuée que nous allons utiliser dans la suite.

Dans le chapitre suivant nous présentons un état de l'art sur la tolérance aux fautes dans les systèmes temps réel embarqués.

Chapitre 2

Tolérance aux fautes des systèmes temps réel embarqués

2.1 Introduction

Les systèmes réactifs réalisent des traitements complexes critiques et sont soumis à des contraintes strictes en terme de temps et de fiabilité. En effet, au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une faute, ces systèmes doivent être extrêmement sûrs. Pour cette raison, ils doivent mettre en oeuvre des mécanismes de tolérance aux fautes comme la redondance matérielle/logicielle [CP99, PIEP09, CSC05, IPEP05].

La majorité des méthodes existant dans la littérature pour la conception des systèmes tolérants aux fautes se concentrent sur les problèmes qui résultent de défaillances de matériel (processeurs et liens de communications). Ils supposent que la programmation (l'ensemble des traitements) est correcte et validée. Ces méthodes sont basées sur la réplication passive [QHPL00, DGLS01] ou active [Lap92] et sont vitales dans la conception de systèmes sûrs de fonctionnement appelés dans la suite *systèmes tolérants aux fautes*. La tolérance aux fautes permet à un système de continuer à délivrer un service conforme à sa spécification en présence de fautes.

2.2 Terminologies de la tolérance aux fautes

La défaillance d'un système est la conséquence d'une erreur qui elle-même est la conséquence d'une faute activée. Comme un système informatique est souvent composé de plusieurs sous-systèmes, la défaillance d'un sous-système peut créer et/ou activer une faute dans un autre sous-système, ou dans le système lui même. La relation entre ces trois termes fautes, erreur et défaillance relativement aux

sous-systèmes du système est représentée dans la figure 2.1 [Lap92].

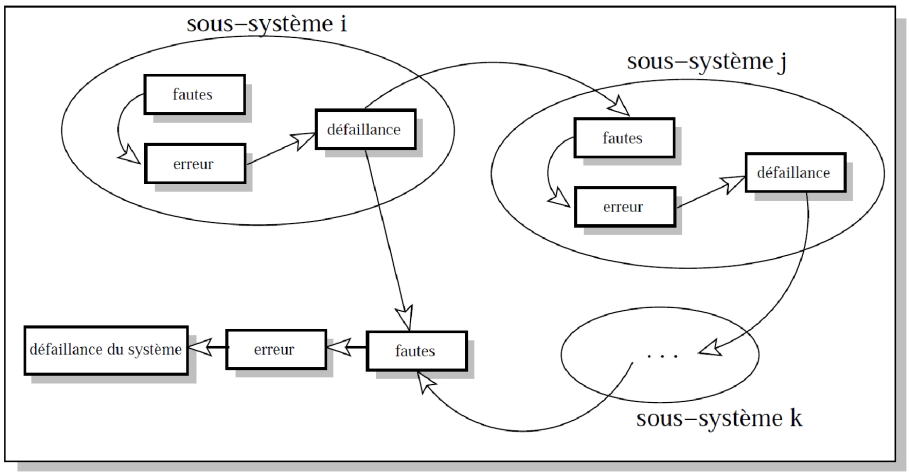


FIGURE 2.1 – Relation entre faute, erreur et défaillance.

2.2.1 Faute

Définition 2.1 Faute : *La faute dans un système informatique représente soit un défaut d'un composant matériel, ou soit un défaut d'un composant logiciel de ce système. Elle peut être créée de manière intentionnelle ou accidentelle, à cause des phénomènes physiques ou à cause des imperfections humaines. Durant l'exécution du système, la faute reste dormante jusqu'à ce qu'un évènement intentionnel ou accidentel provoque son activation.*

Classification des fautes selon leurs sources

Les fautes et leurs sources sont extrêmement diverses. Les trois points de vue principaux selon lesquels on peut les classer sont leur nature, leur origine et leur persistance [Lap92].

La nature des fautes conduit à distinguer :

– les **fautes accidentelles**, qui apparaissent ou sont créées de manière fortuite,

– les **fautes intentionnelles**, qui sont créées délibérément, avec une intention qui peut être présumée nuisible.

L'origine des fautes regroupe elle-même les points de vue suivants :

– la cause phénoménologique, qui conduit à distinguer :

  - les **fautes physiques**, qui sont dues à des phénomènes physiques adverses.
  - les **fautes humaines**, qui résultent d'imperfections humaines ;
- les frontières du système, qui conduisent à distinguer :
  - les **fautes internes**, qui sont les parties de l'état du système qui, lorsqu'activées par les traitements, produiront une ou des erreurs,
  - les **fautes externes**, qui résultent de l'interférence ou des interactions du système avec son environnement physique (perturbations électro-magnétiques, radiation, température, vibrations, ...) ou humain ;
- la phase de création par rapport à la vie du système, qui conduit à distinguer :
  - les **fautes de conception**, qui résultent d'imperfections commises soit au cours du développement du système (de l'expression des besoins à la recette, y compris l'établissement des procédures d'exploitation ou de maintenance), soit au cours de modifications ultérieures,
  - les **fautes opérationnelles**, qui apparaissent durant l'exploitation du système.
- La persistance temporelle conduit à distinguer :
  - les **fautes permanentes**, dont la présence n'est pas reliée à des conditions ponctuelles, internes (processus de traitement) ou externes (environnement),
  - les **fautes temporaires**, qui sont reliées à de telles conditions, et qui sont donc présentes pour une durée limitée.

Classification des fautes selon leurs persistances temporelles

On peut distinguer trois types de fautes suivant leurs persistances temporelles qui peuvent affecter un système :

Définition 2.2 Faute permanente : *Une faute permanente se caractérise par sa durée permanente, une fois activée, durant l'exploitation du système. Elle persiste donc indéfiniment (jusqu'à réparation) après son occurrence.*

Un exemple typique de faute permanente est la faute de conception.

Définition 2.3 Faute transitoire : *Une faute transitoire se caractérise par sa durée limitée, une fois activée, durant l'exploitation du système.*

Ce genre de faute est souvent observée dans les systèmes de communication, où la présence des radiations électromagnétiques peut corrompre les données en-

voyées sur une liaison physique de communication. Ceci provoque une faute transitoire qui ne dure que la période de la présence de ces radiations.

Définition 2.4 Faute intermittente : *Une faute intermittente est une faute transitoire qui se répète arbitrairement.*

Suivant les exigences fonctionnelles et temporelles d'un système réactif, la défaillance de ce système peut être la conséquence de deux sources de fautes : fautes fonctionnelles (ou fautes de valeur) et fautes temporelles.

Définition 2.5 Faute fonctionnelle : *La valeur délivrée par le système est fausse, c'est-à-dire qu'elle n'est pas conforme à sa spécification, ou elle est en dehors de l'intervalle des valeurs attendues.*

Définition 2.6 Faute temporelle : *L'instant auquel la valeur est délivrée est en dehors de l'intervalle de temps spécifié. Dans ce cas, la valeur est considérée soit temporellement délivrée trop tôt, soit trop tard, soit infiniment tard (jamais délivrée). La faute temporelle dans le cas où la valeur n'est jamais délivrée est appelée* **faute par omission**.

Le concept de la tolérance aux fautes a été défini dans la littérature par plusieurs travaux [Avi67, BA97].

Définition 2.7 Tolérance aux fautes : *on dit qu'un système informatique est tolérant aux fautes si ses programmes peuvent être exécutés correctement même en présence de fautes.*

2.2.2 Erreur

Définition 2.8 *L'activation du système peut se manifester par la présence d'un état interne erroné dans ce système. Sous des circonstances particulières cet état interne erroné peut conduire à une* **erreur***, c'est-à-dire à un résultat incorrect ou imprécis.*

2.2.3 Défaillance

Définition 2.9 *Une erreur peut changer le comportement d'un système et provoque le non respect de sa spécification : c'est une* **défaillance** *du système.*

Puisque les hypothèses d'occurrence des fautes diffèrent d'un système à un autre, le type et le niveau de la redondance introduite dans un système dépend directement de ces hypothèses. Les hypothèses de défaillance des systèmes les plus utilisées dans la littérature [Lap92] sont les suivantes (figure 2.2) :

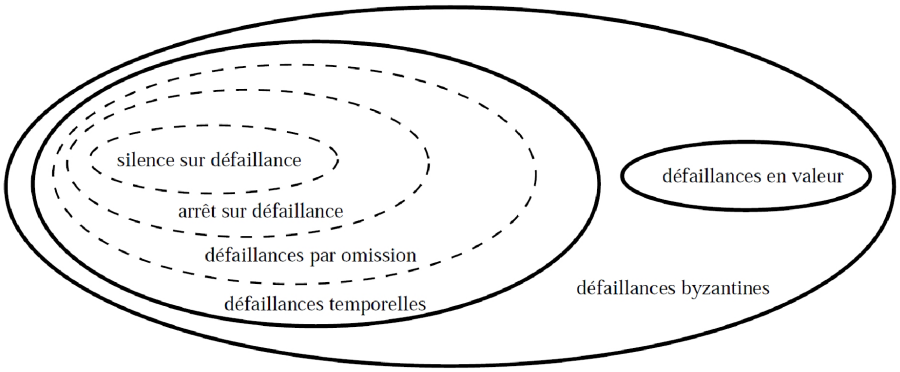


FIGURE 2.2 – Couverture entre les hypothèses de défaillance.

- **systèmes à défaillances en valeur** : ces systèmes supposent que les valeurs sont délivrées à temps et qu'ils défaillent uniquement si les valeurs délivrées sont fausses ;
- **systèmes à silence sur défaillance** : un système à silence sur défaillances soit fonctionne correctement (valeur correcte et délivrée à temps), soit il est défaillant et il cesse de fonctionner ;
- **systèmes à arrêt sur défaillance** : ce sont des systèmes à silence sur défaillance mais qui délivrent un message aux autres systèmes avant qu'il n'arrête son fonctionnement [SS83] ;
- **systèmes à défaillance par omission** : ces systèmes supposent que les valeurs sont correctes et qu'ils défaillent uniquement si la valeur n'est jamais délivrée. Par exemple un système qui perd des messages sortants (omission en émission) ;
- **systèmes à défaillance temporelle** : ces systèmes supposent que les valeurs délivrées sont correctes et qu'ils défaillent uniquement si la valeur est temporellement délivrée trop tôt ou trop tard ;
- **systèmes à défaillance byzantine** : ces systèmes défaillent d'une manière arbitraire, ce mode de défaillance est parfois considéré par des systèmes à très haute fiabilité (nucléaire, spatial).

2.3 Tolérance aux fautes

2.3.1 Principes de la tolérance aux fautes

Détection des erreurs

La détection des erreurs est l'opération la plus importante dans la tolérance aux fautes. Elle permet d'identifier le type et l'origine des erreurs. Cette détection peut être faite soit au niveau de l'environnement du système, soit au niveau de l'application du système [BA97]. Au niveau de l'environnement, c'est l'exécutif de l'application qui se charge de détecter certaines erreurs, qui peuvent être par exemple de type *division par zéro*, *erreur d'entrée/sortie*, *accès interdit au périphérique*. Les techniques de détection d'erreurs au niveau de l'application sont nombreuses. Parmi les techniques de base on trouve la comparaison des résultats de composants logiciels répliqués et la vérification des temps de délivrance des résultats. La réussite d'une telle technique de détection des erreurs dépend de deux paramètres qui sont la *latence* (délai entre la production et la détection de l'erreur), et le *taux de couverture* (pourcentage d'erreurs détectées).

Traitement des erreurs

Cette phase consiste à traiter les états erronés (ou erreurs) détectés par la première phase, à l'aide d'une des deux techniques de base suivante :

- **recouvrement des erreurs** : il consiste à substituer un état exempt d'erreur à l'état erroné. Cette substitution peut prendre deux formes :[AL81] :
  - *reprise* : le système est ramené dans un état survenu avant que l'erreur ne survienne. Cela nécessite d'établir des *points de reprise* qui sont des instants dont l'état courant peut être restaurer ultérieurement ;
  - *poursuite* : la transformation de l'état erroné consiste à trouver un nouvel état à partir duquel le système peut fonctionner souvent en mode dégradé.
- **compensation des erreurs** : la compensation consiste en la reconstruction totale d'un état correct en utilisant un ensemble d'informations redondantes existantes dans le système.

Le choix entre ces deux techniques est un compromis entre plusieurs facteurs, tels que la complexité du système, les contraintes temporelles et matérielles et la criticité du système. Étant donné que nous visons des systèmes réactifs embarqués critiques, nous ne nous intéressons dans ce travail qu'aux techniques basées sur la redondance d'informations, donc la compensation des erreurs.

Nous présentons dans ce qui suit les deux types de tolérance aux fautes, celle qui concerne le logiciel et celle qui concerne le matériel.

2.3.2 Tolérance aux fautes logicielles et matérielles

Un système peut défaillir à cause d'une faute logicielle ou matérielle.

Tolérance aux fautes logicielles

Le problème de conception et d'écriture de logiciels est intrinsèquement difficile [KK07]. Les chercheurs reconnaissent l'existence de difficultés *essentielles* et *accidentelles* pour produire un logiciel correct. Les difficultés essentielles résultent du défi inhérent à la compréhension d'une application et d'un environnement de fonctionnement souvent complexe, alors que les difficultés accidentelles, considérant qu'on ait produit un bon logiciel, proviennent du programmeur qui peut faire des erreurs dans les programmes. Un exemple intéressant de faute logicielle est l'explosion de la fusée Ariane 5 en juin 1996 à cause des erreurs de conception du logiciel. Il existe deux techniques de base pour la tolérance aux fautes logicielles :

- *Uni-version du logiciel* : le but principal des approches utilisant cette technique est de tolérer les fautes logicielles en utilisant une seule version du logiciel, ou d'un de ses composants. Pour tolérer la faute de chaque uni-version du logiciel, le concepteur du système doit la modifier en lui ajoutant des mécanismes de détection et de traitement d'erreurs. Parmi les approches utilisant cette technique, on trouve : le traitement d'exceptions, la détection d'erreur, le point de reprise (check-point and restart),
- *Multi-version du logiciel* : c'est une technique de tolérance aux fautes logicielles basée sur le principe de la redondance logicielle, où chaque composant logiciel est répliqué en plusieurs versions programmées différemment. L'avantage par rapport à la technique précédente est que ces versions logicielles peuvent être exécutées en séquence ou en parallèle pour tolérer les fautes de certaines versions. De plus, les algorithmes implantant ces versions logicielles peuvent être, tous ou certains, développés par différents concepteurs sur différents outils. Parmi les approches utilisant cette technique, on trouve : N-version de programmation. C'est le cas des commandes de vol des avions Airbus et Boeing.

Tolérance aux fautes matérielles

Les tolérances aux fautes matérielles constituent le domaine le plus développé de la tolérance aux fautes générale. Différentes techniques ont été développées et utilisées dans des domaines allant du téléphone aux missions spatiales. On peut tolérer les fautes soit par logiciel soit par matériel. La solution matérielle consiste à répliquer des composant matériels, et même si le coût des processeurs ne cesse de décroître, cette solution augmente la consommation de l'énergie, c'est pourquoi

on opte souvent pour la solution logicielle basée sur la redondance logicielle. Dans la suite du manuscrit, on désigne par *faute* une *faute matérielle*.

Nous ne ferons pas un état de l'art sur la tolérance aux fautes logicielles qui est un domaine de recherche en soi. On fait donc l'hypothèse que le logiciel est sans fautes et donc dans la suite on ne considérera que la tolérance aux fautes matérielles.

2.4 Redondances pour la tolérance aux fautes matérielles

Afin de réaliser la tolérance aux fautes matérielles, deux types de redondances sont utilisés : redondance logicielle et redondance matérielle.

2.4.1 Redondance logicielle

Afin d'illustrer les différents types de redondance logicielle, nous considérons dans cette section le graphe d'algorithme et d'architecture de la figure 2.3. Le graphe d'algorithme (figure 2.3.a) est composé de deux composants logiciels A et B et d'une dépendance de données permettant d'envoyer la donnée m. Le graphe d'architecture (figure 2.3.b) est composé de quatre processeurs P_1, P_2, P_3 et P_4, et de quatre média de communication l_{12}, l_{13}, l_{24} et l_{34}.

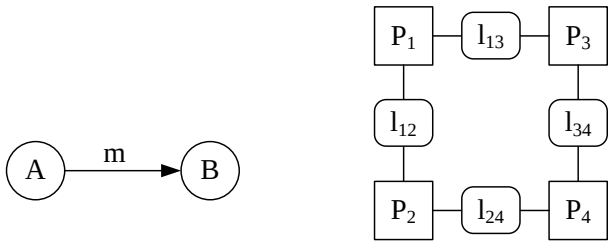


FIGURE 2.3 – Graphes d'algorithme et d'architecture

Les méthodes de tolérance aux fautes matérielles diffèrent selon l'origine des fautes matérielles (processeur, média de communication, capteurs et actionneurs, etc.), et le type de fautes pris en compte (fautes transitoires, fautes permanentes, etc.). La redondance logicielle nécessite une architecture distribuée afin de pouvoir allouer les répliques sur des processeurs différents. C'est la technique de tolérance aux fautes matérielles la plus adaptée aux systèmes embarqués car dans ces

systèmes le nombre de composants matériels ne peut pas être facilement augmenté [GS96, GS97].

On présente dans ce qui suit trois grandes classes d'algorithmes de distribution/ordonnancement temps réel et tolérants aux fautes, à savoir les algorithmes basés sur la *redondance active*, les algorithmes basés sur la *redondance passive* et les algorithmes basés sur la *redondance hybride*, mélange des deux précédentes. Ces algorithmes s'appliquent à deux types de composants matériels : les processeurs et les média de communication.

2.4.1.1 Redondance active

La redondance active est basée sur la réplication des composants logiciels de l'algorithme. Toutes les répliques sont exécutées de sorte à ce qu'en présence de fautes il y'en ait toujours au moins une qui soit exécutée. Le nombre n de répliques de chaque composant logiciel dépend directement du nombre de fautes k et aussi du type de ces fautes. Par exemple, pour tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué en une réplique primaire et en k répliques de sauvegardes, d'où $n = k + 1$.

Définition 2.10 *Une route de communication, notée R, est composée d'un ou plusieurs média de communication : $R = l_{12} \bullet l_{23} \bullet ... \bullet l_{ij} \bullet ...$ où l_{ij} désigne le médium de communication reliant les deux processeurs P_i et P_j i.e. le couple $(P1, P2)$, et $\bullet$ est une relation d'ordre total sur les média.*

Deux routes sont dites disjointes si elles n'ont aucun médium de communication commun.

Tolérance aux fautes des processeurs

Une faute d'un processeur implique l'inactivité de tous les composants logiciels implantés sur ce processeur. La redondance active consiste à répliquer chaque composant logiciel c_i d'un algorithme sur n processeurs pour tolérer au plus $n - 1$ fautes de processeurs.

Dans la figure 2.4.a le composant logiciel A (resp. B) a été alloué au processeur P_1 (resp. P_2). Le composant logiciel A (resp. B) du graphe d'algorithme est répliqué en deux répliques A_1, A_2 (resp. B_1, B_2), qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_4 et P_2) afin de tolérer une faute permanente d'un seul processeur. Les répliques $A1$ et $A2$ envoient le message m aux répliques B_1 et $B2$. Dans la figure 2.4.b, si P_1 défaille, P_2 exécute la réplique A_2 et envoie le message m à P_4 via la route $R = l_{24}$.

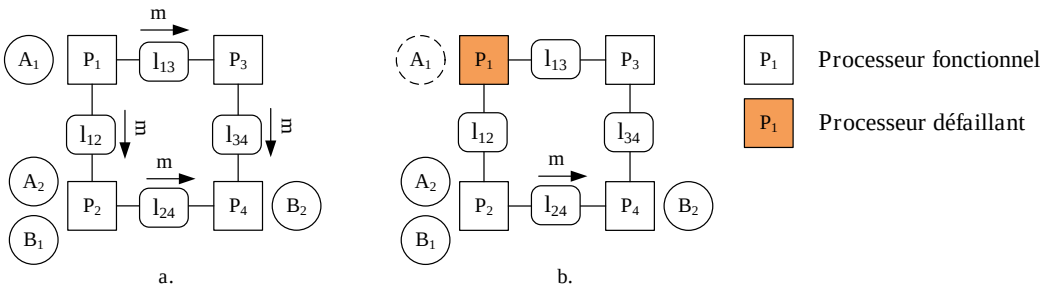


FIGURE 2.4 – Tolérance aux fautes des processeurs par redondance active

Tolérance aux fautes des média de communication

Les fautes des média de communication engendrent la perte de messages dans les réseaux de communication. Ces fautes peuvent être tolérées par la transmission de ces mêmes messages via plusieurs routes disjointes.

Dans la figure 2.5 le message m est transmis via deux routes disjointes $R_1 = l_{13} \bullet l_{34}$ et $R_2 = l_{12} \bullet l_{24}$. Par exemple si le médium l_{13} défaille, le message m est envoyé via la seconde route $R_2 = l_{12} \bullet l_{24}$.

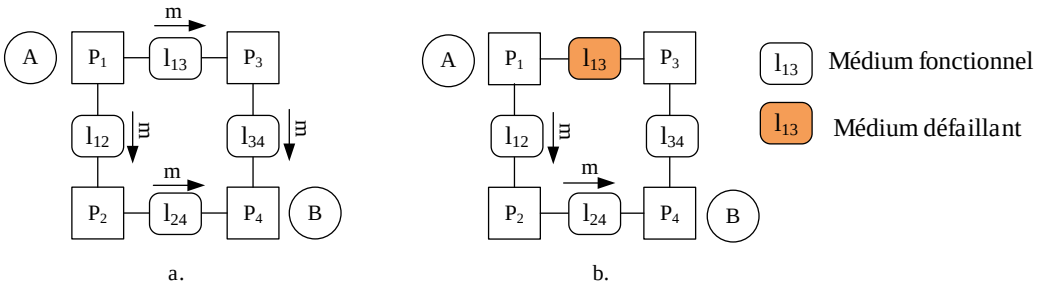


FIGURE 2.5 – Tolérance aux fautes des média de communication par redondance active

2.4.1.2 Redondance passive

La redondance passive est basée aussi sur la réplication des composants logiciels de l'algorithme. A la différence de la redondance active une seule des répliques de chaque composant logiciel, appelée réplique primaire, est exécutée. Alors que les autres répliques, appelées répliques de sauvegardes, ne seront exécutées que si une faute provoque une erreur puis une défaillance du composant matériel implantant la réplique primaire. Le nombre n de répliques de chaque composant logiciel dépend directement du nombre de fautes k et aussi du type de ces fautes. Par exemple, pour tolérer au plus k fautes permanentes de processeurs,

chaque composant logiciel est répliqué en une réplique primaire et en k répliques de sauvegardes, d'où $n = k + 1$.

On présente séparément la tolérance aux fautes des processeurs et des média de communication

Tolérance aux fautes des processeurs

Dans la figure 2.6.a, le composant logiciel A (resp. B) est répliqué en deux répliques, qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_4 et P_2) afin de tolérer une faute permanente d'un seul processeur. Les répliques A_2 et B_1 sont des répliques de sauvegarde qui ne seront exécutées qu'en cas de défaillance, alors que les répliques primaires A_1 et B_2 sont toujours exécutées.

La réplique primaire A_1 envoie le message m à la réplique primaire B_1 via la route $R = l_{13} \bullet l_{34}$. Dans la figure 2.6.b, si P_1 défaille alors il faut détecter sur P_2 la défaillance de P_1, et P_2 exécute la réplique de sauvegarde A_2 et envoie le message m à P_4 via la seule nouvelle route $R' = l_{24}$.

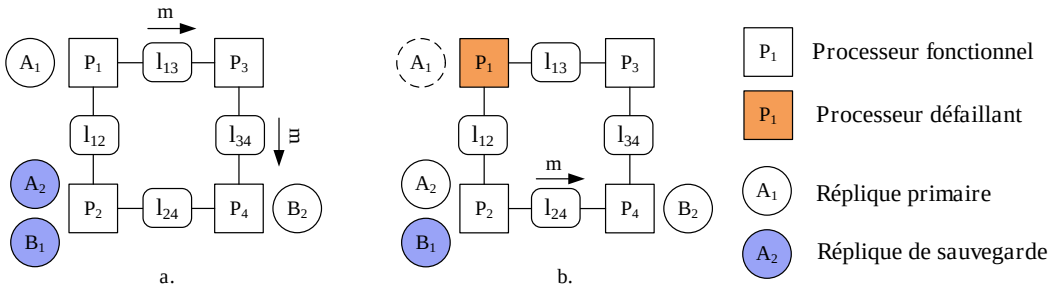


FIGURE 2.6 – Tolérance aux fautes des processeurs par redondance passive

Tolérance aux fautes des média de communication

La perte d'un message dans le cas de la redondance passive des communications, due aux fautes des média de communication, peut être tolérée par la retransmission de ce message.

Dans la figure 2.7.a, le composant logiciel A envoie le message m au composant logiciel B via la route $R = l_{13} \bullet l_{34}$. Dans la figure 2.7.b, si le médium de communication l_{13} défaille alors il faut détecter sur le processeur P_1 la défaillance de l_{13} et envoyer le message m à P_4 via la route $R' = l_{12} \bullet l_{24}$.

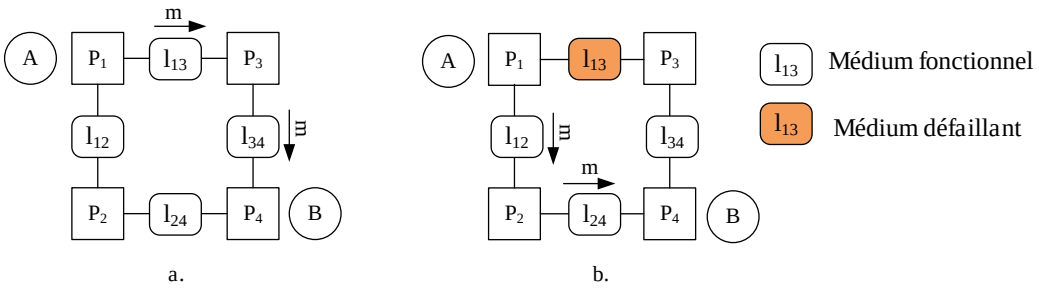


FIGURE 2.7 – Tolérance aux fautes des média de communication par redondance passive

2.4.1.3 Redondance hybride

La redondance hybride est une combinaison de la redondance active et passive des composants logiciels. Par exemple, pour tolérer une fautes permanente d'un processeur ou d'un médium de communication, on utilise la redondance active pour les composants logiciels de l'algorithme et la redondance passive pour les communications.

Dans la figure 2.8.a les deux composants logiciels A et B ont des répliques *actives*, tandis que les communications ont des répliques *passives*. Le composant logiciel A (resp. B) du graphe d'algorithme est répliqué en deux répliques actives A_1, A_2 (resp. B_1, B_2), qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_4 et P_2) afin de tolérer une faute permanente d'un seul processeur. Puisque les communications ont des répliques passives, le message m ne sera envoyé par la réplique A_1 que via la route $R = l_{13} \bullet l_{34}$.

Dans la figure figure 2.8.b, si le médium l_{13} défaille alors il faut que le processeur P_2 détecte la défaillance de ce médium et renvoie le message m à P_4 via la nouvelle route $R' = l_{24}$.

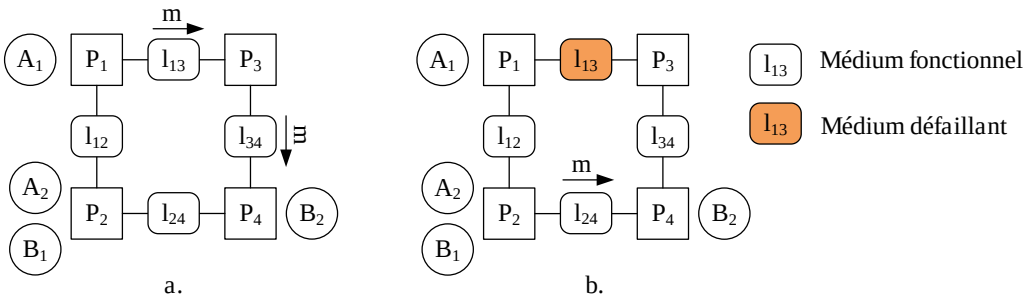


FIGURE 2.8 – Tolérance aux fautes des média de communication par redondance hybride

2.4.1.4 Comparaison entre les trois types de redondance

Le tableau 4.1 montre une comparaison entre les différentes approches de tolérance aux fautes basées sur la redondance active, passive ou hybride des composants logiciels d'un algorithme [Kal04].

Une propriété intéressante de la redondance active vient du fait qu'une faute n'augmente pas la latence du système temps réel, ce qui n'est pas le cas dans la redondance passive, où la faute de la réplique primaire peut de manière significative augmenter la latence du système. Cependant, la redondance passive présente l'avantage de réduire la surcharge sur les processeurs et sur le réseau de communication, ce qui permet une meilleure exploitation des ressources matérielles offertes par l'architecture. Le choix d'une telle stratégie de réplication se fait en fonction des contraintes et des besoins applicatifs. Par exemple, les concepteurs de systèmes réactifs tolérants aux fautes préfèrent utiliser la réplication active en cas de défaillances fréquentes des composants matériels, et la réplication passive lorsque le nombre de communications est élevé.

Critère de comparaison	**Redondance active**	**Redondance passive**	**Redondance hybride**
Sûrcout	un surcôut élevé	un surcôut moins élevé	le surcôut dépend du niveau de la réplication active par rapport à la réplication passive
Traitement de défaillances (Temps de réponse)	un temps de réponse prévisible, et généralement rapide dans des architectures offrant un taux élevé de parallélisme	meilleur temps de réponse en absence de défaillances. La défaillance de la réplique primaire peut de manière significative augmenter le temps de réponse	le temps de réponse dépend du niveau de la réplication active par rapport à la réplication passive
Réprise après défaillance	immédiate	non immédiate	non immédiate

TABLE 2.1 – Comparaison entre les trois approches de redondance

2.4.2 Redondance matérielle

2.4.2.1 Généralités

Dans beaucoup de systèmes dont la sûreté de fonctionnement est critique, comme les commandes de vol ou les circuits hydrauliques dans les avions, certains actionneurs et capteurs sont parfois triplés . Une défaillance d'un composant peut donc être compensée par les deux autres composants. Puisque la défaillance de chaque composant est indépendante de celle des deux autres, et que les composants sont supposés avoir une bonne fiabilité, la probabilité que les trois composants soit fautifs est extrêmement petite.

La redondance matérielle est présente dans divers domaines, elle est utilisée pour la tolérance aux fautes des circuits logiques nano-électroniques, dans la conceptions de circuits intégrés qui résistent au radiations de l'espace [ML03], etc.

Supposons que nous appliquons les mêmes entrées à deux circuits logiques et que nous comparons les sorties. Si nous obtenons le même résultat, nous pouvons conclure que les deux composants sont fonctionnels (ce qui est fort probable), ou qu'ils présentent une défaillance tous les deux (situation peu probable). Le problème se pose si l'un des deux défaille, dans ce cas on ne peut pas savoir lequel des deux est défectueux juste en comparant leurs sorties. C'est le principal inconvénient des systèmes DMR (Dual Modural Redundant). En rajoutant un composant à la redondance la situation s'améliore : avoir les mêmes sorties implique que les trois composants sont fonctionnels (cas très probable) ou qu'ils défaillent simultanément (cas très peu probable), et si on a deux sorties identiques il est plus probable que le composant ayant une sortie différente défaille. Cette redondance est appelée *redondance modulaire triple* TMR (Triple Modular Redundancy) [BWM92, SY91].

2.4.2.2 Redondance modulaire triple

La figure 2.9 montre un circuit TMR (Triple Modular Redundant) avec trois circuits logiques identiques A, B et C ayant la même entrée [xZbX09, DK08]. Les sorties sont comparées à l'aide du composant $voteur$ qui donne un résultat correct à la majorité des votes. Si aucun ou l'un des composant défaille le résultat délivré est correct, mais si les deux ou les trois composant défaillent, le résultat sera erroné. Certains circuits défaillant ont une sortie mise à 1 ou à 0, dans ce cas cette information sera prise en compte par le voteur.

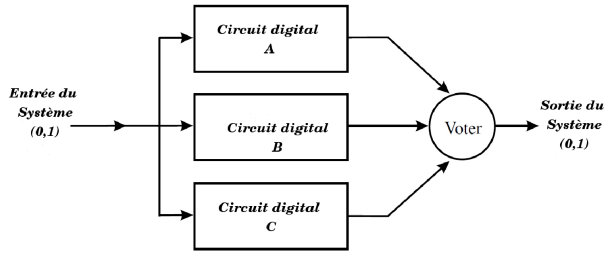


FIGURE 2.9 – Rodondance modulaire triple.

2.4.2.3 Cas général de la redondance modulaire

La redondance modulaire NMR (N-Modular Redundancy) [LPAHP01, BWM92, BCS69] est une généralité de la TMR. Elle présente l'avantage de *la majorité à raison*. Dans ce cas la probabilité que le résultat obtenu en sortie soit correct est plus élevée que celle obtenu par la TMR.

Cependant, on ne peut pas augmenter indéfiniment le nombre de redondance matérielles à cause des problèmes de coût, de consommation électrique, etc.

2.5 Conclusion

Nous avons présenté dans ce chapitre les terminologies et les principes de la tolérance aux fautes. Comme nous nous intéressons qu'aux fautes matérielles (processeurs et média de communication), nous avons présenté deux types de redondances pour la tolérance aux fautes matérielles : redondance logicielle et redondance matérielle. Pour la redondance logicielle nous avons présenté trois types de redondances : active, passive et hybride que nous avons comparées. Enfin nous avons présenté les principes de la redondance matérielle basée sur la redondance modulaire.

Ce chapitre conclut la première partie consacrée à l'état de l'art. Dans la partie suivante nous faisons une étude d'ordonnançabilité de tâches périodiques strictes et de tâches sporadiques dans les deux cas monoprocesseur et multiprocesseur.

Deuxième partie

Ordonnancement temps réel monoprocesseur

Chapitre 3

Ordonnancement temps réel monoprocesseur non préemptif périodique strict

3.1 Introduction

Comme nous nous intéressons aux applications de contrôle/commande robotiques critiques, les tâches d'entrées/sorties gérant les capteurs/actionneurs ne doivent pas avoir de gigue sur les entrées et les sorties. De plus la gigue sur les entrées peut causer des pertes de données issues des capteurs. Par ailleurs le temps de réponse d'une tâche NPPS est constant et est égal à sa durée d'exécution. En revanche le temps de réponse d'une tâche préemptive est variable et dépend de la durée d'exécution des tâches plus prioritaires qui viennent préempter cette tâche, et dépend aussi du coût de la préemption en fonction du RTOS (Real Time Operating System) utilisé. Les tâches préemptives ont alors des temps de réponse qui varient d'une instance à une autre ce qui introduit une gigue de sortie. De plus la gigue sur les sortie peut causer une dégradation des performances dans les systèmes de contrôle/commande bouclés, comme elle peut être une source d'instabilité dans de tels systèmes lorsqu'elle n'est pas prise en compte [MFFR01, Lin02, BI07, BRC08].

Comme nous avons fait l'hypothèse que le logiciel était sans faute (cf. section 2.3.2), nous considérons des tâches temps réel *NPPS* afin d'éviter les problèmes cités ci-dessus. Le fait que les tâches soient non préemptives permet de proposer des algorithmes d'ordonnancement de type *hors ligne*. Ainsi dans les chapitres 3, 5 et 6 les algorithmes d'ordonnancement proposés sont de type *hors ligne*. Dans ce cas l'ordonnancement est réalisé en temps réel en lisant une *table d'ordonnancement* contenant les dates de début d'exécutions de toutes les instances de tâches.

On pourrait aussi le réaliser en temps réel en utilisant un ordonnancement en ligne qui, quelle que soit sa politique, choisirait une tâche dont la date de début d'exécution correspondrait à celle donnée dans la table précédente. Un tel algorithme d'ordonnancement en ligne serait de type *offset free* [GDF97, GGN06].

En revanche les algorithmes d'ordonnancement du chapitre 4 qui traite de la combinaison de tâches NPPS et de tâches NPPS, sont eux de type *en ligne*. Comme certaines tâches sont NPPS, ces dernières sont asynchrones.

Dans ce chapitre on présente l'analyse d'ordonnançabilité monoprocesseur de tâches temps réel NPPS. On commence par donner la stratégie d'ordonnancement, ensuite on étudie le cas particulier des tâches harmoniques où toutes les périodes des tâches sont multiples les unes par rapport aux autres, puis on généralisera pour le cas des tâches non harmoniques. Dans les deux cas nous proposons des conditions d'ordonnançabilité et des algorithmes d'ordonnancement. Finalement on décrit la phase transitoire et la phase permanente d'un ordonnancement de tâches NPPS.

3.2 Stratégie d'ordonnancement

On considère un ensemble $\Gamma_n = \{\tau_i(s_i, C_i, T_i), i = 1, ..., n\}$ de n tâches NPPS où la première date de début d'exécution s_i^0 est notée s_i. On rappelle que le problème consistant à trouver un ordonnancement de tâches NPPS est NP-difficile. Pour trouver un ordonnancement valide, deux cas sont possibles : soit on dispose d'un algorithme d'ordonnancement utilisant une condition d'ordonnançabilité pour l'ensemble des tâches, soit ce n'est pas le cas. Dans ce dernier cas on construit un ordonnancement valide de façon itérative (heuristique) en appliquant une condition d'ordonnançabilité sur un sous-ensemble de tâches déjà ordonnancées et une nouvelle tâches *candidate* à l'ordonnancement. Une telle condition d'ordonnançabilité est plus facile à trouver qu'une condition pour l'ensemble des tâches de Γ_n, car ce problème est NP-difficile.

Soit Γ_s un sous-ensemble de Γ_n qui va contenir les tâches ordonnançables de Γ_n, initialisé avec la tâche τ_1. On appelle tâche *candidate* la tâche τ_i à ordonnancer à la i^{me} itération. Si les tâches du sous-ensemble $\Gamma_s \cup \{\tau_i\}$ satisfont la condition d'ordonnançabilité alors τ_i est ordonnançable et est incluse dans Γ_s, sinon τ_i n'est pas ordonnançable et par conséquent Γ_n ne l'est pas. Cette stratégie permet à la fois de déterminer si un ensemble de tâches est ordonnançable ou non, et de trouver le sous-ensemble de tâches ordonnançables.

3.3 Analyse d'ordonnançabilité de tâches harmoniques

On commence par l'ordonnancement de tâches harmoniques pour lesquelles toutes les périodes sont multiples les unes des autres.

On considère un ensemble de tâches harmoniques $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$. L'analyse d'ordonnançabilité qui suit est fondée sur un tri des tâches τ_i selon l'ordre croissant de leurs périodes.

3.3.1 Arbre d'ordonnancement

Nous allons illustrer l'analyse d'ordonnançabilité à l'aide d'un exemple de trois tâches harmoniques triées selon l'ordre croissant de leurs périodes $\tau_1(C_1, T_1)$, $\tau_2(C_2, T_2)$ et $\tau_3(C_3, T_3)$, avec $T_2 = 3T_1$ et $T_3 = 2T_2 = 6T_1$.

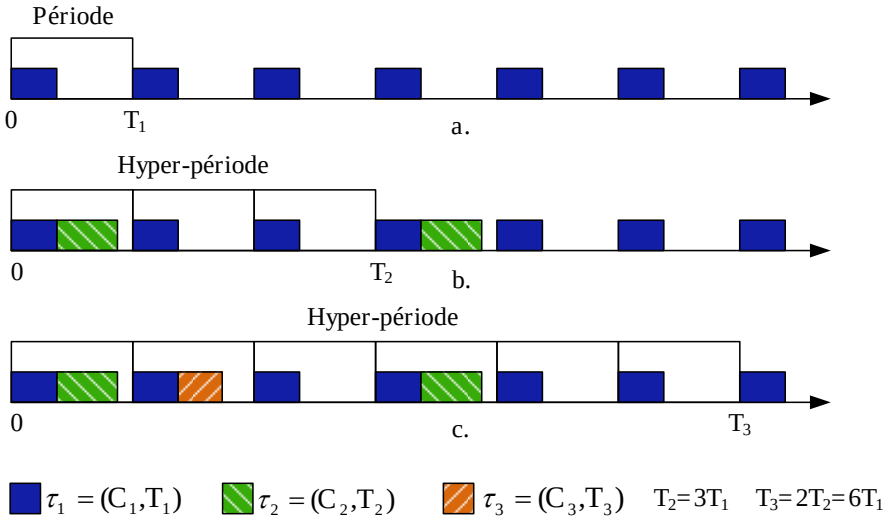


FIGURE 3.1 – Ordonnancement de tâches harmoniques

La figure 3.1 montre le diagramme d'ordonnancement de ces trois tâches. On commence par ordonnancer la tâche τ_1 (figure 3.1.a) puis on ordonnance la tâche τ_2 (figure 3.1.b). L'hyper-période des tâches ordonnancées τ_1 et τ_2 est égale au $PPCM(T_1, T_2) = T_2 = 3T_1$, donc l'ordonnancement de ces deux tâches est une répétition de l'ordonnancement inclus dans l'intervalle $[0, T_2]$. Dans cet intervalle, la tâche τ_1 est ordonnancée immédiatement après la τ_2, donc sans temps creux, selon un algorithme First-Fit. En effet ce problème d'ordonnancement est équivalent au problème de remplissages de boîtes avec des objets de tailles différentes (bin-packing), où les boîtes sont les intervalles $[kT_1, (k+1)T_1],\ k \in \mathbb{N}$, et les objets sont les durées d'exécution des tâches.

La figure 3.1.c montre l'ordonnancement des trois tâches τ_1, τ_2 et τ_3 qui a une hyper-période égale au $PPCM(T_1, T_2, T_3) = T_3 = 2T_2$. L'ordonnancement de ces trois tâches est une répétition de l'intervalle $[0, T_3]$.

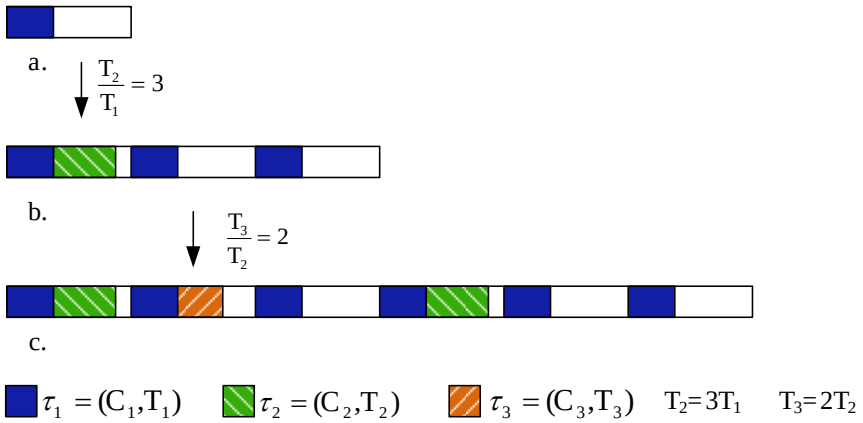


FIGURE 3.2 – Réduction de l'ordonnancement à l'hyper-période

Nous pouvons réduire l'analyse d'ordonnançabilité à l'hyper-période de chaque sous-ensemble ordonnancé, parce que l'ordonnancement à l'intérieur du premier intervalle de temps de longueur égale à l'hyper-période est répétée indéfiniment. La figure 3.2 montre le diagramme d'ordonnancement réduit des tâches τ_1, τ_2 et τ_3. La figure 3.2.a montre la tâche τ_1 sur une période T_1. La figure 3.2.b montre les tâches τ_1 et τ_2 sur une hyper-période égale à $T_2 = 3T_1$ et la figure 3.2.c montre les tâches τ_1, τ_2 et τ_3 sur l'hyper-période égale à $T_3 = 2T_2$.

Les diagrammes d'ordonnancement de la figure 3.2 peuvent être représentés en utilisant un arbre d'ordonnancement comme le montre la figure 3.3. Le tronc de l'arbre est une case de taille égale à la période T_1. Il correspond au diagramme de la figure 3.2.a. L'objet qui occupe cette case est de taille égale à la durée d'exécution C_1 de la tâche τ_1 et le temps creux dans lequel on ordonnance la deuxième tâche τ_2 est de longueur $T_1 - C_1$. Le tronc de cet arbre a $(\frac{T_2}{T_1} = 3)$ fils identiques. La 2^{me} tâche τ_2 est ordonnancée dans un temps creux disponible dans l'un de ces trois fils. Cela correspond à la figure 3.2.b. Chacun des trois fils (nœuds) aura $(\frac{T_3}{T_2} = 2)$ fils qui sont les feuilles de l'arbre. La tâche τ_3 est ordonnancée dans un temps creux disponible dans l'un de ces deux fils. Cela correspond à la figure 3.2.c.

Afin d'obtenir l'ordonnancement des tâches, il faut mettre les feuilles de l'arbre dans le bon ordre. Pour cela, à une itération donnée, l'ensemble des nœuds représente l'exécution des tâches dans une hyper-période. À l'itération suivante le premier fils de n'importe quel nœud sera exécuté dans la première hyper-période,

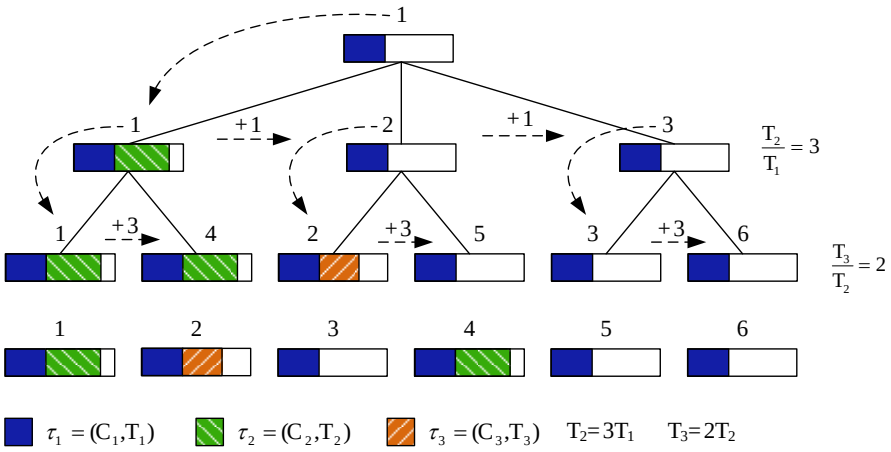


FIGURE 3.3 – Arbre d'ordonnancement de tâches harmoniques

le deuxième nœud sera exécuté dans la deuxième hyper-période ainsi de suite. On peut ordonner les nœuds de la manière suivante : le tronc de l'arbre prend l'ordre 1, ensuite à chaque itération i, l'ordre l de chaque nœud est hérité par son premier fils, et l'ordre du fils suivant est augmenté de $\frac{T_{i-1}}{T_1}$. Donc l'ordre du k^{eme} fils est donné par

$$l + (k-1)\frac{T_{i-1}}{T_1}$$

$\frac{T_{i-1}}{T_1}$ étant le nombre de nœuds à l'itération précédentes $i-1$. En connaissant l'ordre des feuilles on peut facilement calculer les dates de début d'exécution de toutes les tâches.

La figure 3.3 illustre ce résultat qui montre l'ordre des nœuds à chaque itération conduisant à l'ordonnancement donné par la figure 3.2.c.

De manière générale, l'ordonnancement de tâches harmoniques se fait itérativement en commençant par la tâche τ_1. Le tronc de l'arbre a $\frac{T_2}{T_1}$ fils identiques. À la i^{me} itération chaque fils a $\frac{T_i}{T_{i-1}}$ fils identiques. La i^{me} tâche τ_i est ordonnancée dans un temps creux disponible dans l'un de ces fils. Une tâche $\tau_i(C_i, T_i)$ est ordonnançable s'il existe, parmi tous les fils de la i^{me} itération, un fils qui contient un temps creux supérieur ou égal à la durée d'exécution C_i. Le nombre de feuilles de cet arbre est donc égal à

$$\frac{T_i}{T_{i-1}} \cdot \frac{T_{i-1}}{T_{i-2}} \cdot \ldots \cdot \frac{T_2}{T_1} = \frac{T_i}{T_1}.$$

F. Eisenbrand [EHN+10b, EHN+10a] a utilisé ce principe d'arbre d'ordonnancement de tâches harmoniques pour étudier la complexité du problème et montrer

que les tâches sont ordonnançables mais sans donner de conditions d'ordonnançabilité ni proposer d'ordonnancement, alors que nous allons donner des conditions d'ordonnançabilité que nous utilisons pour produire un ordonnancement valide.

Dans la section suivante nous étudions les conditions d'ordonnançabilité en distinguant deux types de tâches harmoniques :

1. toutes les tâches ont des périodes distinctes,
2. toutes les tâches n'ont pas des périodes distinctes, avec deux sous-cas :
   (a) les tâches ayant les mêmes périodes ont les mêmes WCETs,
   (b) les tâches ayant les mêmes périodes ont des WCETs distincts.

3.3.2 Toutes les tâches ont des périodes distinctes

On considère l'ensemble des tâches harmoniques $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$ rangé selon l'ordre croissant de leurs périodes

$$\forall i = 2, ..., n,\ T_i \geq T_{i-1}.$$

Comme toutes les périodes sont distinctes on a

$$\forall i = 2, ..., n,\ T_i > T_{i-1}$$

Comme les tâches sont harmoniques, ceci peut aussi s'écrire sous la forme :

$$\forall i = 2, ..., n,\ T_i \geq 2 \cdot T_{i-1}$$

ou encore

$$\forall i = 2, ..., n,\ \frac{T_i}{T_{i-1}} \geq 2.$$

Le théorème (3.1) donne une condition d'ordonnançabilité *nécessaire et suffisante* pour ce cas de tâches harmoniques.

Théorème 3.1 *Soit* $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$ *un ensemble de tâches harmoniques tel que* $\forall i = 2, ..., n,\ T_i > T_{i-1}$. *L'ensemble* Γ_n *est ordonnançable* ***si et seulement si***

$$\forall i = 2, ..., n,\ C_i \leq T_1 - C_1 \tag{3.1}$$

Preuve
On considère un ensemble de tâches harmoniques Γ_n tel que
$\forall i = 2, ..., n,\ T_i > T_{i-1}$ ou encore $\forall i = 2, ..., n,\ \frac{T_i}{T_{i-1}} \geq 2$. Comme chaque nœud de l'arbre d'ordonnancement comporte $\frac{T_i}{T_{i-1}}$ fils (figure 3.4), tous les nœuds de l'arbre d'ordonnancement de Γ_n ont au moins deux fils.

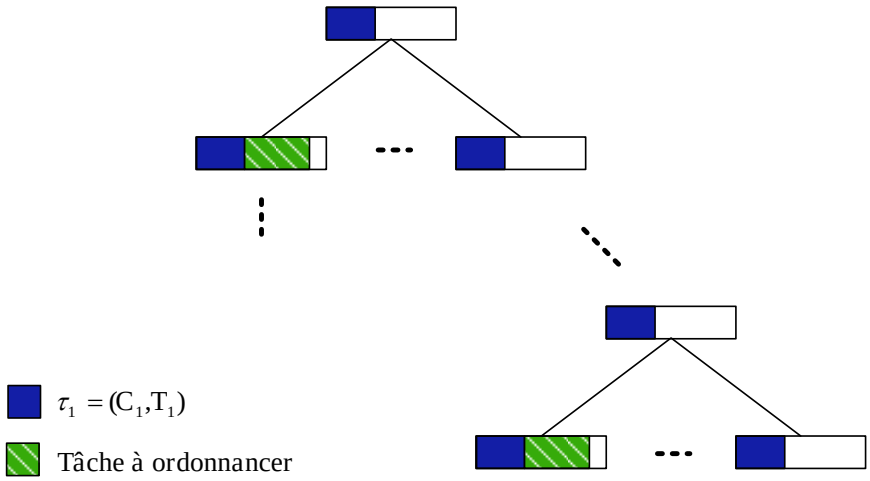


FIGURE 3.4 – Arbre d'ordonnancement de tâches harmoniques

On commence par prouver que la condition (3.1) est suffisante. Pour ce faire on doit prouver que si la condition (3.1) est satisfaite avec $\forall i = 2, ..., n,\ \frac{T_i}{T_{i-1}} \geq 2$ alors l'ensemble des tâches Γ_n est ordonnançable.

Pour cela on ordonnance la première tâche $\tau_1(C_1, T_1)$. Le tronc de l'arbre a un temps creux de taille $T_1 - C_1$. Comme $\frac{T_2}{T_1} \geq 2$, le tronc de l'arbre a au moins deux fils identiques. Comme $C_1 + C_2 \leq T_1$ (équation 3.1) ou encore $C_2 \leq T_1 - C_1$, alors τ_2 peut être ordonnancée dans le temps creux du premier fils qui a un temps creux de taille $T_1 - C_1$, et il nous reste au moins un fils identique au tronc.

À l'itération suivante, un des fils identiques au tronc aura $(\frac{T_3}{T_2})$ fils. Comme $\frac{T_3}{T_2} \geq 2$, l'un de ces fils sera utilisé pour ordonnancer la tâches τ_3 et le reste des fils sera utilisé pour ordonnancer la tâche suivante. On procède de la même manière jusqu'à ordonnancer la dernière tâche τ_n. Donc si la condition (3.1) est satisfaite alors Γ_n est ordonnançable.

On prouve maintenant la nécessité de la condition (3.1) : si Γ_n est ordonnançable alors la condition (3.1) est vérifiée. On procède par contraposée et on doit prouver que si la condition (3.1) n'est pas vérifiée alors Γ_n n'est pas ordonnançable.

La condition (3.1) n'est pas vérifiée veut dire que :

$$\exists i \geq 2,\ C_1 + C_i > T_1$$

ou encore

$$\exists i \geq 2,\ C_i > T_1 - C_1$$

On commence par ordonnancer la tâche τ_1. Les temps creux restées entres deux instances successives de τ_1 ont une taille de $T_1 - C_1$. Comme il existe une tâche τ_i qui a une durée d'exécution supérieure à $T_1 - C_1$ alors cette tâche n'est pas ordonnançable, donc l'ensemble des tâches Γ_n n'est pas ordonnançable. ∎

Remarque

On peut déduire du théorème précédent que quel que soit le nombre de tâches harmoniques à ordonnancer, ces tâches sont ordonnançables *si et seulement si* elles satisfont la condition $\forall i = 2, ..., n,\ C_i \leq T_1 - C_1$ avec $T_i > T_{i-1}$.

Ce résultat reste vrai même dans le cas où le nombre de tâches tend vers l'infini. Ce résultat est contre intuitif car on imagine que la charge du processeur va tendre vers l'infini quand le nombre de tâche tendra vers l'infini, mais cela n'est pas vrai comme on va le montrer.

Calculons la charge U du processeur :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Dans le cas limite où

$$\forall i = 2, ..., n,\ C_i = T_1 - C_1$$

et

$$\frac{T_i}{T_{i-1}} = 2$$

on a

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{C_1}{T_1} + \frac{T_1 - C_1}{2T_1} + \frac{T_1 - C_1}{2T_2} + \frac{T_1 - C_1}{2T_3} + ... + \frac{T_1 - C_1}{2T_{n-1}}$$

$$= \frac{C_1}{T_1} + \frac{T_1 - C_1}{2T_1} + \frac{T_1 - C_1}{2^2 T_1} + \frac{T_1 - C_1}{2^3 T_1} + ... + \frac{T_1 - C_1}{2^{n-1} T_1}$$

$$= \frac{C_1}{T_1} + \frac{T_1 - C_1}{T_1} \sum_{i=1}^{n-1} \left(\frac{1}{2}\right)^i$$

donc

$$\lim_{n \to +\infty} U = \frac{C_1}{T_1} + \frac{T_1 - C_1}{T_1} \sum_{i=1}^{+\infty} \left(\frac{1}{2}\right)^i$$

et comme la série de terme général $\left(\frac{1}{2}\right)^n$ est convergente et sa somme vaut

$$\sum_{i=1}^{+\infty} \left(\frac{1}{2}\right)^i = 1$$

on a

$$\lim_{n \to +\infty} U = \frac{C_1}{T_1} + \frac{T_1 - C_1}{T_1} \cdot 1 = 1.$$

Donc quel que soit le nombre de tâches à ordonnancer, la charge du processeur n'excédera jamais 1.

La figure 3.5 montre un arbre d'ordonnancement de tâches harmoniques qui illustre ce résultat. On voit bien qu'il reste toujours une feuille de l'arbre qui est identique au tronc avec un temps creux de taille $T_1 - C_1$.

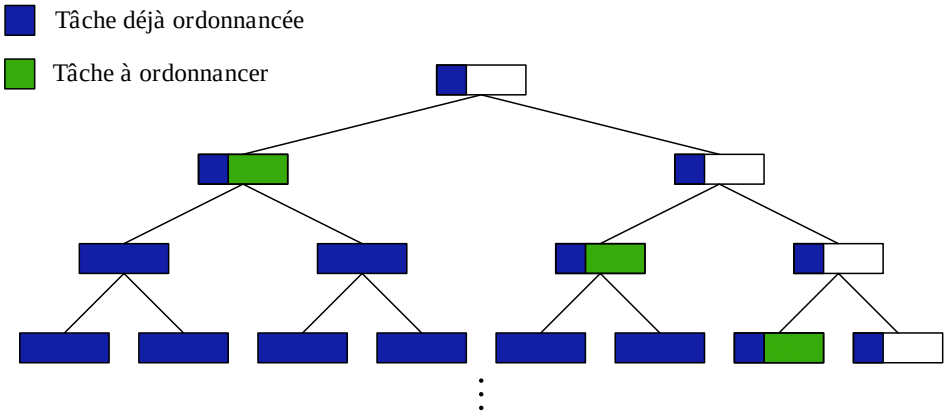


FIGURE 3.5 – Arbre d'ordonnancement de tâches harmoniques

3.3.3 Certaines tâches ont des périodes identiques

Notations

Afin de simplifier les formules mathématiques et puisque certaines tâches ont des périodes identiques, nous proposons les notations suivantes : une tâche $\tau_{i,j}(C_{i,j}, T_{i,j})$ est caractérisée par une durée d'exécution $C_{i,j}$ et une période $T_{i,j}$. Pour un i donné, toutes les tâches $\tau_{i,j}$ ont la même période, i.e. $\forall j \neq j',\ T_{i,j} = T_{i,j'} = T_i$. On considère un ensemble de tâches $\Gamma_n = \{\tau_{i,j}(C_{i,j}, T_{i,j}),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ à ordonnancer tel que $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$.

Dans un premier temps nous traitons le cas où les tâches ont des périodes identiques et ont aussi des durées d'exécution identiques.

Certaines tâches ont des périodes identiques et des WCETs identiques

Dans ce premier cas $\forall j \neq j',\ T_{i,j} = T_{i,j'} = T_i,\ C_{i,j} = C_{i,j'} = C_i$. L'ensemble $\Gamma_n = \{\tau_{i,j}(C_i, T_i),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ contient donc n sous-ensembles de m_i tâches ayant la même période T_i et la même durée d'exécution C_i.

Le théorème suivant donne une condition d'ordonnançabilité suffisante pour ce type d'ensemble de tâches.

Théorème 3.2 *Un ensemble de tâches harmoniques* $\Gamma_n = \{\tau_{i,j}(C_i, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ *où* $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$ *est ordonnançable* **si**

$$\begin{cases} T_1 - m_1 \cdot C_1 > 0 \\ \forall i = 2, ..., n,\ a_i \geq 0 \end{cases} \tag{3.2}$$

avec

$$a_i = a_{i-1} \cdot \frac{T_i}{T_{i-1}} - \left\lceil \frac{m_i}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_i} \right\rfloor} \right\rceil, i = 2, ..., n$$

et

$$a_1 = 1$$

Preuve

On considère l'ensemble de tâches $\Gamma_n = \{\tau_{i,j}(C_i, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ où $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$.

Pour $i = 1$ on commence par ordonnances les m_1 tâches $\tau_{1,j}(C_1, T_1)$ ayant la plus petite période T_1. Comme ces tâches ont la même période, elles sont ordonnançables si la somme de leurs durées d'exécution est inférieure ou égale à leur période, donc on a

$$\sum_{j=1}^{m_1} C_1 = m_1 C_1 \leq T_1.$$

Le cas limite où $\sum_{j=1}^{m_1} C_1 = T_1$ veut dire que l'ordonnancement des tâches $\tau_{1,j}$ ne laisse aucun temps creux pour ordonnancer les tâches $\tau_{1,j}$, donc on a

$$\sum_{j=1}^{m_1} C_1 = m_1 C_1 > T_1$$

d'où

$$T_1 - m_1 C_1 > 0.$$

Le tronc de l'arbre d'ordonnancement est une case de taille égale à T_1 qui contient un temps creux d'une taille égale à $T_1 - m_1 C_1 > 0$.

À la i^{me} itération on ordonnance les tâches $\tau_{i,j}$ dans les fils identiques au tronc. On considère les autres fils sont entièrement remplis. On définit la variable a_i comme étant le nombre de fils restant après l'ordonnancement des tâches $\tau_{i,j}$.

Pour $i = 2$ le tronc a $(\frac{T_2}{T_1})$ fils. Le nombre de fils ayant des temps creux est égal à $(a_1 \cdot \frac{T_2}{T_1})$. Chacun de ces fils peux contenir $\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_2} \right\rfloor$ tâches $\tau_{2,j}(C_2, T_2)$. Donc le nombre de fils qui peuvent contenir les m_2 tâches $\tau_{2,j}$ est égal à

$$\left\lceil \frac{m_2}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_i} \right\rfloor} \right\rceil$$

et le le nombre de fils restant libres (identiques au tronc) est étal à

$$a_2 = a_1 \cdot \frac{T_2}{T_1} - \left\lceil \frac{m_2}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_2} \right\rfloor} \right\rceil$$

Les tâches $\tau_{2,j}$ sont donc ordonnançables si $a_2 \geq 0$.

Pour $i = k$ on suppose que toutes les tâches $\tau_{i,j},\ i < k$ ont été ordonnancées. On dispose à cette itération de $a_{k-1} \cdot \frac{T_k}{T_{k-1}}$ fils identiques au tronc. Chacun de ces fils peut contenir $\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_k} \right\rfloor$ tâches $\tau_{k,j}$, et le nombre de fils qui contiennent ces tâches est égal à $\left\lceil \frac{m_k}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_k} \right\rfloor} \right\rceil$. Donc le nombre de fils identiques au tronc restant est égal à

$$a_k = a_{k-1} \cdot \frac{T_k}{T_{k-1}} - \left\lceil \frac{m_k}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_k} \right\rfloor} \right\rceil .$$

Les tâches $\tau_{k,j}$ sont donc ordonnançables si $a_2 \geq 0$.

On suit le même raisonnement jusqu'à la n^{me} itération où les tâches $\tau_{n,j}$ seront ordonnançables si $a_n \geq 0$. ∎

Dans le théorème suivant nous réduisons le système d'équations de la condition 3.2 à une seule équation afin de simplifier les calculs.

Théorème 3.3 *Un ensemble de tâches harmoniques* $\Gamma_n = \{\tau_{i,j}(C_i, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ *où* $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$ *est ordonnançable* **si**

$$\frac{T_n}{T_1} - \sum_{i=2}^{n} \frac{T_n}{T_{i-1}} \left\lceil \frac{m_i}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_i} \right\rfloor} \right\rceil \geq 0 \tag{3.3}$$

avec $T_1 - m_1 \cdot C_1 > 0$

Preuve
On considère l'ensemble de tâches $\Gamma_n = \{\tau_{i,j}(C_i, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ où $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$.

Pour $i = 2, ..., n$ on définit deux variables *strictement positives* $\alpha_i = \frac{T_i}{T_{i-1}}$ et $\beta_i = \left\lceil \frac{m_k}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_i} \right\rfloor} \right\rceil$. On a donc $a_i = \alpha_i \cdot a_{i-1} - \beta_i$.

D'après la condition 3.2 du théorème 3.2 on a

$$\begin{aligned} a_n \geq 0 \quad &\Leftrightarrow \alpha_n \cdot a_{n-1} - \beta_n \geq 0 \\ &\Leftrightarrow \alpha_n \cdot a_{n-1} \geq \beta_n \\ &\Leftrightarrow a_{n-1} \geq (\tfrac{\beta_n}{\alpha_n}) \\ &\Rightarrow a_{n-1} \geq 0 \end{aligned} \tag{3.4}$$

Donc

$$(a_n \geq 0) \Rightarrow (a_{n-1} \geq 0) \Rightarrow ... \Rightarrow (a_2 \geq 0) \Rightarrow (a_1 \geq 0).$$

Il suffit donc de montrer que $(a_n \geq 0)$ pour en déduire que $\forall i = 1, ..., n-1,\ a_i \geq 0$. On a

$$\begin{aligned} a_n &= \alpha_n \cdot a_{n-1} - \beta_n \\ &= \alpha_n \cdot (\alpha_{n-1} \cdot a_{n-2} - \beta_{n-1}) - \beta_n \\ &... \\ &= \alpha_n ... \alpha_2 \cdot a_1 - \alpha_n ... \alpha_3 \cdot \beta_2 - ... - \alpha_n \cdot \beta_{n-1} - \beta_n \\ &= a_1 \prod_{i=2}^{n} \alpha_i - \sum_{i=2}^{n} \beta_i \prod_{j=i}^{n} \alpha_i \end{aligned}$$

comme $a_1 = 1$ et

$$\prod_{i=a}^{b} \alpha_i = \frac{T_a}{T_{a-1}} \cdot \frac{T_{a+1}}{T_a} ... \frac{T_{b-1}}{T_{b-2}} \cdot \frac{T_b}{T_{b-1}} = \frac{T_b}{T_{a-1}}$$

alors

$$a_n = \frac{T_n}{T_1} - \sum_{i=2}^{n} \frac{T_n}{T_{i-1}} \beta_i$$

$$a_n = \frac{T_n}{T_1} - \sum_{i=2}^{n} \frac{T_n}{T_{i-1}} \left\lceil \frac{m_i}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{C_i} \right\rfloor} \right\rceil$$

Comme $a_n \geq 0$, alors $\forall i = 1, ..., n-1,\ a_i \geq 0$ et donc Γ_n est ordonnançable.

■

Certaines tâches ont des périodes identiques mais des WCETs différents

Dans le cas général des tâches harmoniques où certaines tâches peuvent avoir les mêmes périodes mais différentes durées d'exécution, le corollaire suivant donne une condition d'ordonnançabilité suffisante.

Corollaire 3.1 *Un ensemble de tâches harmoniques* $\Gamma_n = \{\tau_{i,j}(C_{i,j}, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ *où* $\forall i = 1, ..., (n-1),\ T_i < T_{i+1}$ *est ordonnançable* **si**

$$\frac{T_n}{T_1} - \sum_{i=2}^{n} \frac{T_n}{T_{i-1}} \left\lceil \frac{m_i}{\left\lfloor \frac{T_1 - m_1 \cdot C_1}{\overline{C_i}} \right\rfloor} \right\rceil \geq 0 \tag{3.5}$$

où

$$T_1 - m_1 \cdot C_1 > 0$$

et

$$\overline{C_i} = \max_{j=1,...,m_i} C_i$$

Preuve
Ce corollaire est une conséquence directe du théorème 3.3. Le pire cas d'ordonnancement des tâches $\tau_{i,j}(C_{i,j}, T_j)$ est lorsqu'on ordonnance les tâches $\tau_{i,j}(\overline{C_i}, T_j)$ avec $\overline{C_i} = \max_{j=1,...,m_i} C_i$. On se ramène alors au théorème 3.3 avec les tâches $\tau_{i,j}(\overline{C_i}, T_j)$ au lieu de $\tau_{i,j}(C_i, T_j)$. ∎

3.3.4 Algorithme d'ordonnancement

On a proposé l'algorithme d'ordonnancement 1. Les théorèmes 3.2, 3.3 et le corollaire 3.1 nous permettent seulement de vérifier si un ensemble de tâches est ordonnançable ou non. Afin d'ordonnancer un ensemble de tâches harmoniques, nous proposons l'algorithme suivant de type First-Fit qui correspond à un problème de remplissages de boîtes avec des objets de tailles différentes (bin-packing), où les boîtes sont les intervalles $[kT_1, (k+1)T_1],\ k \in \mathbb{N}$, et les objets sont les durées d'exécution des tâches. Cet algorithme ordonnance une tâche dans la première case libre de l'arbre d'ordonnancement.

Algorithme 1 Algorithme d'ordonnancement de tâches harmoniques

Soit $\Gamma_n = \{\tau_{i,j}(C_i, T_j),\ i = 1, ..., n,\ j = 1, ..., m_i\}$ l'ensemble de tâches harmoniques à ordonnancer.
Décomposer Γ en n sous ensembles de tâches ayant des périodes identiques : $\Gamma = \bigcup_{i=1}^{n} \Gamma_i$ où $\Gamma_i = \{\tau_{i,j}(C_{i,j}, T_{i,j}),\ j = 1, ..., m_i\ T_{i,j} = T_i\}$ et $T_{i+1} > T_i$ pour $i = 1, ..., n-1$.
Soit $\Gamma_{ns} = \emptyset$ l'ensemble de tâches non ordonnançables.
si $\sum_{j=1}^{m_1} C_{1,j} \leq T_1$ **alors**
    Ordonnancer les m_1 tâches $\tau_{1,j}$ dans la case initiale.
fin si
pour $i = 2$ à n **faire**
    Dupliquer $\frac{T_i}{T_{i-1}}$ fois les intervales $[0, T_{i-1}]$ pour obtenir l'interval $[0, T_i]$.
    pour $j = 1$ à m_i **faire**
        si il existe un temps creux de taille supérieure ou égale à $C_{i,j}$ **alors**
            ordonnancer $\tau_{i,j}$ dans ce temps creux.
        sinon
            déplacer $\tau_{i,j}$ dans Γ_{ns}.
        fin si
    fin pour
fin pour
si $\Gamma_{ns} = \emptyset$ **alors**
    Γ est ordonnançable.
sinon
    Γ est non ordonnançable, mais $\Gamma \setminus \Gamma_{ns}$ est ordonnançable.
fin si

3.4 Analyse d'ordonnançabilité de tâches non harmoniques

Nous rappelons qu'une tâche NPPS $\tau_i(C_i, T_i)$ est décrite par sa date de début d'exécution s_i, sa durée d'exécution C_i et sa période T_i (cf. chapitre 1 section 1.2.1.1).

3.4.1 Conditions d'ordonnançabilité pour deux tâches

Dans tout ce qui suit, on notera $g_{i,j}$ le $PGCD$ des périodes des deux tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$.

Le théorème suivant a été donné la première fois par Korst [KAL96].

Théorème 3.4 *Deux tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ sont ordonnançables si et seulement si*

$$C_1 \leq (s_2 - s_1)\ mod(g_{1,2}) \leq g_{1,2} - C_2 \tag{3.6}$$

Exemple

On considère les deux tâches $\tau_1(1, 8)$ et $\tau_1(2, 12)$ avec $s_1 = 0$ et $s_1 = 5$. $g_{1,2} = PGCD(8, 12) = 4$, $(s_2 - s_1)\ mod(g_{1,2}) = 5\ mod(4) = 1$ et $g_{1,2} - C_2 = 2$. La condition (3.6) est satisfaite donc les tâches τ_1 et τ_2 sont ordonnançables, i.e. ne se chevauchent pas, comme le montre la figure 3.6.

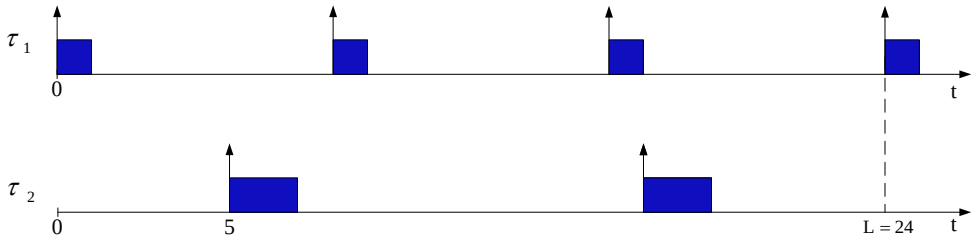


FIGURE 3.6 – Ordonnancement de deux tâches

Le corollaire suivant a aussi été donné par Korst [KAL96].

Corollaire 3.2 *Deux tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ sont ordonnançables si et seulement si*

$$C_1 + C_2 \leq g_{1,2} \tag{3.7}$$

Nous proposons les deux corollaires suivants qui donnent des conditions de non ordonnançabilités basées sur les conditions (3.6) et (3.7)

Corollaire 3.3 *Deux tâches $\tau_1(C_1, T_1, s_1)$ et $\tau_2(C_2, T_2, s_2)$ ne sont pas ordonnançables si et seulement si*

$$(s_2 - s_1)mod(g_{1,2}) \in [0\ ,\ C_1[\cup](g_{1,2} - C_2)\ ,\ g_{1,2}[\tag{3.8}$$

Preuve
Ce corollaire est une conséquence directe du théorème 3.4. Un système de tâche n'est pas ordonnançable si et seulement si la condition (3.6) n'est pas satisfaite, ce qui est équivalent à

$$\begin{cases} C_1 > (s_2 - s_1)\ mod(g_{1,2}) \\ ou \\ (s_2 - s_1)\ mod(g_{1,2}) > g_{1,2} - C_2 \end{cases} \tag{3.9}$$

Comme $0 \leq (s_2 - s_1)\ mod(g_{1,2}) < mod(g_{1,2})$, le système d'équations (3.9) devient

$$\begin{cases} 0 \leq (s_2 - s_1)\ mod(g_{1,2}) < C_1 \\ ou \\ g_{1,2} - C_2 < (s_2 - s_1)\ mod(g_{1,2}) < g_{1,2} \end{cases}$$

ou encore

$$(s_2 - s_1)mod(g_{1,2}) \in [0\ ,\ C_1[\cup](g_{1,2} - C_2)\ ,\ g_{1,2}[$$

■

Exemple
Considérons le système de tâches de l'exemple précédent et changeons la date de début d'exécution de la tâche τ_2 avec $s_2 = 3$. On a :

$$(s_2 - s_1)mod(g_{1,2}) = 3mod(4) = 3$$

et

$$[0\ ,\ C_1[\cup](g_{1,2} - C_2)\ , g_{1,2}[= [0\ ,\ 1[\cup](4 - 2)\ ,\ 4[$$

$$= [0\ ,\ 1[\cup]2\ ,\ 4[\ .$$

La condition (3.6) n'est pas satisfaite donc les tâches τ_1 et τ_2 ne sont pas ordonnançables. Comme on peut le voir sur la figure 3.7, la troisième instance τ_1^3 de la tâche τ_1 commence son exécution avant que la deuxième instance τ_2^2 de la tâche τ_2 n'ait terminé son exécution, et par conséquent ces deux instances se chevauchent.

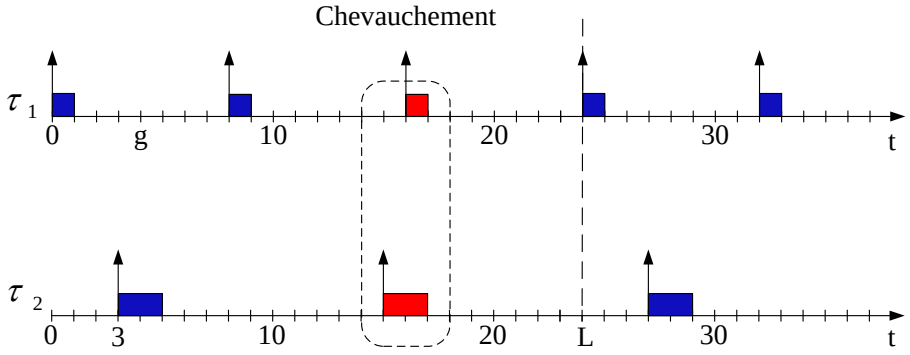


FIGURE 3.7 – Chevauchement de deux instances

Le corollaire suivant est un cas particulier du corollaire 3.3 dans lequel deux tâches ayant des périodes premières entre elles ne sont pas ordonnançables.

Corollaire 3.4 *Deux tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ ne sont pas ordonnançables si*

$$g_{1,2} = 1 \tag{3.10}$$

Preuve
D'après le corollaire 3.3, les deux tâches τ_1 et τ_2 sont ordonnançables si et seulement si $C_1 + C_1 \leq g_{1,2}$. Comme $g_{i_j} = 1$ et $C_i \geq 1$, $i = 1, ..., 2$, alors $C_1 + C_1 \geq 2$ et donc la condition (3.8) n'est pas satisfaite et par conséquent les deux tâches τ_1 et τ_2 ne sont pas ordonnançables. ∎

3.4.2 Conditions d'ordonnançabilité pour plus de deux tâches

Contrairement à l'analyse d'ordonnançabilité de deux tâches où nous avons présenté deux conditions d'ordonnançabilité nécessaires et suffisantes, toutes les conditions d'ordonnançabilité de plus de deux tâches sont des conditions suffisantes. Nous rappelons que l'analyse d'ordonnançabilité de plus de deux tâches NPPS est un problème NP-difficile. En effet pour savoir si un ensemble de n tâches est ordonnançable ou pas, il faut essayer tous les ordonnancements possibles avec des dates de début d'exécution allant de 0 à l'hyper-périodes L de l'ensemble des tâches, ce qui donne L^n scénarios de démarrage possibles. Pour chacun de ces scénarios il faut appliquer la condition nécessaire et suffisante (3.6) à tous les couples de tâches, ce qui fera $\frac{n \cdot (n-1)}{2}$ tests par scénario. Le nombre total de test sera égal à $\frac{n \cdot (n-1)}{2} \cdot L^n$. Ces tests sont très couteux c'est pourquoi nous

nous intéressons à des conditions d'ordonnançabilité suffisantes qui sont moins coûteuses.

O. Kermia a proposé une condition suffisante qui est la généralisation de la condition (3.8), mais cette condition est très restrictive dans la mesure où elle donne un faible taux de succès d'ordonnancement. Nous avons proposé une condition d'ordonnaçabilité moins restrictive qui permet d'ordonnancer des tâches non ordonnançables avec la condition proposée par O. Kermia.

Définition 3.1 *(**Ordonnancement sans temps creux**) un ensemble de tâches* $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ *est ordonnancé sans temps creux si les dates de début d'exécution des tâches satisfont la condition suivante*

$$s_i = \sum_{k=1}^{i-1} C_k \tag{3.11}$$

Le théorème suivant a été proposé par O. Kermia [KS08]

Théorème 3.5 *Soit* $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ *un ensemble de tâches à ordonnancer sans temps creux.* Γ_n *est ordonnançable si*

$$\sum_{i=1}^{n} C_i \leq g \tag{3.12}$$

Les dates de début d'exécution des tâches τ_i *sont données par*

$$s_i = \sum_{k=1}^{i-1} C_k + lg \tag{3.13}$$

avec $g = PGCD(T_i, i = 1, ..., n)$ *et* $l \in \mathbb{N}$.

Exemple
Considérons quatre tâches à ordonnancer : $\tau_1(1, 6)$, $\tau_2(1, 8)$, $\tau_3(1, 12)$ et $\tau_4(1, 24)$. On a $g = PGCD(T_1, T_2, T_3, T_4) = 2$ et $\sum_{i=1}^{4} C_i = 4$. Ces quatre tâches ne satisfont donc pas la condition (3.12)

$$\sum_{i=1}^{4} C_i > g.$$

Cependant cet ensemble de tâches est ordonnançable comme le montre la figure 3.8.

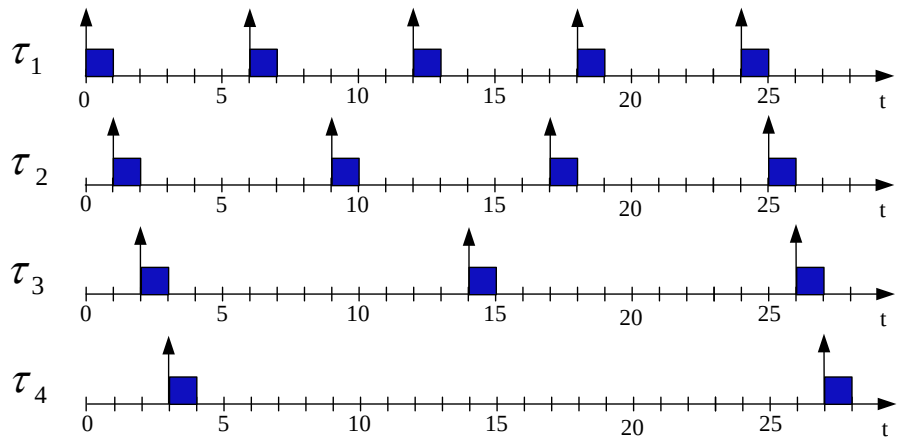


FIGURE 3.8 – Diagramme d'ordonnancement de quatre tâches

Remarque 3.1 *Les tâches qui satisfont deux à deux la condition du corollaire 3.3 ne sont pas forcément ordonnançables comme le montre le contre exemple suivant.*

Exemple
Considérons les trois tâches suivantes : $\tau_1(1,2)$, $\tau_1(1,2)$, $\tau_1(1,4)$ à ordonnancer sans temps creux. On a $\forall i,j = 1,...,2$, $C_i + C_j = 2$, $g_{i,j} = 2$. La condition du corollaire 3.3 est donc satisfaite mais ce système de tâches n'est pas ordonnançable. En effet le facteur d'utilisation U défini par $U = \sum_i \frac{C_i}{T_i}$ est supérieur à 1 :

$$U = \frac{1}{2} + \frac{1}{2} + \frac{1}{4} = \frac{5}{4} > 1.$$

Avant de présenter nos nouvelles conditions d'ordonnançabilité, nous proposons d'abord les trois lemmes suivants qui nous servirons dans les prochaines preuves.

Lemme 3.1 *Soit $\Gamma_n = \{\tau_i(C_i,T_i), i = 1,...,n\}$ un ensemble de tâches ordonnançables. L'ensemble Γ_n reste ordonnançable si l'on substitue k tâches $\tau'_j = (C'_j, T'_j)$ à n'importe quelle tâche $\tau_i(C_i,T_i) \in \Gamma_n$, avec $k \in \mathbb{N}^*$ et*

$$\begin{cases} C'_j = C_i \\ T'_j = kT_i \\ s'_j = s_i + (j-1)T_i,\ j = 1,...,k \end{cases}$$

Preuve
On distingue deux cas :

Premier Cas : $k = 1$ **:** dans ce cas la tâche τ'_1 est égale à la tâche τ_i, donc si l'on substitue la tâche τ'_1 à la tâche τ_i l'ensemble Γ_n reste ordonnançable.

Second Cas : $k > 1$ **:** l'ensemble des k tâches τ'_j a une hyper-période égale à $T'_j = kT_i$, on va donc s'intéresser à l'intervalle $I_k = [s'^0_1, s'^0_1 + kT_i]$.

La date de début d'exécution de chaque instance τ^l_i dans l'intervalle I_k est donnée par $s^l_i = s^0_i + lT_i$, avec $l < k$. Cela correspond à la date de début d'exécution s'_{l+1} de l'instance τ'_{l+1}. Donc pour chaque instance de la tâche τ_i il existe une instance d'une tâche τ'_j ayant la même date de début d'exécution et la même durée d'exécution, et donc on peut substituer k tâches $\tau'_j = (C'_j, T'_j)$ à n'importe quelle tâche $\tau_i(C_i, T_i) \in \Gamma_n$ et Γ_n restera ordonnançable. ∎

Lemme 3.2 *Soit* $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ *un ensemble de tâches ordonnançables. L'ensemble* Γ_n *reste ordonnançable si l'on substitue* k *tâches* $\tau'_j = (C'_j, T'_j)$ *à n'importe quelle tâche* $\tau_i(C_i, T_i) \in \Gamma_n$*, avec* $k \in \mathbb{N}^*$ *et*

$$\begin{cases} \sum_{j=1}^{k} C'_j \leq C_i \\ T'_j = T_i \\ s'_1 = s_i \; et \; s'_j = s_i + \sum_{l=1}^{j-1} C'_l \; j = 2, ..., k \end{cases}$$

Preuve
La démonstration de ce lemme est triviale. Chaque tâche $\tau_i(C_i, T_i)$ de l'ensemble Γ_n peut être remplacée par un ensemble de tâches ayant la même période T_i et dont la somme de leurs durées d'exécution ne dépasse pas C_i. Il suffit juste de les ordonnancer sans temps creux avec comme première date de début d'exécution la date de début d'exécution s_i de la tâche τ_i. ∎

Lemme 3.3 *Soit* $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ *un ensemble de tâches ordonnançables.* Γ_n *reste ordonnançable si on augmente la date de début d'exécution de n'importe quelle tâche* $\tau_i \in \Gamma_n$ *par un nombre entier de périodes* T_i *:* $s'_i = s_i + kT_i, \; k \in \mathbb{N}$.

Preuve
Comme on traite les tâches NPPS, une tâche reste toujours ordonnançable si on retarde sa date de début d'exécution d'une période T_i. ∎

Conformément à la stratégie présentée dans la section 3.2, nous considérons dans la suite un ensemble de tâches $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ qui satisfait la

condition restrictive (3.12), et une tâche candidate à l'ordonnancement telle que l'ensemble $\Gamma_n \cup \{\tau_c\}$ ne satisfait pas la condition (3.12). Nous distinguons deux cas : (i) la période de la tâche candidate est multiple de g et (ii) la période de la tâche candidate n'est pas multiple de g, où g est le $PGCD$ des périodes des tâches de Γ_n. Nous commençons par présenter le premier cas.

3.4.2.1 Tâche candidate à période multiple du PGCD

Nous proposons dans cette section deux théorèmes permettant d'exploiter les temps creux laissés par une tâche ordonnancée, ensuite nous présentons une combinaison de ces deux théorèmes permettant d'exploiter les temps creux de plusieurs tâches ordonnancées sans temps creux afin d'ordonnancer une tâche candidate.

Théorème 3.6 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ un ensemble de tâches qui satisfont la condition (3.12). Soit τ_c une tâche à ordonnancer telle que $\Gamma_n \cup \{\tau_c\}$ ne satisfait pas la condition (3.12). τ_c est ordonnançabe si : $\exists \tau_i(C_i, T_i) \in \Gamma_n$ avec $T_i > g$ telle que*

$$C_c \leq C_i \cdot \delta\left(T_c mod(T_i)\right) \tag{3.14}$$

Les dates de début d'exécution possibles de la tâche τ_c sont données par :

$$s_c = s_i + lg + kT_i + \alpha \tag{3.15}$$

avec $k \in \mathbb{N}$, $1 \leq l \leq (\frac{T_i}{g} - 1)$, $0 \leq \alpha \leq (C_i - C_c)$, mod est la fonction $modulo$ et δ est le symbole de Kronecker :

$$\forall x \in R,\ \delta(x) = \begin{cases} 1 \ \ si \ \ x = 0 \\ 0 \ \ sinon \end{cases}$$

Preuve
La condition (3.14) est équivalente à : $C_c \leq C_i$ et T_c est multiple de T_i.
Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ un ensemble de tâches qui satisfont la condition (3.12).
Soit la tâche $\tau_i(s_i, C_i, T_i) \in \Gamma_n$ avec $T_i > g$ et la tâche candidate τ_c qui satisfont la condition (3.14).

Considérons la tâche $\tau'_i = (s_i, C_i, g)$ qui a la même date de début d'exécution et la même durée d'exécution de que la tâche τ_i, mais une période égale au $PGCD$ des tâches de Γ_n.

Soit l'ensemble de tâches Γ'_n où l'on substitue la tâche τ'_i à la tâche τ_i :

$$\Gamma'_n = \{\tau_1, ..., \tau_{i-1}, \tau'_i, \tau_{i+1}, ..., \tau_n\}$$

Les tâches des ensembles Γ_n et Γ'_n ont les mêmes $PGCD$ de leurs périodes et la même somme des durées d'exécutions. Comme Γ_n satisfait la condition (3.12) alors Γ'_n satisfait aussi est ordonnançable. Γ'_n est donc ordonnançable.

D'après le lemme 3.1 et comme T_i est multiple de g, si l'on substitue les deux tâches $\tau_i(C_i, T_i)$ et $\tau''_i = (C_i, T_i)$, avec $s''_i = s_i + lg$ et $l = 1, ..., (\frac{T_i}{g} - 1)$, à la tâche $\tau'_i = (C_i, g)$, l'ensemble Γ'_n reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma''_n = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau''_i, \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.3, si l'on substitue la tâche $\tau'''_i = (C_i, T_i)$, avec $s'''_i = s_i + lg + kT_i$ et $k \in \mathbb{N}$, à la tâche $\tau''_i = (C_i, T_i)$, l'ensemble Γ''_n reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma'''_n = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau'''_i, \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.1 et comme T_c est multiple de T_i, si l'on substitue la tâche $\tau''''_i = (C_i, T_c)$ avec $s''''_i = s_i + lg + kT_i$, à la tâche $\tau'''_i = (C_i, T_i)$, l'ensemble Γ''_n reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma''''_n = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau''''_i, \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.2, si l'on substitue la tâche $\tau_c(C_c, T_c)$ à la tâche $\tau''''_i = (C_i, T_c)$, l'ensemble Γ'''_n reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma^*_n = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau_c, \tau_{i+1}, ..., \tau_n\} = \Gamma_n \cup \{\tau_c\}.$$

Finalement si $C_c = C_i$ alors on ne peut ordonnancer τ_c qu'à la date $s_c = s_i + lg + kT_i$ mais si $C_c < C_i$ alors on peut ordonnancer τ_c au plus tard à $s_c = s_i + lg + kT_i + (C_i - C_c)$.

La tâche $\tau_c(C_c, T_c)$ est donc ordonnançable, avec $s_c = s_i + lg + kT_i$ et $l = 1, ..., (\frac{T_i}{g} - 1)$ et $k \in \mathbb{N}$. ∎

Théorème 3.7 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ un ensemble de tâches qui satisfont la condition (3.12). Soit τ_c une tâche à ordonnancer telle que $\Gamma_n \cup \{\tau_c\}$ ne satisfait pas la condition (3.12). τ_c est ordonnançable si : $\exists \tau_i(C_i, T_i) \in \Gamma_n$ telle que*

$$C_c \leq C_i \cdot \delta \left(T_c \cdot mod(2g) + T_i \cdot mod(2g)\right) \tag{3.16}$$

Les dates de début d'exécution de la tâche τ_c sont données par :

$$s_c = s_i + (2k + 1)g + \alpha \tag{3.17}$$

avec $k \in \mathbb{N}$ et $0 \leq \alpha \leq (C_i - C_c)$.

Preuve
La condition (3.16) est équivalente à : $C_c \leq C_i$ et T_c et T_i sont multiples de $2g$.
Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ un ensemble de tâches qui satisfont la condition (3.12).
Soit la tâche $\tau_i(C_i, T_i) \in \Gamma_n$ et la tâche candidate τ_c qui satisfont la condition (3.16).

Considérons la tâche $\tau_i' = (C_i, g)$ avec $s_i' = s_i$, qui a la même date de début d'exécution et la même durée d'exécution de que la tâche τ_i, mais une période égale au $PGCD$ des tâches de Γ_n.

Soit l'ensemble de tâches Γ_n' où l'on remplace la tâche τ_i par la tâche τ_i' :

$$\Gamma_n' = \{\tau_1, ..., \tau_{i-1}, \tau_i', \tau_{i+1}, ..., \tau_n\}$$

Les tâches des ensembles Γ_n et Γ_n' ont les même $PGCD$ de leurs périodes et la même somme des durées d'exécutions. Comme Γ_n satisfait la condition (3.12) alors Γ_n' satisfait aussi est ordonnançable. Γ_n' est donc ordonnançable.

D'après le lemme 3.1, si on remplace la tâche $\tau_i' = (C_i, g)$ par les deux tâches $\tau_i'' = (C_i, 2g)$ et $\tau_i''' = (C_i, 2g)$ avec $s_i'' = s_i$ et $s_i''' = s_i + g$, l'ensemble Γ_n' reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma_n'' = \{\tau_1, ..., \tau_{i-1}, \tau_i'', \tau_i''', \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.1 et comme T_i est multiple de $2g$, si on remplace la tâche $\tau_i'' = (C_i, 2g)$ par la tâche $\tau_i(C_i, T_i)$, l'ensemble Γ_n'' reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma_n''' = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau_i''', \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.3, si on remplace la tâche $\tau_i''' = (C_i, 2g)$ par la tâche $\tau_i'''' = (C_i, 2g)$, avec $s_i'''' = s_i + (2k+1)g$ et $k \in \mathbb{N}$, l'ensemble Γ_n''' reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma_n'''' = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau_i'''', \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.1 et comme T_c est multiple de $2g$, si on remplace la tâche $\tau_i'''' = (C_i, 2g)$ par la tâche $\tau_i''''' = (C_i, T_c)$, avec $s_i''''' = s_i''''$ et $k \in \mathbb{N}$, l'ensemble Γ_n'''' reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma_n''''' = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau_i''''', \tau_{i+1}, ..., \tau_n\}.$$

D'après le lemme 3.2 et comme $C_c \leq C_i$, si on remplace la tâche $\tau_i''''' = (C_i, T_c)$ par la tâche $\tau_c(C_c, T_c)$, avec avec $s_c = s_i'''''$ et $k \in \mathbb{N}$, l'ensemble Γ_n''''' reste ordonnançable. On obtient donc l'ensemble ordonnançable :

$$\Gamma_n^* = \{\tau_1, ..., \tau_{i-1}, \tau_i, \tau_c, \tau_{i+1}, ..., \tau_n\}.$$

Finalement si $C_c = C_i$ alors on ne peut ordonnancer τ_c qu'à la date $s_c = s_i + (2k+1)g$ mais si $C_c < C_i$ alors on peut ordonnancer τ_c au plus tard à $s_c = s_i + (2k+1)g + (C_i - C_c)$.

La tâche $\tau_c(C_c, T_c)$ est donc ordonnançable, avec $k \in \mathbb{N}$. ∎

Le théorème suivant est une combinaison des théorèmes 3.6 et 3.7.

Théorème 3.8 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ un ensemble de tâches qui satisfont la condition (3.12). Soit τ_c une tâche à ordonnancer telle que $\Gamma_n \cup \{\tau_c\}$ ne satisfait pas la condition (3.12). τ_c est ordonnançable s'il existe m tâches $\{\tau_i, \tau_{i+1}, ..., \tau_{i+m-1}\} \subset \Gamma_n$, ordonnancées sans temps creux avec $T_k > g$ telle que*

$$C_c \leq \left(\sum_{k=i}^{i+m-1} C_k \right) \cdot \delta \left(\sum_{k=i}^{i+m-1} T_c mod(T_k) \cdot (T_c \cdot mod(2g) + T_k \cdot mod(2g)) \right) \tag{3.18}$$

Les dates de début d'exécution de la tâche τ_c sont données par :

$$s_c = s_i + g + kL + \alpha \tag{3.19}$$

avec $k \in \mathbb{N}$, $L = PPCM(T_i, ..., T_{i+m-1})$ et

$$0 \leq \alpha \leq \left(\sum_{k=i}^{i+m-1} C_k \right) - C_i.$$

Preuve

Ce théorème est une combinaison des théorèmes 3.6 et 3.7.

La condition (3.18) est équivalente à : $C_c \leq \sum_{k=i}^{i+m-1} C_k$ et

$$\sum_{k=i}^{i+m-1} T_c mod(T_k) \cdot (T_c \cdot mod(2g) + T_k \cdot mod(2g)) = 0$$

ou encore

$$\forall k = 1, ..., (i+m-1),\ T_c mod(T_k) = 0 \ \ ou \ \ T_c \cdot mod(2g) + T_k \cdot mod(2g) = 0$$

ce qui revient à dire que chacune des m tâches τ_i satisfait soit la condition (3.14) soit la condition (3.16).

Si on découpe la tâche τ_c en m tâches $\tau_{c,k} = (C_{c,k}, T_c)$ avec $s_i = s_k + g$ et telles que $\sum_k C_{c_k} = C_c$, alors d'après les conditions (3.14) et (3.16), les m tâches $\tau_{c,k}$ sont ordonnançables.

Comme l'ordonnancement des m tâches τ_i a une hyper-période $L = PPCM(t_I, ..., t_{I+M-1})$, alors τ_c peut être ordonnancée à $s_c = s_i + g + kL$ avec $k \in \mathbb{N}$.

Finalement si $C_c = \sum_{k=i}^{i+m-1} C_k$ alors on ne peut ordonnancer τ_c qu'à la date $s_c = s_i + g + kL$ mais si $C_c < \sum_{k=i}^{i+m-1} C_k$ alors on peut ordonnancer τ_c au plus tard à $s_c = s_i + g + kL + \sum_{k=i}^{i+m-1} C_k$.

■

Attention, le théorème 3.8 n'est valable que si les m tâches $\{\tau_i, \tau_{i+1}, ..., \tau_{i+m-1}\} \subset \Gamma_n$ satisfaisant la condition (3.18) s'exécutent sans temps creux, sinon ce théorème n'est plus valable.

3.4.2.2 Tâche candidate à période non multiple du PGCD

Dans cette section nous considérons que la période de la tâche candidate τ_c n'est pas multiple de g. Nous proposons d'abord un théorème permettant d'ordonnancer une tâche candidate dans les temps creux laissés par deux tâches seulement déjà ordonnancées, de telle manière que les instances de τ_c soient ordonnancées périodiquement dans les temps creux laissés par la première tâche et les l'instances suivantes soient ordonnancées dans les temps creux laissés par la seconde tâche. Ensuite nous donnons une extension de ce théorème dans le cas où l'on a plus de deux tâches déjà ordonnancées.

Théorème 3.9 *Soit $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ deux tâches qui satisfont la condition (3.12) avec $T_1 > g_{1,2}$ et $T_2 > g_{1,2}$. Soit $\tau_c(C_c, T_c)$ une tâche à ordonnancer tel que l'ensemble de tâches $\{\tau_1, \tau_2, \tau_c\}$ ne satisfait pas la condition (3.12). τ_c est ordonnançabe si*

$$C_c \leq min(C_1, C_2) \cdot \delta\left((2T_c)\ mod(L)\right) \tag{3.20}$$

avec $L = PPCM(T_1, T_2)$.

Preuve

Soit deux tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ deux tâches qui satisfont la condition (3.12) avec $T_1 > g_{1,2}$ et $T_2 > g_{1,2}$. Soit $\tau_c(C_c, T_c)$ une tâche candidate qui satisfait la condition (3.20). La condition (3.20) signifie que $C_c \leq min(C_1, C_2)$ et $2T_c$ est multiple du $PPCM(T_1, T_2)$ ou encore $2T_c$ est multiple de T_1 et de T_2. Sans perte de généralité considérons que $C_2 \geq C_1$ et que les dates de démarrage sont choisis comme suit : $s_1 = 0$, $s_2 = T_c$ et $s_3 = g_{1,2}$. Les tâches $\tau_1(C_1, T_1)$ et $\tau_2(C_2, T_2)$ sont supposées satisfaire la condition (3.12) donc $C_1 + C_2 \leq g_{1,2}$.

D'après le lemme 3.1 les tâches τ_1, τ_2 et τ_c sont ordonnançables si les tâches τ_1, τ_2, τ'_c et τ''_c sont ordonnançables avec $\tau'_c(C_c, 2T_c)$ et $\tau''_c(C_c, 2T_c)$ et $s'_c = s_c + T_c$ et $s''_c = s_c + T_c$.

On a

$$C_c \leq min(C_1, C_2) \leq C_1$$

et

$$\delta((2T_c)mod(T_1)) = \delta(0) = 1$$

avec

$$s_c = s_1 + g$$

donc les tâches $\tau_1(C_1, T_1)$ et $\tau'_c(C_c, 2T_c)$ satisfont la condition d'ordonnançabilité (3.14) et sont ainsi ordonnançables.

De la même manière on a

$$C_c \leq min(C_1, C_2) \leq C_2$$

et

$$\delta((2T_c)mod(T_2)) = \delta(0) = 1$$

avec

$$s_c = s_2 + g$$

donc les tâches $\tau_2(C_2, T_2)$ et $\tau'_c(C_c, 2T_c)$ satisfont la condition d'ordonnançabilité (3.14) et sont ainsi ordonnançables.

D'après le lemme 3.1 les tâches τ_1, τ_2 et τ_3 sont ordonnançables. ∎

Exemple
Considérons trois tâches $\tau_1(1, 6)$, $\tau_2(1, 10)$ et $\tau_3(1, 15)$. Les tâches τ_1 et τ_2 satisfont la condition (3.12), ces deux tâches sont donc ordonnançables avec $s_1 = 0$ et $s_2 = 1$. On a $L = PPCM(6, 10, 15) = 30$. Comme

$$min(C_1, C_2) \cdot \delta\left((2T_3)\ mod(L)\right) = min(1, 1) \cdot \delta(30mod(30)) = 1$$

et $C_3 = 1$ alors la tâche τ_3 satisfait la condition (3.20), elle est donc ordonnançable avec $s_3 = 2$. La figure 3.9 montre l'ordonnancement de ces trois tâches sur l'hyper-période de longueur $L = 30$.

Le théorème suivant donne une extension du théorème (3.9) pour plus de deux tâches.

Théorème 3.10 *Soit l'ensemble de tâches $\Gamma_n = \{\tau_i(C_i, T_i),\ i = 1, ..., n$ qui satisfait la condition (3.12) avec $T_i > g$, $i = 1, ..., n$. Soit $\tau_c(C_c, T_c)$ une tâche à ordonnancer tel que l'ensemble de tâches $\Gamma_n \cup \{\tau_c\}$ ne satisfait pas la condition (3.12). τ_c est ordonnançabe si*

$$C_c \leq min_{i=1,...,n}(C_i) \cdot \delta((nT_c)\ mod(L)) \tag{3.21}$$

avec $L = PPCM(T_i,\ i = 1, ..., n)$.

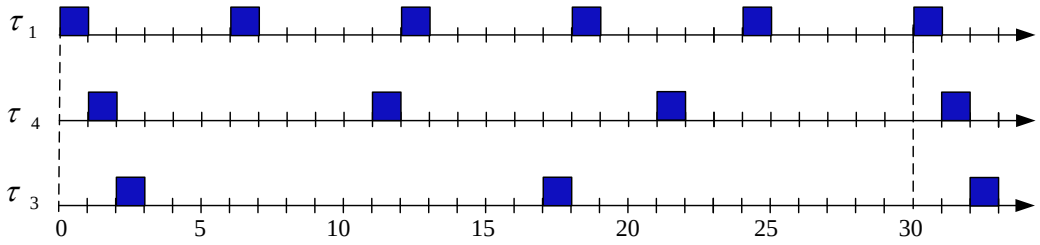


FIGURE 3.9 – Ordonnancement des tâches τ_1, τ_2 et τ_3

Preuve
Considérons l'ensemble de tâches $\Gamma_n = \{\tau_i(C_i, T_i),\ i = 1, ..., n$ qui satisfait la condition (3.12) avec $T_i > g$, $i = 1, ..., n$ et la tâche τ_c qui satisfait la condition (3.21). D'après le lemme 3.1 l'ensemble $\Gamma_n \cup \{\tau_c\}$ reste ordonnançable si on l'on substitue n tâches $\tau_c^i(C_c, nT_c)$ à la tâche τ_c, avec $i = 1, ..., n$.

Pour chaque tâche $\tau_i \in \Gamma$ et τ_c^i on a :

$$C_c \leq min_{i=1,...,n}(C_i) \leq C_i$$

et comme $(nT_c)\ mod(L) = 0$ alors

$$(nT_c)mod(T_i) = 0.$$

on a aussi

$$s_c^i = s_i + g$$

donc les tâches τ_i et τ_c^i satisfont la condition (3.14) et elles sont ainsi ordonnançables.

Comme chaque tâche $\tau_i \in \Gamma_n$ et chaque tâche τ_c^i satisfont la condition (3.14) alors l'ensemble $\Gamma \cup \{\tau_c^i,\ i = 1, ..., n\}$ est ordonnançable. D'après le lemme 3.1 l'ensemble $\Gamma \cup \{\tau_c\}$ est ordonnançable. ∎

Exemple
Soit l'ensemble de tâches $\Gamma_3 = \{\tau_1(1, 6), \tau_2(1, 12), \tau_3(1, 15)\}$. Cet ensemble satisfait la condition (3.12) : $(C_1 + C_2 + C_3 = 3) \leq (g = 3)$. Ces tâches sont donc ordonnançables avec $s_1 = 0$, $s_2 = 1$ et $s_3 = 2$. Soit la tâche $\tau_c(1, 20)$ qui ne satisfait pas la condition (3.12) avec les tâches de Γ_3 :

$$(C_1 + C_2 + C_3 + C_c = 4) \not\leq (g = 1).$$

On a $L = PPCM(6, 12, 15) = 60$. Comme

$$min_{i=1,...,3}(C_i) \cdot \delta((3T_c)\ mod(L)) = 1 \cdot \delta(60\ mod(60)) = 1$$

alors la tâche τ_c satisfait la condition (3.21) et elle est ainsi ordonnançable avec $s_c = 3$, comme le montre la figure 3.10.

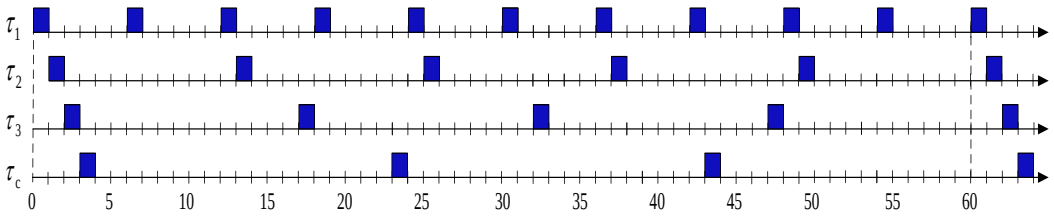


FIGURE 3.10 – Ordonnancement des tâches τ_1, τ_2, τ_3 et τ_4

3.4.3 Tri des tâches selon les multiplicités de leurs périodes

Dans la littérature des algorithmes d'ordonnancement statiques le tri des tâches, selon l'ordre croissant de leurs périodes, est largement répandu sans que des raisons objectives n'aient été données.

Comme une tâche a plus de chance de satisfaire la condition (3.18) si sa période est multiple de la période d'une autre tâche nous définissons alors la priorité p_i d'une tâche τ_i par le nombre de tâches ayant des périodes divisibles de la tâche en question. La priorité p_i est donc donnée par :

$$p_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \delta(T_i \, mod(T_j)) \tag{3.22}$$

Les tâches sont alors triées selon l'ordre croissant de leurs priorités p_i, et les tâches ayant des priorités égales sont triées dans n'importe quel ordre. L'exemple suivant illustre l'utilité de cet ordre.

Exemple
Considérons l'ensemble de tâches $\Gamma_3\{\tau_1(1,6), \tau_2(1,12), \tau_3(1,14)\}$ trié selon l'ordre croissant des périodes.

Les tâches τ_1 et τ_2 satisfont la condition (3.12) donc elles sont ordonnançables : $C_1 + C_2 = 2 \leq PGCD(T_1, T_2) = 6$. Cependant la tâche τ_3 ne satisfait pas la condition (3.18).

Selon l'équation (3.22) on a les priorités suivantes : $p_1 = 0$, $p_2 = 1$ et $p_3 = 0$. Les deux tâches τ_1 et τ_3 ont alors les plus hautes priorités et la tâche τ_2 a la plus faible priorité. Comme les tâches τ_1 et τ_3 ont des priorités égales alors on peut avoir deux tris possibles :

1. τ_1, τ_3, τ_2,
2. τ_3, τ_1, τ_2.

Pour les deux tris, les deux premières tâches τ_1 et τ_3 satisfont la condition (3.12) donc elles sont ordonnançables : $C_1 + C_3 = 2 \leq PGCD(T_1, T_2) = 2$. Les tâches τ_2 et τ_1 satisfont la condition (3.18) et elles sont donc ordonnançables.

3.4.4 Algorithme d'ordonnancement

On a proposé l'algorithme d'ordonnancement 2. Cet algorithme commence par trier les tâches selon la multiplicité de leurs périodes. Ensuite il initialise un ensemble vide Γ_{s1} (resp. Γ_{s2}) qui va contenir les tâches satisfaisant la condition restrictive (3.12) (resp. la condition (3.18) ou 3.21).

L'algorithme déplace d'abord les tâches de Γ qui satisfont la condition (3.12) vers l'ensemble Γ_{s1}, ensuite il déplace les tâches restantes dans Γ qui satisfont la condition (3.18) vers l'ensemble Γ_{s2}. S'il ne reste aucune tâche dans Γ alors toutes les tâches sont ordonnançables, sinon les tâches de $\Gamma_{s1} \cup \Gamma_{s2}$ sont ordonnançables et les tâches de Γ ne sont pas ordonnançables.

On a aussi proposé l'algorithme optimal 3 qui peut ordonnancer aussi bien des tâches harmoniques que des tâches non harmoniques. Pour un ensemble de tâches $\Gamma = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ à ordonnancer, cet algorithme parcourt les dates de début d'exécution s_i de toutes les tâches comprises dans une hyper-période de longueur $PPCM(T_i)$, et vérifie pour tous les couples de tâches si la condition d'ordonnançabilité nécessaire et suffisante (3.6) est satisfaite. Dès que cette dernière condition est satisfaite par tout les couples de tâches l'algorithme s'arrête en concluant que Γ est ordonnançable, sinon que Γ n'est pas ordonnançable.

L'exemple suivant illustre l'algorithme 2.

Exemple
Considérons quatre tâches à ordonnancer :
$\Gamma_4 = \{\tau_1(1, 12),\ \tau_2(3, 16),\ \tau_3(1, 24\},\ \tau_4(1, 40)\}$, $g = PGCD(12, 16) = 4$ et $C_1 + C_2 = 4$.

La condition (3.10) donne : $C_1 + C_2 = PGCD(T_1, T_2)$, alors τ_1 et τ_2 sont ordonnançables avec $s_1 = 0$ et $s_2 = 1$.

Cette condition ne donne aucune information sur l'ordonnançabilité des tâches de Γ.

L'ensemble des tâches Γ_{s1} est donc initialisé par

$$\Gamma_{s1} = \{\tau_1(1, 12),\ \tau_2(3, 16)\}.$$

L'ensemble Γ devient : $\Gamma = \{\tau_3(1, 24\},\ \tau_4(1, 40)\}$.

Appliquons la condition (3.18) sur les tâches de Γ.

Algorithme 2 Algorithme d'ordonnancement de tâches non harmoniques

Soit $\Gamma = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ l'ensemble de tâches à ordonnancer
Trier les tâches selon les multiplicités de leurs périodes
Initialiser par l'ensemble vide, l'ensemble Γ_{s1} qui va contenir les tâches satisfaisant la condition restrictive (3.12)
Initialiser par l'ensemble vide, l'ensemble Γ_{s2} qui va contenir les tâches ordonnançable selon la condition (3.18)
On suppose que $s_1 = 0$
$\Gamma_{s1} = \{\tau_1\}$ et $\Gamma := \Gamma \backslash \{\tau_1\}$
pour chaque τ_i de Γ **faire**
    si la condition (3.12) est satisfaite pour $\Gamma_{s1} \cup \{\tau_i\}$ **alors**
        calculer s_c selon la condition (3.13)
        déplacer la tâche τ_k de Γ vers Γ_{s1}
    sinon
        passer à la tâche suivante de Γ
    fin si
fin pour
pour chaque τ_c de Γ **faire**
    si il existe un sous-ensemble $\{\tau_i, ..., \tau_{i+m}\}$ de m tâches de Γ qui satisfait la condition (3.18) ou (3.21) avec τ_c **alors**
        déplacer ce sous-ensemble et la tâche τ_c vers Γ_{s2}
    sinon
        passer à la tâche suivante de Γ
    fin si
fin pour
si il ne reste aucune tâche dans Γ **alors**
    toutes les tâches sont ordonnançables
sinon
    les tâches de $\Gamma_{s1} \cup \Gamma_{s2}$ sont ordonnançables
    les tâches de Γ ne sont pas ordonnançables
fin si

Pour $\tau_c = \tau_3(1, 24)$, la condition (3.18) est satisfaite par les tâches τ_c et τ_1 :

$$1 \leq 1 \cdot \delta[(24 mod(12)) \cdot (24 mod(8) + 12 mod(8))]$$

$$1 \leq 1 \cdot \delta[0 \cdot 8] = 1$$

Alors la tâche τ_3 est ordonnançable. Selon la condition (3.19) sa date de début d'exécution est donnée par $s_3 = s_1 + g = 4$. donc

$$\Gamma_{s1} = \{\tau_2(3, 16)\}$$

Algorithme 3 Algorithme d'ordonnancement optimal

1: Soit $\Gamma = \{\tau_i(C_i, T_i), i = 1, ..., n\}$ l'ensemble de tâches à ordonnancer
2: Calculer l'hyper-période $L = PPCM(T_i, i = 1, ..., n)$
3: **pour chaque** date de début d'exécution $s_i \in [0, L]$ avec $i = 1, ..., n$ **faire**
4: **pour chaque** couple de tâches $(\tau_i, \tau_j) \in \Gamma^2$ **faire**
5: vérifier si la condition (3.6) est satisfaite
6: **fin pour**
7: **si** la condition (3.6) est satisfaite pour tout les couples **alors**
8: Γ est ordonnançable
9: **sinon**
10: Passer aux dates de début d'exécution suivantes
11: **fin si**
12: **fin pour**

et

$$\Gamma_{s2} = \{\tau_1(1, 12),\ \tau_3(1, 24)\}$$

et

$$\Gamma = \{\tau_4(1, 40)\}$$

Pour $\tau_c = \tau_4(2, 40)$, la condition (3.18) est satisfaite par les tâches τ_c et τ_2 :

$$2 \leq 3 \cdot \delta[(40mod(16)) \cdot (40mod(8) + 16mod(8))]$$

$$1 \leq 1 \cdot \delta[8 \cdot 0] = 1$$

Alors la tâche τ_4 est ordonnançable. Selon la condition (3.19) sa date de début d'exécution est donnée par $s_4 = s_2 + g = 5$. donc

$$\Gamma_{s1} = \emptyset$$

et

$$\Gamma_{s2} = \{\tau_1(1, 12),\ \tau_3(1, 24),\ \tau_2(3, 16),\ \tau_4(1, 40)\}$$

et

$$\Gamma = \emptyset$$

Finalement les quatre tâches sont ordonnançables.

La figure 3.11 montre un *diagramme linéaire* et un *diagramme circulaire* [BLOS95, LBOS95] d'ordonnancement des tâches de l'ensemble Γ pendant une hyper-période égale au $PPCM(T_1, T_2, T_3, T_4) = 108$. Ce résultat est obtenu en utilisant l'outil SAS [YGSdR09]. Le diagramme circulaire est été introduit par P. Meumeu dans [Yom09] pour donner une représentation plus compacte de l'ordonnancement.

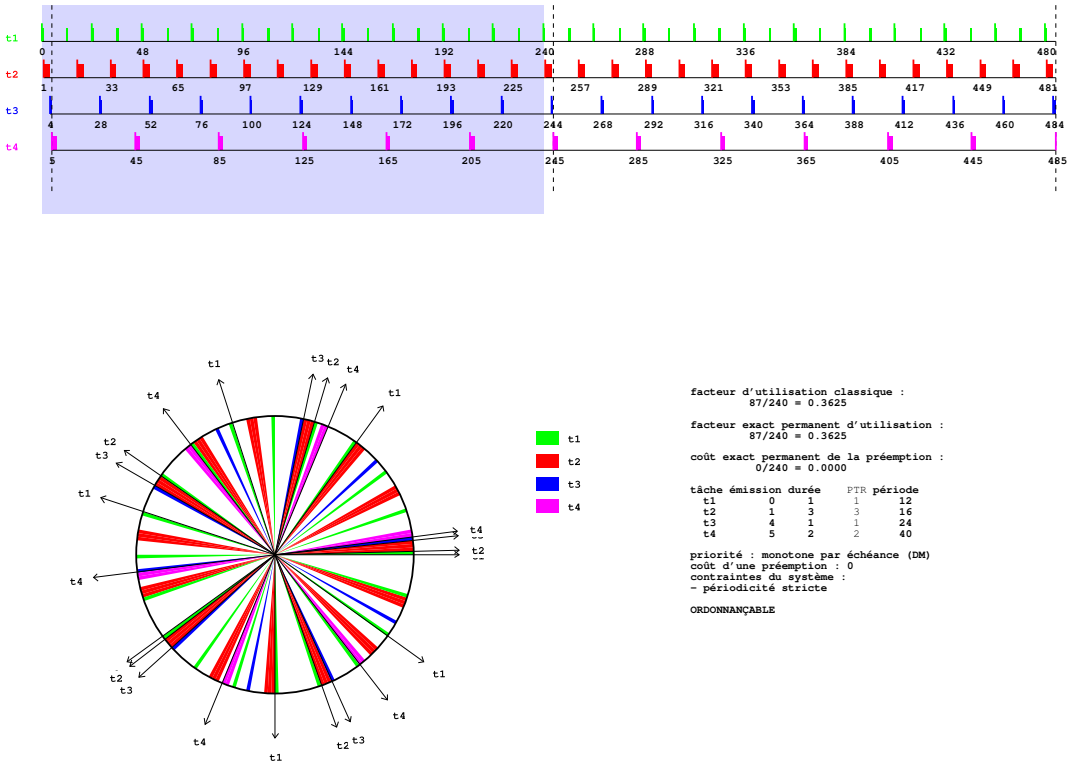


FIGURE 3.11 – Ordonnancement de quatre tâches NPPS

3.5 Phase transitoire et phase permanente

L'ordonnancement d'un ensemble de tâches NPPS $\Gamma_n = \{\tau_i(r_i, C_i, D_i, T_i), i = 1, ..., n\}$ est formé d'une phase transitoire unique $[0, \phi]$ et d'une phase permanente $[\phi, \phi + 2 \cdot L]$ qui se répète un nombre infini de fois, où ϕ est la longueur de la phase transitoire telle que $\phi \in [\underset{i=1,...,n}{max}(r_i), \underset{i=1,...,n}{max}(r_i) + L]$ où L est l'hyper-période donnée par $L = PPCM(T_i,\ i = 1, ..., n)$ [LM80, CGG04].

Dans le cas des tâches NPPS, l'hyper-période est toujours donnée par $L = PPCM(T_i,\ i = 1, ..., n)$ [KALW91, KAL96, CS03], cependant le calcul de la phase transitoire n'est pas le même que celui des tâches PP. Nous proposons le théorème suivant qui permet de calculer la phase transitoire que l'on notera $\Phi = [0, \phi]$ pour ce type de tâches.

Théorème 3.11 *Soit $\Gamma_n = \{\tau_i(C_i, T_i),\ i = 1, ..., n\}$ un ensemble de tâches NPPS de dates de début d'exécution s_i obtenues après ordonnancement. La phase tran-*

sitoire Φ, si elle existe, est donnée par $\Phi = [0, \phi]$ avec

$$\phi = \underset{i=1,\ldots,n}{Max}\ (0,\ s_i + C_i - T_i) \tag{3.23}$$

Preuve
Soit Γ_n l'ensemble de tâches ordonnancé avec une phase transitoire $\Phi = [0, \phi]$. ϕ est le plus petit entier naturel tel que la première phase permanente $[\phi,\ \phi + L]$ contient $(\frac{L}{T_i})$ instances de chaque tâche τ_i. Donc la $(\frac{L}{T_i} - 1)^{eme}$ instance $\tau_i^{(\frac{L}{T_i}-1)}$ de chaque tâche τ_i doit terminer son exécution avant la date $(\phi + L)$, ainsi

$$s_i + (\frac{L}{T_i} - 1)T_i + C_i \leq \phi + L$$

donc

$$\phi \geq s_i + C_i - T_i. \tag{3.24}$$

■

De plus pour une tâche τ_i, la date de début d'exécution s_i' de sa première instance dans la première phase permanente $[\phi, \phi + L]$ relativement à la date ϕ, est égale à sa date de début d'exécution s_i'' de sa première instance dans la deuxième phase permanente $[\phi + L, \phi + 2L]$ relativement à la date $(\phi + L)$.

On a

$$s_i' = s_i + \left\lceil \frac{\phi - s_i}{T_i} \right\rceil T_i - \phi.$$

et

$$s_i'' = s_i + \left\lceil \frac{(L + \phi) - s_i}{T_i} \right\rceil T_i - (L + \phi) = s_i + \left\lceil \frac{\phi - s_i}{T_i} \right\rceil T_i + L - L - \phi$$

$$= s_i + \left\lceil \frac{\phi - s_i}{T_i} \right\rceil T_i - \phi.$$

On a donc $s_i' = s_i''$, et comme ϕ est le plus petit entier qui satisfait la condition (3.24) alors on obtient

$$\phi = \underset{i=1,\ldots,n}{Max}(0, s_i + C_i - T_i).$$

Remarque 3.2 *On remarque que la phase transitoire d'un ordonnancement de tâches NPPS est inférieure ou égale à celle d'un ordonnancement de tâches NPPS. En effet comme $T_i - C_i > 0$ et $s_i \geq 0$ alors*

$$\underset{i=1,\ldots,n}{Max}\ (0, s_i - (T_i - C_i)) \leq \underset{i=1,\ldots,n}{Max}\ (0, s_i) \leq \underset{i=1,\ldots,n}{Max}\ (s_i) = \underset{i=1,\ldots,n}{Max}\ (r_i)$$

donc

$$\phi \leq \underset{i=1,\ldots,n}{Max}\ (r_i)$$

Exemple
Considérons l'ordonnancement de trois tâches $\tau_1(1,4)$, $\tau_2(1,6)$ et $\tau_3(1,12)$ de dates de début d'exécution $s_1 = 0$, $s_2 = 9$ et $s_3 = 4$ obtenues après ordonnancement. D'après le théorème 3.11 cet ordonnancement a une phase transitoire de longueur

$$\phi = \underset{i=1,\ldots,3}{Max}\,(0,\ s_i + C_i - T_i) = Max(0,-3,4,-12) = 4,$$

et une phase permanente de longueur $L = PPCM(T_1,T_2,T_3) = 12$ comme le montre la figure 3.12.

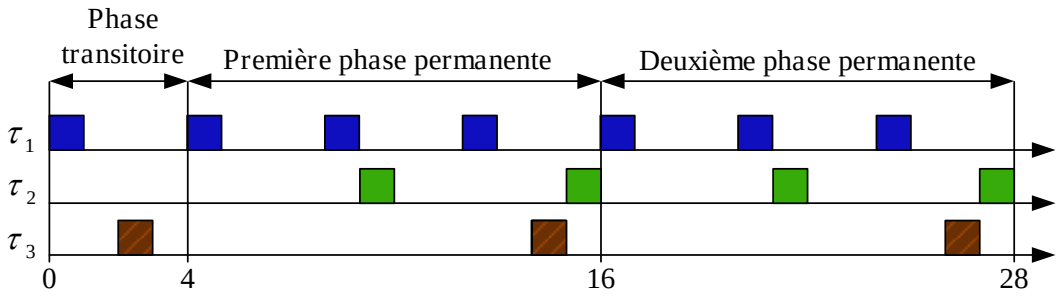


FIGURE 3.12 – Ordonnancement avec phase transitoire

3.6 Analyse de performances

Dans cette section nous présentons une analyse de performances pour les conditions d'ordonnançabilité suffisantes données précédemment concernant les tâches NPPS. Pour cela on génère aléatoirement des jeux de tâches et on les ordonnance selon les algorithmes 1 et 2. Les ordonnancements obtenus sont comparés avec l'algorithme optimal 3 afin de calculer le taux de succès.

Définition 3.2 *Le taux de succès SR (Success Ratio) par rapport à une condition suffisante (algorithmes 1 et 2) est le nombre de jeux de tâches ordonnançables selon cette condition suffisante divisé par le nombre de jeux de tâches ordonnançables avec une condition nécessaire et suffisante (algorithme optimal 3).*

3.6.1 Algorithme de génération de tâches

Bini et Buttazzo ont proposé l'algorithme *UUnifast* [BB05] qui génère une distribution non biaisée de n facteurs d'utilisations U_i. Le Programme 1 Matlab correspond à cet algorithme, où n est le nombre de tâches et Ut est la valeur de l'utilisation cible.

Programme 1 Programme Matlab de l'algorithme UUnifast

```
function vectU =UUniFast (n, Ut)
sumU = Ut ;
for i=1 :n-1,
    nextSumU = sumU .* rand ^ (1/(n − i)) ;
    vectU(i) = sumU - nextSumU ;
    sumU = nextSumU ;
end
vectU(n) = USum ;
```

L'algorithme UUnifast permet de générer n facteurs d'utilisation u_i. Afin de générer n tâches $\tau_i(C_i, T_i)$, nous avons proposé un algorithme qui utilise l'algorithme UUnifast pour la génération des facteurs d'utilisation, et une loi de distribution normale pour la génération des durées d'exécutions C_i et les périodes T_i. Les dates de début d'exécution s_i sont initialisées avec des zéros. On rappelle que ces dates sont calculées par les algorithmes d'ordonnancement 1, 2 et 3.

Le Programme 2 Matlab correspond à cet algorithme, où n est le nombre de tâches, Tm est la moyenne des périodes et Ut le facteur d'utilisation cible.

Programme 2 Programme Matlab de génération de tâches non harmoniques

```
function Gsp = gen_non_harmonic(n,Tm,Ut)
Ui = UUnifast(n,Ut) ;                    % Génération des ui
T1 =normrnd(Tm , 0.5*Tm , n,1) ; % Génération des Ti
T = ceil(max(2*ones(n,1), T1)) ;         % Ti entiers et vaux au moins 2
C1 = Ui .* T ;                           % Calcul des Ci
C = floor(max(ones(n,1),c2)) ;           % Ci entiers et vaux au moins 1
S = zeros(n,1) ;                         % Initialisation des si par 0
G = [S C T] ;                            % Ensemble de tâches générés
```

L'algorithme correspondant au programme Matlab de génération de tâches selon la loi normale procède en trois étapes comme suit :

1. générer un ensemble de n valeurs de facteur utilisation unitaire u_i en utilisant l'algorithme UUnifast pour une cible de facteur d'utilisation de U ;
2. générer un ensemble de n périodes T_i en utilisant une loi de distribution normale, avec une moyenne de Tm et un écart type $\sigma = 0.5 \cdot Tm$;
3. calculer les durées d'exécutions C_i de chaque tâche : $C_i = u_i \cdot T_i$.

3.6.2 Simulations et discussion des résultats

Tout les résultats de simulations ont été obtenus avec une machine *Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz*.

L'algorithme de simulation 4 utilisé pour calculer les différents taux de succès procède en cinq étapes :

Algorithme 4 Algorithme de simulation

1: Générer un ensemble de n tâches avec un facteur d'utilisation cible Ut, une période moyenne Tm et un écart type σ
2: Calculer le facteur d'utilisation U de l'ensemble de tâches générées
3: **si** U est proche de Ut (contenu dans une marge donnée) **alors**
4: Utiliser cet ensemble de tâches pour les simulations
5: **sinon**
6: Ignorer cet ensemble de tâches et en générer un nouveau
7: **fin si**
8: Calculer la moyenne des taux de succès

Pour simplifier les notations nous proposons de donner un nom aux différentes conditions que nous utilisons :

- CNS pour condition nécessaire et suffisante (3.6),
- CS1 pour condition suffisante (3.12),
- CS2 pour condition suffisante (3.18).

Nous nous intéressons dans nos simulations aux ensembles des jeux de tâches suivants donnés dans la figure 3.13 :

- ensemble des jeux de tâches générés,
- ensemble des jeux de tâches ordonnançables avec la condition CNS qui est un sous-ensemble de l'ensemble précédent,
- ensemble des jeux de tâches ordonnançables avec la condition $CS1$ ou $CS2$ qui est un sous-ensemble de l'ensemble précédent,
- ensemble des jeux de tâches ordonnançables avec la condition $CS1$ qui est un sous-ensemble de l'ensemble précédent.

On génère 50000 jeux de tâches pour chaque valeur du facteur d'utilisation allant de 0 à 1 par incrément de 0.1. La condition CNS est très coûteuse surtout lorsque le nombre de tâches devient important. C'est la raison pour laquelle on simule des jeux de tâches qui ne dépassent pas 4 tâches comme dans les figures 3.14. En effet calculer le taux de succès pour une seule valeur du facteur d'utilisation pour 5 tâches a pris plus de deux jours.

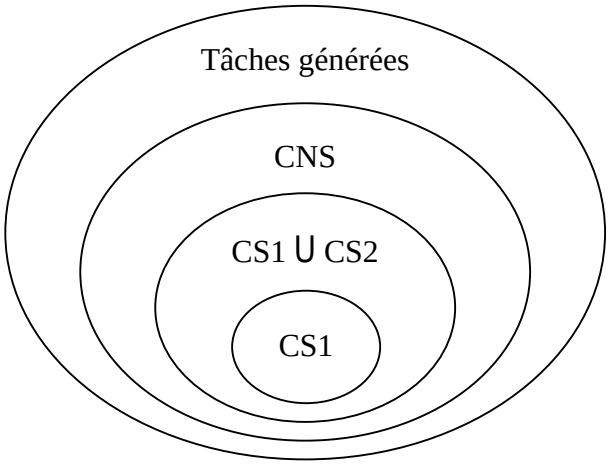


FIGURE 3.13 – Différents ensemble de tâches

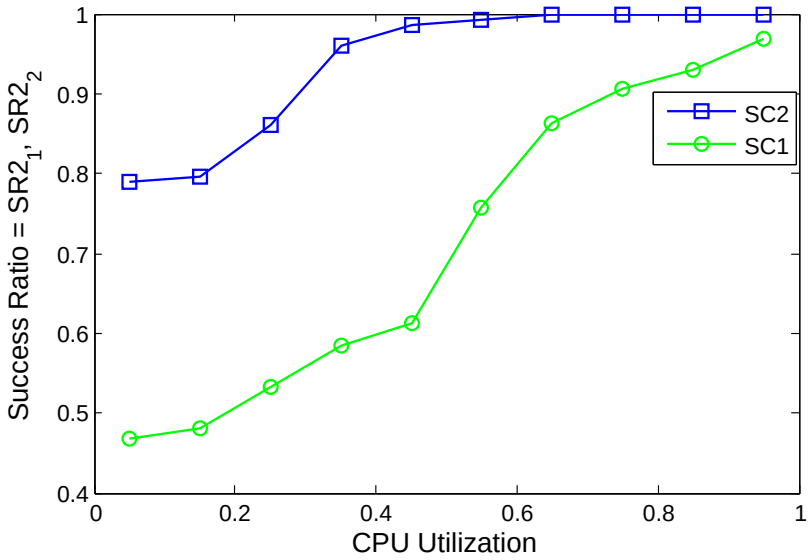


FIGURE 3.14 – Taux de succès de tâches NPPS en fonction du facteur d'utilisation pour les conditions $CS1$ et $CS2$

La figure 3.14 montre le taux de succès $SR1$ (resp. $SR2$) selon la condition $CS1$ (resp. $CS2$) pour des jeux de tâches non harmoniques. Cette figure montre que la condition $CS2$ est meilleure que la condition $CS1$. En effet le taux de

succès $SR2$ peut dépasser de 40.6% le taux de succès $SR1$. Le taux de succès $SR2$ est égal à 100% pour un facteur d'utilisation supérieur à 60%.

Ces résultats montrent clairement que l'on obtient de meilleurs taux de succès en utilisant notre condition $CS2$ comparé à la condition $CS1$. De plus les taux de succès en utilisant $CS2$ sont toujours supérieurs à 80%, donc notre condition suffisante $CS2$ donne des résultats proches de la condition nécessaire et suffisante CNS.

3.7 Conclusion

Nous avons présenté dans ce chapitre l'analyse d'ordonnançabilité de tâches NPPS dans le cas monoprocesseur. Nous avons proposé dans un premier temps une stratégie d'ordonnancement qui consiste à ordonnancer une tâche candidate avec un ensemble de tâches déjà ordonnançables. Nous avons utilisé cette même stratégie dans les deux cas de tâches harmoniques et non harmoniques. Ensuite nous avons distingué deux sous-cas des tâches harmoniques : des tâches ayant toutes des périodes distinctes, et des tâches ayant certaines périodes identiques ; le second cas est plus compliqué à étudier que le premier. Nous avons proposé un théorème qui donne une condition d'ordonnançabilité nécessaire et suffisante dans le premier cas, et un théorème qui donne une condition d'ordonnançabilité suffisante dans le second cas. Pour le cas général qui est le cas des tâches non harmoniques, nous avons présenté une analyse d'ordonnançabilité pour plus de deux tâches en proposant un théorème qui donne une condition d'ordonnançabilité moins restrictive que la condition d'ordonnançabilité existante. Finalement nous avons donné un théorème permettant de calculer la phase transitoire et la phase permanente d'un ordonnancement de tâches NPPS.

Dans le chapitre suivant nous présentons une analyse d'ordonnançabilité d'une combinaison de tâches NPPS et de tâches NPPS.

Chapitre 4

Ordonnancement temps réel monoprocesseur combinant non préemptif et préemptif

4.1 Introduction

On présente dans ce chapitre une étude d'ordonnançabilité hiérarchique [DB05, ZGD11] d'une combinaison de tâches NPPS et de tâches PP. Cette étude mène à une condition d'ordonnançabilité nécessaire et suffisante.

On suppose que les tâches NPPS ont déjà été ordonnancées et les tâches PP sont à ordonnancer. On rappelle que les tâches NPPS sont des tâches qui ne doivent pas avoir de jitter telles les tâches de commande et les tâches liées aux capteurs/actionneurs. On traite un problème général d'ordonnancement à priorité fixe où l'on attribue la même plus haute priorité aux tâches NPPS et les basses priorités aux tâches PP. Pour les tâches PP on a choisi de les ordonnancer selon l'algorithme DM (Deadline Monotonic), mais ce qui suit reste valable pour d'autres algorithmes d'ordonnancement à priorité fixe.

Cet ordonnancement est équivalent au problème d'ordonnancement de tâches temps réel à priorité fixe avec un seuil de préemption associé à chaque tâche [WS99], pour lequel la priorité de chaque tâche NPPS est supérieure ou égale à son seuil afin de ne pas permettre la préemption, et la priorité de chaque tâche PP est inférieure à son seuil afin de permettre la préemption.

Nous rappelons que les tâches NPPS sont notées $\tau_i(C_i, T_i)$, et les tâches PP sont notées $\tau_i(r_i, C_i, D_i, T_i)$ (cf. chapitre 1 section 1.2.1.1).

Dans la suite du chapitre on utilise les notations suivantes :

– Γ^S représente l'ensemble de tâches NPPS ;

– Γ^{NS} représente l'ensemble de tâches PP ;

– $hp^{NS}(i)$ représente l'ensemble de tâches de Γ^{NS} ayant des priorités plus hautes que celle d'une tâche τ_i de Γ^{NS}.

4.2 Combinaison de tâches NPPS et PP

4.2.1 Stratégie d'ordonnancement

L'analyse d'ordonnancement d'une combinaison de tâches NPPS et de tâches PP est basée sur la stratégie d'ordonnancement suivante :

– on considère qu'un système de tâches NPPS est déjà ordonnancé selon l'étude d'ordonnançabilité présentée dans le chapitre 3. On attribue à ces tâches les plus hautes priorités ;
– on considère un ensemble de tâches PP à ordonnancer. On attribue à ces tâches des priorités selon l'algorithme DM,
– pour qu'une tâche PP soit ordonnançable, il faut que son pire temps de réponse soit inférieur à son échéance relative,
– on calcule pour chaque tâche PP son pire temps de réponse et ce suivant l'ordre croissant de leurs priorités,
– si le pire temps de réponse de toutes les tâches PP est inférieur à leurs échéances relatives alors ces tâches sont ordonnançables, sinon elle ne sont pas ordonnançables.

À noter que les algorithmes d'ordonnancement proposés dans ce chapitre sont de type *hors ligne*. De plus les tâches NPPS sont considérés asynchrones alors que les tâches PP peuvent être synchrones ou asynchrones.

4.2.2 Analyse d'ordonnançabilité

Afin d'étudier l'ordonnançabilité des tâches PP, alors que des tâches NPPS sont déjà ordonnancées, on doit tout d'abord déterminer les instants critiques où l'on obtient les pires temps de réponse notés WCRT (Worst Case Response Time) d'une tâche PP τ_i^j.

Il a été démontré dans [JP86] qu'un instant critique est atteint lorsque la date d'activation d'une tâche coïncide avec les dates d'activations d'une tâche de plus haute priorité. Comparées aux tâches PP, les tâches NPPS ont toutes la plus haute priorité.

On définit un ensemble Ψ contenant tous les instants critiques des tâches NPPS dans l'intervalle de temps $[0, \phi + L[$ qui comprend la phase transitoire de longueur ϕ et une unique hyper-période de la phase permanente de longueur L (cf. section 3.5).

Afin d'étudier l'ordonnançabilité des tâches PP, on doit calculer les pires temps de réponse WCRT de ces tâches lorsque leurs dates d'activations coïncident avec celles des tâches NPPS (de haute priorité). Cela revient à considérer toutes les dates appartenant à l'ensemble Ψ.

Le lemme suivant a été donné dans [JP86].

Lemme 4.1 *Le pire temps de réponse d'une tâche PP τ_i est obtenu lorsque sa date d'activation est égale à la date d'activation d'une tâche NPPS $r_i^k = S_j^l$, où S_j^l est la j^{me} date de début d'exécution d'une tâche NPPS.*

Afin qu'une tâche PP puisse être ordonnançable en présence de tâches NPPS, le pire temps de réponse de cette tâche doit être inférieur à son échéance relative pour toutes les tâches appartenant à l'ensemble Ψ.

Cependant il est inutile de calculer les temps de réponse pour tous les instants de l'ensemble Ψ comme le montre le lemme suivant.

Lemme 4.2 *Considérons un ensemble de tâches NPPS déjà ordonnancées. Pour chaque séquence d'instances de tâches NPPS $seq = \{S_i, S_{i+1}, ..., S_j\} \subset \Psi$ ordonnancées sans temps creux, i.e. $\exists S_i^k,\ S_j^l \in \Psi$ tels que $S_i^k - S_j^l = C_j$, le pire temps de réponse d'une tâche PP est obtenu lorsque la date d'activation de cette dernière est égale à la date d'activation de la première instance de cette séquence.*

Preuve
Considérons une séquence seq d'exécutions consécutives des tâches NPPS appartenant à Γ^S. On suppose que 0 est l'origine de temps et qu'elle correspond à la date d'activation de la première tâche NPPS dans la séquence seq. Comme les tâches PP ont des priorités plus basses que celles des tâches NPPS, une tâche PP activée à la date $t_i \geq 0$ ne peut commencer son exécution que lorsque toutes les tâches NPPS dans seq auront terminé leurs exécutions, i.e. après la somme des durées d'exécution des tâches NPPS. Le pire temps de réponse est donc obtenu lorsqu'une tâche PP est activée à la date 0. ∎

Donc afin de minimiser le nombre de calculs des temps de réponse, pour chaque séquence $seq = \{S_i, S_{i+1}, ..., S_j\}$ s'exécutant sans temps creux, on remplace l'ensemble de ces dates d'activations par la première date d'activation S_i.

Le lemme suivant nous permet de réduire l'ensemble Ψ aux dates d'activations contenues uniquement dans la phase permanente.

Lemme 4.3 *Dans l'ensemble des dates d'activations Ψ, seulement les dates d'activations appartenant à la phase permanente seront prises en compte dans le calcul des temps de réponse.*

Preuve
Pour les tâches NPPS, les dates d'activation de la phase transitoire sont incluses dans la phase permanente. En effet la phase transitoire peut contenir au plus autant d'instances que la phase permanente, c'est pourquoi seulement les dates d'activations appartenant à la phase permanente seront prises en compte pour calculer les temps de réponse. ∎

Avant d'ordonnancer les tâches PP, nous nous basons sur les lemmes 4.1, 4.2 et 4.3 pour calculer et réduire l'ensemble des instants critiques Ψ. Nous nous intéressons ensuite à l'ordonnancement des tâches PP. Le théorème suivant permet de calculer le temps demandé au processeur à un instant t pour une tâche τ_i activée à l'instant $r_i \in \Psi$.

Théorème 4.1 *Soit $\tau_i(r_i, C_i, D_i, T_i)$ une tâche PP activée à l'instant $r_i \in \Psi$. Le temps demandé au processeur à l'instant t est donné par*

$$W_i(t) = C_i + \sum_{\tau_j \in \Gamma^S} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_j + \sum_{\tau_j \in hp^{NS}(i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{4.1}$$

où S_j est la date de début d'exécution des tâches NPPS relative à l'instant r_i, donnée par

$$S_j = s_j + \left\lceil \frac{r_i - s_j}{T_j} \right\rceil T_j - r_i \tag{4.2}$$

Preuve
Considérons qu'une tâche PP τ_i est activée à $r_i \in \Psi$. Le temps demandé au processeur à l'instant t (relativement à l'instant r_i) est la somme des temps de calculs suivants :

1. temps d'exécution C_i de la tâche NPPS τ_i
2. temps demandé au processeur par les tâches NPPS tâches NPPS

$$\sum_{\tau_j \in \Gamma^S} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_j$$

3. temps demandé au processeur par les tâches PP ayant des priorités plus hautes que celle de la tâche τ_i :

$$\sum_{\tau_j \in hp^{NS}(i)} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

■

Le théorème suivant donne une condition d'ordonnançabilité nécessaire et suffisante pour un ensemble de tâches PP.

Théorème 4.2 *Un ensemble de tâches PP Γ^{NS} est ordonnançable si et seulement si*

$$\forall r_i \in \Psi,\ \forall \tau_i \in \Gamma^{NS} :\ RT_i \leq D_i \tag{4.3}$$

où RT_i est le temps de réponse de $\tau_i \in \Gamma^{NS}$, solution de

$$RT_i = W(RT_i) \tag{4.4}$$

calculée itérativement en initialisant RT_i à zéro.

Preuve
La preuve est identique à celle donnée dans [JP86] où le pire temps de réponse de chaque tâche doit être inférieur ou égal à son échéance. ■

Exemple
On utilise l'algorithme DM pour ordonnancer la combinaison des tâches NPPS $\Gamma^S = \{\tau_1(1,4), \tau_2(1,6), \tau_3(1,12)\}$ et les tâches PP $\Gamma^{NS} = \{\tau_4(r_4,2,6,8), \tau_5(r_5,2,12,12)\}$.

Les tâches τ_1 et τ_2 satisfont la condition (3.12), elles sont donc ordonnançables et leur dates de début d'exécution sont respectivement $s_1 = 0$ et $s_2 = 1$. Les tâches τ_3 et τ_1 satisfont la condition (3.18), la tâche τ_3 est donc ordonnançable avec $s_3 = 2$.

L'hyper-période est égale à $L = LCM(T_1, T_2, T_3) = 12$, donc l'ensemble des instants critiques est donné par $\Psi = \{0, 1, 2, 4, 7, 8\}$ et qui correspond aux dates de début d'exécution des tâches de Γ^S dans l'intervalle de temps $[0, L]$. Selon le lemme 4.2, l'ensemble des instants critiques est réduit à $\Psi = \{0, 4, 7\}$.

Pour les deux tâches PP on a :

$$W_4(t) = 2 + \sum_{j=1}^{3} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_j$$

et

$$W_5(t) = C_5 + \left\lceil \frac{t}{T_4} \right\rceil C_4 + \sum_{j=1}^{3} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_j.$$

Le temps de réponse des tâches τ_4 et τ_5 pour des dates d'activations égales aux instants $r \in \Psi$ sont donnés par

r	S_1	S_2	S_3	RT_4	RT_5
0	0	1	2	6	12
4	0	3	4	3	7
7	1	0	7	4	9

TABLE 4.1 – Calcul des pires temps de réponse RT_4 et RT_5

Comme $RT_4 \leq D_4$ et $RT_5 \leq D_5$, τ_4 et τ_5 sont ordonnançables comme le montre la figure 4.1.

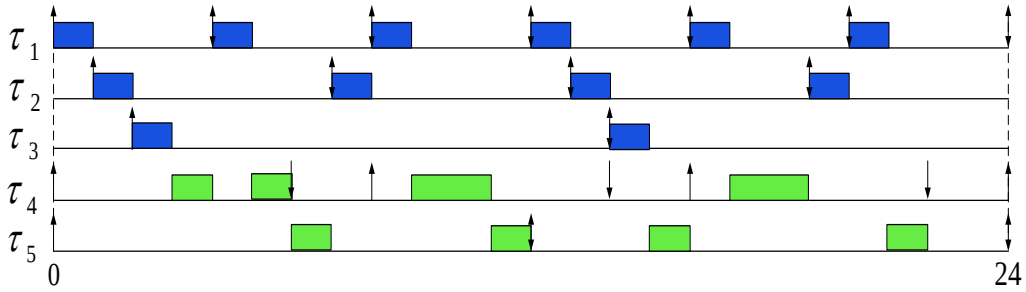


FIGURE 4.1 – Diagramme d'ordonnancement de tâches PP et NPPS

4.3 Ordonnancement de tâches préemptives périodiques strictes

L'objectif de ce travail est d'étendre les travaux d'ordonnancement de tâches NPPS aux tâches préemptives périodiques strictes, que l'on note *PPS*, afin d'améliorer le taux de succès d'ordonnancement de ces tâches. Pour cela nous proposons d'abord dans la section suivante un modèle de tâches PPS, ensuite nous donnerons une condition d'ordonnançabilité nécessaire et suffisante.

4.3.1 Modèle de tâches

Une tâche PPS notée $\tau_i(s_i, C_{i,1}, C_{i,2}, D_i, T_i)$ est constituée d'une partie non préemptive de durée d'exécution $C_{i,1}$ qui doit démarrer son exécution à la date s_i, et d'une partie préemptive d'une durée d'exécution $C_{i,2}$ qui s'exécute après la date $s_i + C_{i,1}$.

Cette tâche peut être décomposée en une tâche NPPS $\tau_{i,1}(s_i, C_{i,1}, D_1, T_i)$ de haute priorité, et une tâche PP $\tau_{i,2}(s_i, C_{i,2}, D_1, T_1)$ de plus faible priorité comme le montre la figure 4.2.

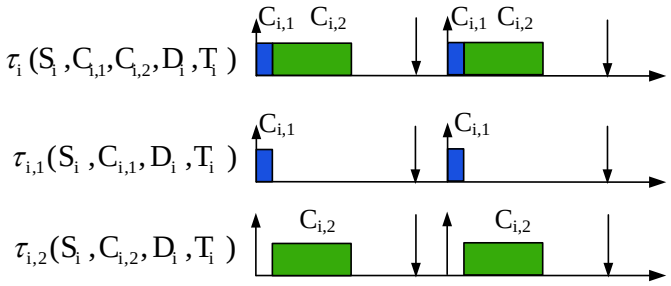


FIGURE 4.2 – Décomposition d'une tâche PPS en une tâche NPPS et une tâche PP

4.3.2 Analyse d'ordonnançabilité

Soit $\Gamma = \{\tau_i(s_i, C_{i,1}, C_{i,2}, D_i, T_i), i = 1..n\}$ un ensemble de tâches PPS ayant les priorités p_i qui peuvent être arbitraires. En décomposant chaque tâche de l'ensemble Γ on obtient les deux ensembles suivants :

- l'ensemble de tâches NPPS de hautes priorités : $\Gamma^S = \{\tau_{i,1}(s_i, C_{i,1}, D_i, T_i), i = 1..n\}$,
- l'ensemble de tâches PP de faibles priorités : $\Gamma^{SN} = \{\tau_{i,2}(s_i, C_{i,2}, D_i - C_i, T_i),\ i = 1..n\}$.

On considère un ordonnancement à priorité fixe où les hautes priorités sont données aux tâches de Γ^S. Nous calculons les priorités des tâches comme suit :

- $p_{i,1} = p_i$
- $p_{i,2} = p_i + n$

où n est le nombre de tâches, p_i et la priorité de la tâche τ_i, $p_{i,1}$ et la priorité de la tâche $\tau_{i,1}$ et $p_{i,2}$ et la priorité de la tâche $\tau_{i,2}$.

On note $hp(i) = \{\tau_{i,2}(s_i, C_{i,2}, D_i, T_i),\ i = 1..(i-1)\}$ l'ensemble de tâches de Γ^{SN} qui sont plus prioritaires que la tâche $\tau_{i,2}$.

Le lemme suivant donne les instants critiques pour une tâche $\tau_{i,2}$.

Lemme 4.4 *Soit $\Gamma = \{\tau_i(s_i, C_{i,1}, C_{i,2}, D_i, T_i), i = 1..n\}$ un ensemble de tâches PPS à ordonnancer. L'ensemble des instants critiques pour une tâche PP $\tau_{i,2}(s_i, C_{i,2}, D_i - C_i, T_i)$ est donné par*

$$\Psi_i = \{s_i^k,\ s_i^k \in [\phi, \phi + L[\} \tag{4.5}$$

où ϕ est la longueur de la phase transitoire des tâches NPPS et L est la longueur de la phase permanente.

Preuve
Comme chaque tâche $\tau_{i,2}$ est activée à l'instant s_i alors ses instants critiques correspondent aux dates de début d'exécutions s_i^k, incluses dans la phase transitoire ϕ et une unique hyper-période de la phase permanente de longueur L : $\Psi_i = \{s_i^k,\ s_i^k \in [0, \phi + L[\}$.

D'après le lemme 4.3, l'ensemble des instants critiques est inclus dans la phase transitoire, donc $\Psi_i = \{s_i^k,\ s_i^k \in [\phi, \phi + L[\}$. ∎

Le théorème suivant donne le temps demandé au processeur à un instant $t \geq 0$ lorsqu'une tâche $\tau_{i,2}$ est activée à l'instant $r_i \in \Psi_i$.

Théorème 4.3 *Soit $\Gamma = \{\tau_i(s_i, C_{i,1}, C_{i,2}, D_i, T_i), i = 1..n\}$ un ensemble de tâches PPS à ordonnancer. Considérons que les ensembles $\Gamma^S = \{\tau_{i,1}(s_i, C_{i,1}, D_i, T_i), i = 1..n\}$ et $hp(i) = \{\tau_{i,2}(s_i, C_{i,2}, D_i - C_i, T_i),\ i = 1..(i-1)\}$ sont déjà ordonnancés. Soit $\tau_{i,2}$ une une tâche activée à l'instant $r_i \in \Psi_i$ Le temps demandé au processeur à l'instant $t \geq 0$ (relativement à l'instant r_i) est donné par*

$$W_{i,2}(t) = C_{i,2} + \sum_{j=i}^{n} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_{j,1} + \sum_{j=1}^{i-1} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_{j,2} + \sum_{j=1}^{i-1} max\left[0,\ (RT_{i,2}(r_i + S_j - T_j) + S_j - T_j)\right] \tag{4.6}$$

où S_j est la date de début d'exécution s_j^k relativement à l'instant r_i donnée par

$$S_j = s_j^0 + \left\lceil \frac{r_i - s_j^0}{T_j} \right\rceil T_j - r_i \tag{4.7}$$

et $RT_{i,2}$ est le temps de réponse de $\tau_{i,2}$, solution de l'équation

$$W_{i,2}(RT_{i,2}) = RT_{i,2}.$$

Preuve
Considérons une tâche $\tau_{i,2}$ activée à l'instant $r_i \in \Psi_i$. Le temps demandé au processeur à l'instant $t \geq 0$ (relativement à l'instant r_i) est la somme des temps de calculs suivants :

1. temps d'exécution $C_{i,2}$ de la tâche $\tau_{i,2}$,
2. temps demandé au processeur par les tâches $\tau_{j,1}$ donné par

$$\sum_{j=i}^{n} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_{j,1},$$

3. temps demandé au processeur par les tâches $\tau_{j,2} \in hp(i)$ plus prioritaires que la tâche $\tau_{i,2}$, donné par :

$$\sum_{j=1}^{i-1} \left\lceil \frac{t - S_j}{T_j} \right\rceil C_{j,2},$$

4. pour chaque tâche $\tau_{j,2} \in hp(i)$ activée avant l'instant s_i (à l'instant $r_i + S_j - T_j$), si elle a terminé son exécution avant l'instant s_i alors elle ne rajoute aucun temps de calcul, par contre si elle ne termine pas son exécution avant l'instant s_i alors elle rajoute un temps de calcul égal à la différence entre son temps de réponse $RT_{i,2}(r_i + S_j - T_j)$, et le temps écoulé entre sa date d'activation $r_i + S_j - T_j$ et l'instant r_i qui est $S_j - T_j$.
   Le temps rajouté par la tâche $\tau_{j,2}$ est donc

$$max\,[0,\ RT_{i,2}(r_i + S_j - T_j) - (T_j - S_j)]$$

   et le temps de calcul rajouté par l'ensemble des tâches $\tau_{j,2} \in hp(i)$ est

$$\sum_{j=1}^{i-1} max\,[0,\ (RT_{i,2}(r_i + S_j - T_j) + S_j - T_j)]\,.$$

■

Le théorème suivant donne une condition d'ordonnançabilité nécessaire et suffisante pour des tâches PPS.

Théorème 4.4 *Soit $\Gamma = \{\tau_i(s_i, C_{i,1}, C_{i,2}, D_i, T_i), i = 1..n\}$ un ensemble de tâches PPS à ordonnancer. Considérons que les ensembles $\Gamma^S = \{\tau_{i,1}(s_i, C_{i,1}, D_i, T_i), i = 1..n\}$ et $hp(i) = \{\tau_{i,2}(s_i, C_{i,2}, D_i - C_i, T_i),\ i = 1..(i-1)\}$ sont déjà ordonnancés. Soit $\tau_{i,2}$ une tâche à ordonnancer. Soit $\Psi_i = \{s_i^k,\ s_i^k \in [\phi, \phi + L[\}$ l'ensemble des instants critiques. $\tau_{i,2}$ est ordonnançable si et seulement si*

$$\forall r_i \in \Psi_i,\ RT_{i,2} \leq D_i \tag{4.8}$$

où $RT_{i,2}$ est le temps de réponse de $\tau_{i,2}$ solution de

$$RT_{i,2} = W(RT_{i,2}) \tag{4.9}$$

calculée itérativement en initialisant $RT_{i,2}$ à zéro.

Preuve
La preuve est identique à celle donnée dans [JP86] où le pire temps de réponse de chaque tâche doit être inférieur ou égal à son échéance. ■

Exemple
Nous considérons l'ensemble de trois tâches PPS $\Gamma = \{\tau_1(0,1,1,4,6), \tau_2(1,1,2,6,9), \tau_3(2,1,2,8,12)\}$ à ordonnancer. Ce problème d'ordonnancement se transforme en un problème d'ordonnancement de la combinaison de des ensembles suivantes :

- l'ensemble de tâches NPPS de hautes priorités attribuées selon l'algorithme DM :
  $\Gamma^S = \{\tau_{1,1}(0,1,4,6), \tau_{2,1}(1,1,6,9), \tau_{3,1}(2,1,8,12)\}$,
- l'ensemble de tâches PP de faibles priorités :
  $\Gamma^{SN} = \{\tau_{1,2}(0,1,4,6), \tau_{2,2}(1,2,6,9), \tau_{3,2}(2,2,8,12)\}$.

Les tâches de Γ^S satisfont la condition d'ordonnançabilité 3.12 donc elles sont ordonnançables. En effet : $(C_1 + C_2 + C_3 = 3) \leq (PGCD(T_1, T_2, T_3) = 3)$.

Nous allons maintenant étudier l'ordonnançabilité des tâches de Γ^{SN}. L'hyper-période est $L = PPCM(6,9,12) = 36$. D'après le théorème 3.11 la phase transitoire est nulle : $\phi = \underset{i=1..3}{Max}(0,\ s_i + C_i - T_i) = max(0,-5,-7,-9) = 0$. Les instants critiques appartiennent donc à l'intervalle $[0, 36[$.

Pour la tâche $\tau_{1,2}(s_i, 1, 4, 6)$ l'ensemble des instants critiques est $\Psi_1 = \{0, 6, 12, 18, 24, 30\}$. Le tableau 4.2 regroupe les instants de démarrages relatives S_i et les temps de réponse $RT_{1,2}$. Comme $RT_{1,2} \leq 4$ alors $\tau_{1,2}$ est ordonnançable.

r_i	S_1	S_2	S_3	$RT_{1,2}$
0	0	1	2	4
6	0	4	8	2
12	0	7	2	2
18	0	1	8	3
24	0	4	2	2
30	0	7	8	2

TABLE 4.2 – Calcul des pires temps de réponse $RT_{1,2}$

Pour la tâche $\tau_{2,2}(s_i, 2, 6, 9)$ l'ensemble des instants critiques est $\Psi_2 = \{1, 10, 19, 28, 37\}$. Le tableau 4.3 regroupe les instants de démarrages relatives S_i et les temps de réponse $RT_{2,2}$. Comme $RT_{2,2} \leq 6$ alors $\tau_{2,2}$ est ordonnançable.

r_i	S_1	S_2	S_3	$RT_{2,2}$
1	0	0	2	5
10	2	0	4	6
19	5	0	7	4
28	2	0	10	5
37	5	0	1	5

TABLE 4.3 – Calcul des pires temps de réponse $RT_{2,2}$

Pour la tâche $\tau_{3,2}(s_i, 2, 8, 12)$ l'ensemble des instants critiques est $\Psi_3 = \{2, 14, 26\}$. Le tableau 4.4 regroupe les instants de démarrages relatives S_i et les temps de réponse $RT_{3,2}$. Comme $RT_{3,2} \leq 8$ alors $\tau_{3,2}$ est ordonnançable.

r_i	S_1	S_2	S_3	$RT_{3,2}$
2	4	8	0	8
8	4	5	0	4
12	4	2	0	8

TABLE 4.4 – Calcul des pires temps de réponse $RT_{3,2}$

Comme tous les temps de réponses des tâches Γ^{SN} sont inférieurs ou égaux à leurs échéances alors Γ^{SN} est ordonnançable et donc Γ est aussi ordonnançable. La figure 4.3 montre l'ordonnancement des tâches de Γ.

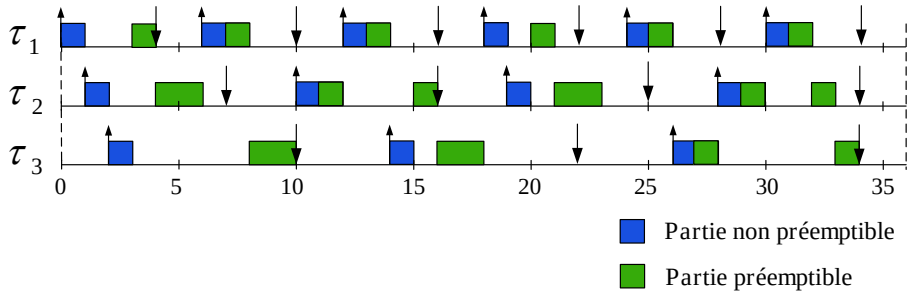


FIGURE 4.3 – Diagramme d'ordonnancement de tâches PPS

4.4 Conclusion

Nous avons présenté dans ce chapitre une étude d'ordonnançabilité d'une combinaison de tâches NPPS et de tâches PP, ayant toutes des priorités fixes.

Les tâches NPPS ont toutes la même plus haute priorités, et les tâches PP ont des priorités croissantes en fonction de leurs échéances relatives (algorithme d'ordonnancement Deadline Monotonic).

Nous avons d'abord montré comment construire l'ensemble des instants critiques correspondant aux dates de début d'exécution des tâches NPPS, puis nous avons proposé deux lemmes permettant de ne considérer que les instants critiques appartenant à la phase permanente, i.e. de ne pas considérer les instants critiques de la phase transitoire, et de ne garder que le premier instant critique lorsque plusieurs instances s'exécutent sans temps creux dans la phase permanente. Ensuite nous avons donné un théorème qui permet de calculer le temps demandé au processeur pour une tâche PP combinée avec les tâches plus prioritaires (NPPS et PPS). Nous avons aussi proposé un théorème qui donne une condition d'ordonnançabilité nécessaire et suffisante, calculant itérativement les pires temps de réponse pour chaque tâche PP et les comparant à leurs échéances relatives. Enfin nous avons étendu ces résultats pour l'analyse d'ordonnançabilité des tâches PPS.

Ce chapitre conclut l'étude d'ordonnançabilité dans le cas monoprocesseur. Dans le chapitre suivant nous présentons l'étude d'ordonnancement multiprocesseur de tâches NPPS, basée sur l'étude d'ordonnancement monoprocesseur que l'on a présentée dans le chapitre 3.

Troisième partie

Ordonnancement temps réel multiprocesseur

Chapitre 5

Ordonnancement temps réel multiprocesseur non préemptif périodique strict

5.1 Introduction

Nous nous intéressons aux applications temps réel de robotique mobile spécifiées à l'aide d'un graphe d'algorithme décrivant les fonctionnalités à réaliser et d'un graphe d'architecture décrivant les processeurs et les média de communication, qui peuvent être de types différents (architecture hétérogène), sur lesquels devront s'exécuter ces fonctionnalités en respectant des contraintes temps réel. On suppose que les sommets et les dépendances du graphe d'algorithme sont étiquetés temporellement par des durées pire temps d'exécution, des périodes et des durées de communication. Dans le cas d'une architecture hétérogène une tâche (resp. une dépendance de donnée) peut être étiquetée par plusieurs durées, une pour chaque processeur (resp. dépendance) concerné. Dès que le graphe d'algorithme est étiqueté temporellement nous avons un graphe de tâches temps réel dépendantes. Comme nous visons des applications de contrôle/commande robotique critiques, nous considérons que ces tâches sont NPPS (cf. section 3.1). Pour éviter qu'il y ait des pertes de données entre les tâches dépendantes, nous considérons que toutes les tâches dépendantes ont des périodes multiples. Cela ne veux pas dire pour autant que le système de tâches doit être harmonique mais seuls les couples des tâches dépendantes doivent être harmoniques (cf. section 5.3). Nous rappelons que nous avons choisi d'utiliser l'approche d'ordonnancement multiprocesseur par partitionnement qui consiste à partitionner l'ensemble des tâches à ordonnancer en sous-ensembles de tâches ordonnancés chacun sur un processeur, ce qui nous permet d'utiliser une stratégie d'ordonnancement monoprocesseur et

donc profiter des résultats d'analyse d'ordonnançabilité présentés dans le chapitre 3.

Afin de réaliser l'ordonnancement multiprocesseur non préemptif périodique strict il faut tout d'abord effectuer une analyse d'ordonnançabilité sur le graphe de tâches dépendantes, puis répéter chaque tâche selon sa période relativement aux autres périodes tout en respectant les dépendances entre les tâches et enfin calculer pour chaque processeur le début d'exécution de chaque tâche.

Nous présenterons donc dans ce chapitre l'algorithme d'ordonnancement multiprocesseur que nous avons proposé, formé de trois algorithmes qui s'exécutent dans l'ordre suivant : (i) algorithme d'analyse d'ordonnançabilité, (ii) algorithme de déroulement, (iii) algorithme d'ordonnancement.

Tout au long de ce chapitre nous considérons un graphe d'algorithme Alg constitué d'un ensemble Γ de n tâches dépendantes, et une architecture Arc de m processeurs et l bus de communication.

5.2 Analyse d'ordonnançabilité

Comme le problème à résoudre est équivalent à un problème de bin-packing (cf. section 1.2.4.2) nous avons étendu l'algorithme First-Fit de la manière suivante. Pour chaque tâche à assigner, en commençant par le premier processeur il essaye d'assigner à tous les processeurs cette tâche en vérifiant une condition d'ordonnançabilité. Mais au lieu de s'arrêter dès qu'il trouve un processeur sur lequel cette tâche est ordonnançable, il continue à parcourir les autres processeurs pour essayer d'y assigner cette tâche. Avec cet algorithme, une tâche peut être assignée à plusieurs processeurs. Ceci va permettre de choisir parmi les différentes assignations celle qui minimise le temps d'exécution de l'ensemble des tâches et des coûts de communication. C'est à l'algorithme d'ordonnancement de décider sur quel processeur une tâche assignée à plusieurs processeurs va être finalement ordonnancée.

L'algorithme 5 décrit l'analyse d'ordonnançabilité.

5.3 Déroulement

Dans cette section nous nous intéressons à la transformation du graphe d'algorithme initial en un graphe déroulé où chaque tâche τ_i sera répétée en $(\frac{L}{T_i})$ instances, où L est l'hyper-période du système de tâches donnée par $L = PPCM(T_i, i = 1, ..., n)$. Cet algorithme est le même que celui donné par O. Kermia dans [Ker09, KS07].

Algorithme 5 Algorithme d'analyse d'ordonnançabilité

```
1: Initialisation de l'ensemble de tâches répliques Γ
2: Trier les tâches selon les périodicités de leurs périodes
3: pour chaque tâche τ_i faire
4:    pour chaque processeur P_j faire
5:       si si la condition d'ordonnançabilité (3.12), (3.14) ou (3.16) est satisfaite
         alors
6:          assigner cette tâche à ce processeur
7:       sinon
8:          cette tâche n'est pas ordonnançable sur ce processeur
9:       fin si
10:   fin pour
11:   si la tâche n'est assignée à aucun processeur on dit que le système de tâches
      n'est pas ordonnançable et on arrête la boucle
12: fin pour
```

5.3.1 Périodicité et transfert de données

Nous avons présenté dans la section 1.3.1 le modèle flot de données, et nous avons souligné qu'il devait être sans perte de données. Pour cette raison, toutes les tâches dépendantes doivent avoir des périodes égales ou, dans le cas général, multiples les unes des autres.

Si l'on considère deux tâches dépendantes τ_i et τ_j ayant respectivement des périodes T_i et T_j, on distingue les trois cas suivants :

1. $T_i = T_j$: dans ce cas les deux tâches τ_i et τ_j s'exécutent au même rythme, et par conséquent on maintient la dépendance $(\tau_i \triangleright \tau_j)$ sans rajout de nouvelles dépendances ;
2. $T_i < T_j$: la tâche τ_i s'exécute plus rapidement que la tâche τ_j : pour une exécution de la tâche τ_j, la tâche τ_i s'exécute $(\frac{T_j}{T_i})$ fois ;
3. $T_i > T_j$: la tâche τ_j s'exécute plus rapidement que la tâche τ_i : pour une exécution de la tâche τ_i, la tâche τ_j s'exécute $(\frac{T_i}{T_j})$ fois.

5.3.2 Algorithme de déroulement

Pour transformer le graphe d'algorithme initial Alg en un d'algorithme déroulé Alg^*, on doit d'abord répéter chaque tâche $(\frac{P}{T_i})$ fois, où P est l'hyper-période du système de tâches, ensuite on crée des relations de précédence entre les répétitions de la même tâche, et des relations de dépendance entres les répétitions des tâches dépendantes (cf. section 1.2.1.2).

Algorithme 6 Algorithme de déroulement

1: Initialisation de l'ensemble de tâches Γ à ordonnancer
2: Répéter chaque tâche suivant la valeur du rapport entre l'hyper-période et sa période
3: Ajouter une précédence entre chaque paire de répétitions successives de la même tâche
4: Ajouter les dépendances entre les répétitions des tâches productrices et celles des tâches consommatrices

L'algorithme 6 décrit les étapes du déroulement.

Exemple
On considère le graphe d'algorithme donné par la figure 5.1. Ce graphe est constitué des tâches τ_1, τ_2, τ_3 et τ_4 ayant des périodes égales à $T_1 = 2$, $T_2 = 4$, $T_3 = 6$ et $T_4 = 12$. La valeur de l'hyper-période est égale à $L = PPCM(T_1, T_2, T_3) = 12$.

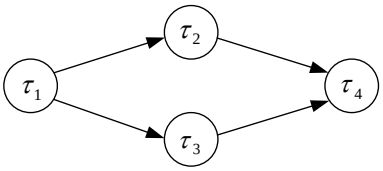


FIGURE 5.1 – Exemple d'un graphe d'algorithme

Le nombre de répétitions des tâches τ_1, τ_2, τ_3 et τ_4 dans une hyper-période est égal à $N_1 = \frac{12}{2} = 6$, $N_2 = \frac{12}{4} = 3$, $N_3 = \frac{12}{6} = 2$ et $N_4 = \frac{12}{12} = 1$.

Aussi, une répétition de τ_2 dépend de $\frac{4}{2} = 2$ répétitions de la tâches τ_1, une répétition de τ_3 dépend de $\frac{6}{2} = 3$ répétitions de la tâches τ_1 et une répétition de τ_4 dépend de $\frac{12}{2} = 6$ répétitions de la tâches τ_2 et de $\frac{12}{3} = 4$ répétitions de la tâches τ_3

La figure 5.2 montre le graphe d'algorithme déroulé contenant des relations de précédences entre les répétitions de la même tâche et des relations de dépendances entre les répétitions de deux tâches différentes.

5.4 Ordonnancement

Avant de présenter l'algorithme d'ordonnancement il faut d'abord expliquer comment prendre en compte le coût des communications dans le calcul des dates de début d'exécution des tâches.

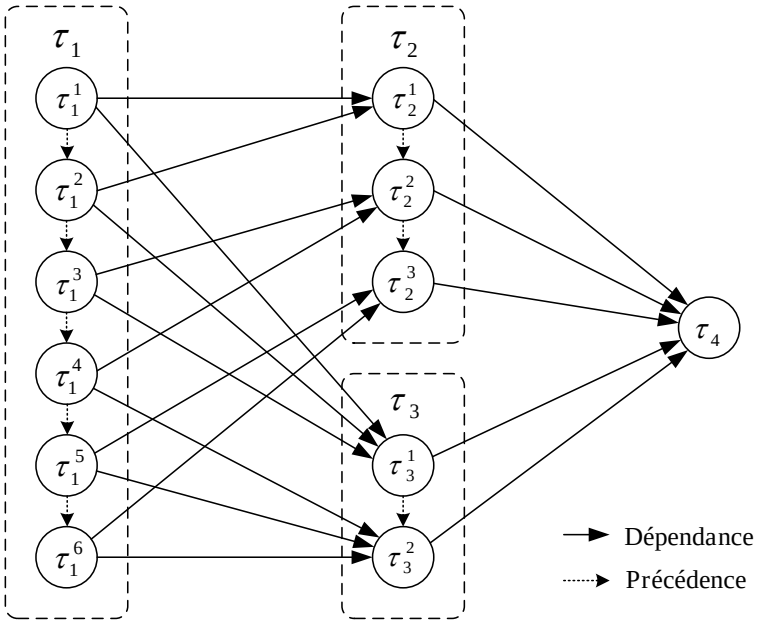


FIGURE 5.2 – Graphe d'algorithme déroulé

5.4.1 Prise en compte des coûts de communications

Nous avons présenté dans le chapitre 3 une étude d'ordonnançabilité monoprocesseur qui permet de savoir si un ensemble de tâches est ordonnançable, et dans ce cas calculer les dates de début d'exécution s_i de chaque tâche τ_i. Comme nous l'avons vu, si une tâche candidate τ_c est ordonnançable alors il existe une infinité de dates de début d'exécution données par les conditions (3.12), (3.14) et (3.16). Par ailleurs nous avons l'objectif de minimiser le makespan (temps total d'exécution de l'ensemble de tâches sur le multiprocesseur en tenant compte des coûts de communication inter-processeur), donc nous avons à calculer la date de démarrage au plus tôt de la tâche τ_c afin qu'elle soit ordonnançable en tenant compte de la communication liée aux dépendances qu'a τ_c avec toutes les tâches prédécesseurs. Une tâche τ_i ne peut commencer son exécution qu'après avoir reçu toutes les données de ces prédécesseurs. On note r_p la date de réception de ces données. Comme nous avons montré qu'il existe une infinité de dates de démarrage pour chaque tâche τ_i, nous allons calculer la plus petite date s_i telle que $s_i \geq r_p$.

Le théorème suivant donne la date de démarrage au plus tôt d'une tâche τ_i qui satisfait la condition (3.12) pour laquelle r_p est la date de réception des données de ses prédécesseurs.

Théorème 5.1 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$ un ensemble de tâches qui satisfait la condition (3.12). Soit r_p la date de réception des données des tâches prédécesseurs de τ_i. La date de démarrage au plus tôt de τ_i après la date r_p est donnée par :*

$$s_i = s_i^0 + \left\lceil \frac{r_p - s_i^0}{g} \right\rceil g \tag{5.1}$$

avec $s_i^0 = \sum_{k=1}^{i-1} C_k$ et $g = PGCD(T_i,\ i = 1, n)$.

Preuve
La date de démarrage d'une tâche τ_i satisfaisant la condition (3.12) est donnée par la condition (3.13) : $s_i = s_i^0 + l \cdot g,\ l \in \mathbb{N}$.

La première instance qui s'exécute après la date r_p est la $\left\lceil \frac{r_p - s_i^0}{g} \right\rceil^{eme}$ instance. Sa date de début d'exécution est donnée par $s_i = s_i^0 + \left\lceil \frac{r_p - s_i^0}{g} \right\rceil g$. ∎

Le théorème suivant donne la date de démarrage au plus tôt d'une tâche τ_c qui satisfait la condition (3.14) pour laquelle r_p est la date de réception des données de ses prédécesseurs, sachant qu'un ensemble de tâches Γ_n satisfaisant la condition (3.12) est déjà ordonnancé.

Théorème 5.2 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$ un ensemble de tâches qui satisfait la condition (3.12) et τ_c une tâche qui satisfait la condition (3.14). Soit r_p la date de réception des données des tâches prédécesseurs de τ_c. La date de démarrage au plus tôt de τ_c après la date r_p est donnée par :*

$$s_c = min_{l=1..(\frac{T_i}{g}-1)} \left[max \left(r_p,\ s_i + l \cdot g + \left\lceil \frac{r_p - (s_i^0 + C_i - C_c + l \cdot g)}{T_i} \right\rceil T_i \right) \right] \tag{5.2}$$

Preuve
La date de démarrage d'une tâche τ_c satisfaisant la condition (3.14) est donnée par la condition (3.15) : $s_c = s_i + lg + kT_i + \alpha$ avec $k \in \mathbb{N},\ 1 \leq l \leq (\frac{T_i}{g} - 1)$ et $0 \leq \alpha \leq (C_i - C_c)$. Donc τ_c démarre au plus tard à

$$s_c = s_i + lg + kT_i + C_i - C_c.$$

La première instance de τ_c qui s'exécute après la date r_p est la

$$\left\lceil \frac{r_p - (s_i^0 + C_i - C_c + l \cdot g)}{T_i} \right\rceil^{eme}$$

instance qui démarre à

$$s_c^l = s_i + l \cdot g + C_i - C_c + \left\lceil \frac{r_p - (s_i^0 + C_i - C_c + l \cdot g)}{T_i} \right\rceil T_i$$

Comme cette instance a une marge de $C_i - C_c$ alors elle peut démarrer au plus tôt à

$$s_c^l = s_i + l \cdot g + \left\lceil \frac{r_p - (s_i^0 + C_i - C_c + l \cdot g)}{T_i} \right\rceil T_i$$

Cependant cette date peut être inférieure à r_p alors on doit prendre le maximum entre r_p et s_c^l donc

$$s_c^l = max\left(r_p,\ s_i + l \cdot g + \left\lceil \frac{r_p - (s_i^0 + C_i - C_c + l \cdot g)}{T_i} \right\rceil T_i\right).$$

Comme l peut prendre des valeurs de 1 à $(\frac{T_i}{g} - 1)$ alors s_c est obtenue en calculant le minimum des valeurs de s_c^l d'où la condition (5.2). ∎

Le théorème suivant donne la date de démarrage au plus tôt d'une tâche τ_c qui satisfait la condition (3.16) pour laquelle r_p est la date de réception des données de ses prédécesseurs, sachant qu'un ensemble de tâches Γ_n satisfaisant la condition (3.12) est déjà ordonnancé.

Théorème 5.3 *Soit $\Gamma_n = \{\tau_i(C_i, T_i), i = 1, n\}$ un ensemble de tâches qui satisfait la condition (3.12) et τ_c une tâche qui satisfait la condition (3.16). Soit r_p la date de réception des données des tâches prédécesseurs de τ_c. La date de démarrage au plus tôt de τ_c après la date r_p est donnée par :*

$$s_c = max\left[r_p,\ s_i + \left(2 \cdot \left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil + 1\right) \cdot g\right] \tag{5.3}$$

Preuve
La date de démarrage d'une tâche τ_c satisfaisant la condition (3.16) est donné par la condition (3.17) : $s_c = s_i + (2k+1)g + \alpha$ avec $k \in \mathbb{N}$ et $0 \leq \alpha \leq (C_i - C_c)$. Donc τ_c démarre au plus tard à

$$s_c = s_i + (2k+1)g + C_i - C_c = s_i + g + C_i - C_c + 2kg.$$

La première instance de τ_c qui s'exécute après la date r_p est la

$$\left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil^{eme}$$

instance qui démarre à

$$s_c^k = (s_i + g + C_i - C_c) + \left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil 2g.$$

Comme cette instance a une marge de $C_i - C_c$ alors elle peut démarrer au plus tôt à

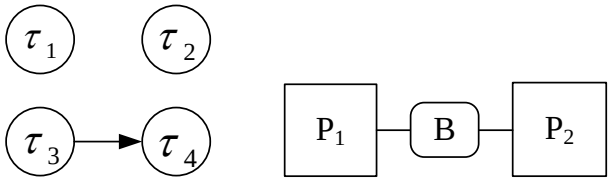
$$s_c^k = (s_i + g + C_i - C_c) + \left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil 2g - (C_i - C_c)$$

$$= s_i + \left(2 \cdot \left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil + 1 \right) \cdot g.$$

Cependant cette date peut être inférieure à r_p alors on doit prendre le maximum entre r_p et s_c^k et on obtient donc la condition (5.3). ∎

L'exemple suivant montre comment calculer les dates de début d'exécution pour quatre tâches.

Exemple
On considère un graphe d'algorithme Alg (figure 5.3(a)) constitué de quatre tâches $\tau_1(1,4)$, $\tau_2(1,5)$,$\tau_3(1,6)$ et $\tau_4(1,20)$ et un graphe d'architecture Arc (figure 5.3(b)) constitué de deux processeurs P_1 et P_2 et d'un bus de communication B pour lequel la durée de communication d'une donnée est $com = 3$.



(a) Graphe d'algorithme Alg (b) Graphe d'architecture Arc

FIGURE 5.3 – Graphe d'algorithme et d'un graphe d'architecture

On commence par ordonnancer la tâche τ_1 sur le processeur P_1. Lorsqu'on cherche à ordonnancer la tâche τ_2 sur P_1, la condition (3.12) n'est pas satisfaite donc τ_2 est ordonnancée sur le processeur P_2. τ_1 et τ_2 peuvent commencer leurs exécutions à $t = 0$ donc $s_1 = s_2 = 0$.

De la même manière les tâches τ_3 et τ_1 satisfont la condition (3.12) : $C_1 + C_2 = 2 \leq g_{1,2} = 2$. τ_3 est donc ordonnançable sur P_1 avec la date de début d'exécution s_3 qui est donnée par l'équation (3.13) : $s_3 = 1$.

Les tâches τ_4 et τ_1 satisfont la condition (3.16) : $1 \leq 1\delta(4mod4 + 20mod4) = \delta(0) = 1$. τ_4 est donc ordonnançable sur P_1. Comme τ_4 a une dépendance avec τ_2, la date de début d'exécution s_4 est donnée par l'équation (5.3) :

$$s_4 = max \left[r_p, 0 + 2 + \left\lceil \frac{r_p - (0 + 2 + 1 - 1)}{4} \right\rceil \cdot 4 \right]$$

$$= max\left[r_p, 2 + 4\left\lceil\frac{r_p - 2}{4}\right\rceil\right].$$

Comme $\frac{T_4}{T_2} = 4$ alors τ_4 démarre après la réceptions de 4 données de τ_2. On a donc $r_p = s_2 + 3T_2 + C_2 + com = 19$, d'où $s_4 = max\left[21, 2 + 4\left\lceil\frac{17}{4}\right\rceil\right] = max(21, 22) = 22$.

D'après le théorème 3.11 cet ordonnancement a une phase transitoire de longueur $\phi = \underset{i=1..4}{max}(0,\ s_i + C_i - T_i) = max(0, -3, -4, -4, 3) = 3$ et une phase permanente de longueur $L = PPCM(T_1, ..., T_4) = 60$.

La figure 5.3 montre l'ordonnancement de ces quatre tâches. On remarque que la tâche τ_4 a reçu les données de sa tâche prédécesseur τ_2 à l'instant $r_p = 19$ mais elle ne commence son exécution qu'à l'instant $s_4 = 22$. Si τ_4 avait commencé son exécution à $t = 21$ sans temps creux, sa troisième instance aurait chevauché la onzième instance de τ_2 à l'instant $t = 61$.

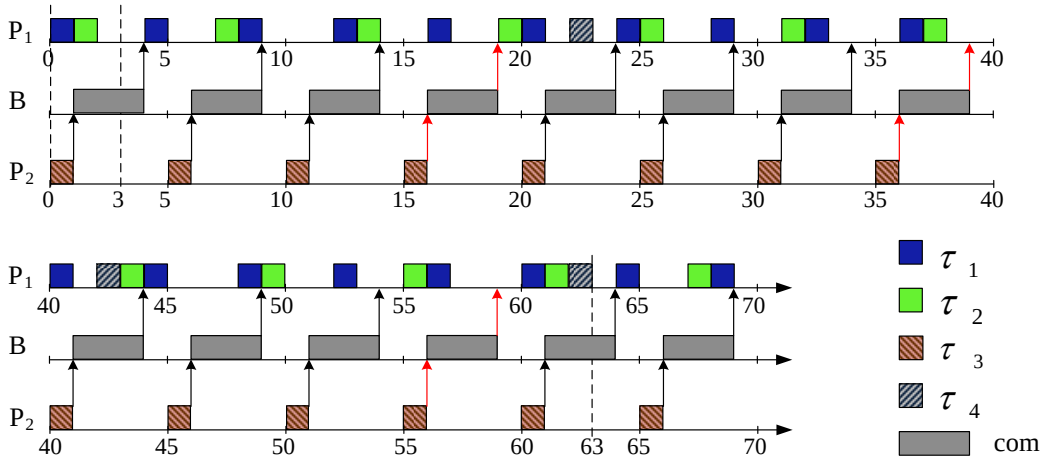


FIGURE 5.4 – Ordonnancement multiprocesseur avec prise en compte du coût de communication

5.4.2 Algorithme d'ordonnancement

Avant de présenter l'algorithme d'ordonnancement nous introduisons les notations et les définitions suivantes qui nous permettront de présenter la fonction de coût utilisée par cet algorithme donnée initialement dans [Sor94].

Notations

– $Exe(\tau_i, P_j)$: coût d'exécution de la tâche τ_i sur le processeur P_j,

- $s^{(n)}(\tau_i, P_j)$: date de début au plus tôt de la tâche τ_i sur le processeur P_j, depuis le début de l'ordonnancement [Sor94],
- $\bar{s}^{(n)}(\tau_i, P_j)$: date de début au plus tard de la tâche τ_i sur le processeur P_j, depuis le début de l'ordonnancement [Sor94].

L'algorithme d'ordonnancement est basé sur une fonction de coût appelée la *pression d'ordonnancement*, dont l'objectif est de minimiser la longueur du chemin critique. Nous définissions les notions suivantes permettant d'expliquer cette fonction de coût.

Définition 5.1 (Chemin critique) *Le chemin critique, noté $R^{(n)}(\tau_i, P_j)$, d'un graphe d'algorithme est le plus long chemin de ce graphe relativement aux coûts des exécutions Exe de chaque tâche de graphe d'algorithme sur chaque processeur de Arc.*

Définition 5.2 (Pénalité d'ordonnancement) *La pénalité d'ordonnancement, notée $P^{(n)}(\tau_i, P_j)$, est une fonction qui donne l'allongement du chemin critique $R^{(n)}(\tau_i, P_j)$ dû aux coûts des communications inter processeur, après avoir placé τ_i sur P_j à la n^{eme} itération de l'algorithme. Elle est définie par :*

$$P^{(n)}(\tau_i, P_j) = R^{(n)} - R^{(n-1)}$$

Définition 5.3 (Flexibilité d'ordonnancement) *La flexibilité d'ordonnancement, notée $F^{(n)}(\tau_i, P_j)$, est une fonction qui donne la marge d'ordonnancement de τ_i sur P_j à la n^eme itération de l'algorithme. Elle est définie par :*

$$F^{(n)}(\tau_i, P_j) = R^{(n)} - \bar{s}^{(n)}(\tau_i, P_j) - s^{(n)}(\tau_i, P_j)$$

Définition 5.4 (Pression d'ordonnancement) *La pression d'ordonnancement, notée $\sigma^{(n)}(\tau_i, P_j)$, est une fonction qui est la composition des fonctions flexibilité d'ordonnancement $F^{(n)}$ et allongement $P^{(n)}$ du chemin critique. Elle est calculée pour chaque tâche candidate τ_i sur chaque processeur P_j par*

$$\sigma^{(n)}(\tau_i, P_j) = P^{(n)}(\tau_i, P_j) - F^{(n)}(\tau_i, P_j)$$

La fonction de coût est basée sur une minimisation et une maximisation de la pression d'ordonnancement $\sigma^{(n)}(\tau_i, P_j)$. Plus de détails sur le calcul de cette fonction de coût se trouvent dans [Sor94, GLS99, Vic99, Gra00].

L'algorithme d'ordonnancement 7 consiste à ordonnancer l'ensemble Γ^* des tâches du graphe d'algorithme déroulé Alg^*, sur les m processeurs de l'architecture. On appelle $\Delta \subset \Gamma^*$ ensemble de tâches candidates dont toutes les tâches prédécesseurs ont déjà été ordonnancées.

Tant que l'ensemble Γ^* n'est pas vide, à chaque itération, l'ensemble Δ est mis à jour. Afin de respecter l'ordre partiel induit par les dépendances entre les tâches, l'algorithme commence (lignes $5-14$) par chercher pour chaque tâche de Δ le meilleur des processeurs, sur lequel elle a été assignée par l'algorithme d'analyse d'ordonnançabilité, en minimisant la fonction de coût pression d'ordonnancement, ce qui conduit à un ensemble de couples (tâche, meilleur processeur). Puis il cherche parmi ces couples le couple (tâche candidate τ_c^k, meilleur processeur P_j) qui maximise la fonction de coût pression d'ordonnancement. Maintenant l'algorithme peut calculer la date r_p de réception des données des prédécesseurs de la tâche candidate τ_c^k et ensuite calculer la date de début d'exécution de cette tâche sur son meilleur processeur P_j.

Si la tâche candidate τ_c^k est une première répétition τ_c^0 du graphe déroulé alors l'algorithme calcule sa date de début d'exécution selon les cas suivants :

– τ_c^0 vérifie la condition (3.12) :

$$\sum_{i=1}^{n} C_i \leq g$$

alors la date de début d'exécution est donnée par l'équation (5.1) :

$$s_c = s_i^0 + \left\lceil \frac{r_p - s_i^0}{g} \right\rceil g$$

– τ_c^0 vérifie la condition (3.14) :

$$C_c \leq C_i \cdot \delta\left(T_c mod(T_i)\right)$$

alors la date de début d'exécution est donnée par l'équation (5.2) :

$$s_c = min_{l=1..(\frac{T_i}{g}-1)} \left[max\left(r_p,\ s_i + lg + \left\lceil \frac{r_p - (s_i^0 + C_i - C_c + lg)}{T_i} \right\rceil T_i \right) \right]$$

– τ_c^0 vérifie la condition (3.16) :

$$C_c \leq C_i \cdot \delta\left(T_c \cdot mod(2g) + T_i \cdot mod(2g)\right)$$

alors la date de début d'exécution est donnée par l'équation (5.3) :

$$s_c = max\left[r_p,\ s_i + g + \left\lceil \frac{r_p - (s_i^0 + g + C_i - C_c)}{2g} \right\rceil \cdot 2g \right]$$

Il calcule ensuite les dates de début d'exécution des autres répétitions τ_c^k : $s_c^k = s_c + kT_c$.

Algorithme 7 Algorithme d'ordonnancement

initialisation de l'ensemble Γ^* des tâches du graphe d'algorithme déroulé
initialisation de l'ensemble P des processeurs du graphe d'architecture
tant que Γ^* n'est pas vide **faire**
  construire l'ensemble des tâches candidates Δ dont les prédécesseurs ont été ordonnancées
  pour chaque tâche $\tau_c^k \in \Delta$ **faire**
    pour chaque processeur P_j de l'ensemble des processeurs sur lesquels τ_c^k a été assignée **faire**
      calculer la fonction de coût pression d'ordonnancement $\sigma(\tau_c^k, P_j)$
    fin pour
    choisir le processeur P_j qui minimise $\sigma(\tau_c^k, P_j)$
    construire la paire (tâche τ_c^k, meilleur processeur P_j)
  fin pour
  choisir parmi ces paires la paire $(\tau_c^k,\ P_j)$ qui maximise $\sigma(\tau_c^k, P_j)$
  calculer la date r_p de réception des données de ses tâches prédécesseurs
  si la tâche sélectionnée est une première répétition τ_c^0 **alors**
    si cette tâche vérifie la condition (3.12) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.1)
    sinon si elle vérifie la condition (3.14) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.2)
    sinon si elle vérifie la condition (3.16) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.3)
    fin si
    ordonnancer cette tâche sur ce processeur P_j
    calculer les dates de début d'exécution des autres répétitions τ_c^k : $s_c^k = s_c + kT_c$
    enlever de l'ensemble Γ^* la tâche τ_c^0 qui vient d'être ordonnancée
  sinon
    si $r_p < s_c^k$, (s_c^k calculé à l'étape 23) **alors**
      ordonnancer cette tâche sur ce processeur P_j avec la date de début d'exécution s_c^k
      enlever de l'ensemble Γ^* la tâche τ_c^0
    sinon
      τ_c^k n'est pas ordonnançable, Γ^* n'est pas ordonnançable, fin de l'algorithme
    fin si
  fin si
fin tant que

Si la tâche candidate τ_c^k n'est pas une première répétition alors l'algorithme vérifie que la date r_p de réception des données des prédécesseurs est inférieure à sa date de début d'exécution s_c^k calculée précédemment. Dans le cas où cette condition n'est pas vérifiée la tâche n'est pas ordonnançable et par conséquent l'ensemble de tâches Γ^* n'est pas ordonnançable.

La tâche candidate ordonnancée est maintenant supprimée de l'ensemble de tâches Γ^*.

5.5 Conclusion

Nous avons présenté dans ce chapitre l'analyse d'ordonnancement multiprocesseur avec une approche d'ordonnancement partitionnée. Cette analyse d'ordonnancement est fondée sur l'analyse d'ordonnancement monoprocesseur présentée dans le chapitre 3. L'ordonnancement d'un système de tâches multiprocesseur est formé de trois algorithmes qui s'exécutent dans l'ordre suivant. L'algorithme d'analyse d'ordonnançabilité permet de faire une étude d'ordonnançabilité monoprocesseur et d'assigner chaque tâche sur les processeurs sur lesquels cette dernière est ordonnançable. L'algorithme de déroulement permet de transformer le graphe d'algorithme en un graphe déroulé où chaque tâche est répétée un nombre de fois égal au rapport entre l'hyper-période et sa période. L'algorithme d'ordonnancement exploite les résultats des deux algorithmes précédents pour choisir sur quel processeur ordonnancer une tâche assignée à plusieurs processeurs et calculer sa date de début d'exécution en prenant en compte des dates de communications inter-processeurs.

Nous présentons dans le chapitre suivant une étude d'ordonnançabilité multiprocesseur tolérante aux fautes.

Chapitre 6

Ordonnancement temps réel multiprocesseur non préemptif périodique strict tolérant aux fautes

6.1 Introduction

Nous avons présenté dans le chapitre précédent une étude d'ordonnancement temps réel multiprocesseur de tâches NPPS. Ces tâches qui sont les composants logiciels évoqués dans le chapitre 2 de l'état de l'art sur la tolérance aux fautes, correspondent aux tâches critiques d'un système temps réel où le non respect des contraintes temporelles (échéance, latence, etc.) peut avoir des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines). Notre étude d'ordonnançabilité garantit que toutes les tâches respectent leurs échéances temporelles sous réserve que les composants matériels d'un tel système soient fiables. Si l'un ou plusieurs composants matériels défaillent, l'ordonnancement précédent ne permet pas de garantir que les contraintes temporelles sont respectées.

Nous nous intéressons dans ce chapitre à l'étude d'ordonnancement temps réel multiprocesseur tolérant aux fautes des processeurs et des bus de communication, qui permet de garantir que les contraintes temporelles soient respectées même en présence de fautes matérielles.

Nous présentons tout d'abord les terminologies liées à la tolérance aux fautes et le modèle de fautes. Ensuite nous présentons la transformation du graphe d'algorithme initial pour la tolérance aux fautes. Enfin nous présentons l'étude d'ordonnancement tolérant aux fautes qui est une extension de celle présentée dans le chapitre 5.

Dans ce chapitre nous considérons un graphe d'algorithme Alg constitué d'un ensemble Γ de n tâches dépendantes, et une architecture Arc de m processeurs et

l bus de communication.

6.2 Présentation du problème de tolérance aux fautes

Nous nous sommes inspirés du travail présenté par H. Kalla présenté dans [Kal04] chapitre 7, sur la tolérance aux fautes des architecture à base de bus de communication. L'approche utilisée est la redondance passive pour les tâches de calcul et active pour les dépendances de données avec une fragmentation de données de communication. Nous rappelons que ces travaux ont été effectués dans le cas monopériode.

Dans le cas multipériode, la redondance passive ne permet pas de prédire le comportement temps réel du système, et par conséquent de garantir que le système reste toujours ordonnançable en présence de fautes. C'est pourquoi nous avons opté pour la redondance active des tâches et des dépendances de données sans fragmentation de données.

6.2.1 Modèle de fautes

Dans notre modèle de fautes nous supposons que :

Hypothèse 6.1 *Les actionneurs et les capteurs sont fiables.*

Hypothèse 6.2 *Les fautes matérielles sont des fautes de processeurs et de bus de communication.*

Chaque machine séquentielle de communication ainsi que sa mémoire distribuée associée (cf. section 1.3.2) est appelée dans la suite "communicateur".

Hypothèse 6.3 *La défaillance d'un bus de communication peut être* partielle *ou* complète*. Un bus de communication est à défaillance complète si tous ses communicateurs sont défaillants. Un bus de communication est à défaillance partielle si certains de ses communicateurs sont défaillants et au moins deux de ces communicateurs restent actifs.*

La figure 6.1 montre une architecture à base de quatre processeurs P_i et de deux bus de communication Bus_i. Chaque processeur P_i contient une opération de calcul Op_i et deux communicateurs Com_i^1 et Com_i^2 qui sont connectés respectivement aux bus Bus_1 et Bus_2. La figure 6.1 montre tous les communicateurs connectés au Bus_1 sont défaillant et par conséquent ce bus a une défaillance complète. Concernant le Bus_2, seulement le connecteur Com_1^2 est défaillant et par conséquent ce bus a une défaillance partielle.

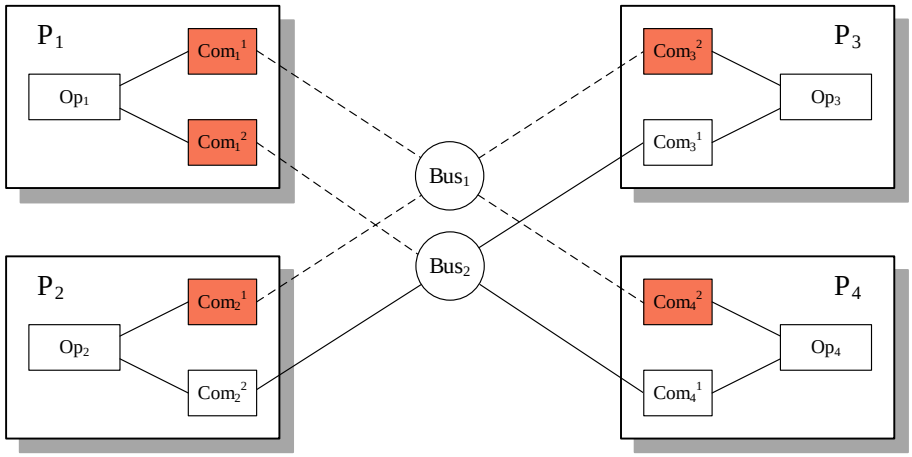


FIGURE 6.1 – Défaillance complète du bus 1 et partielle du bus 2

Hypothèse 6.4 *On tolère au plus N_{pf} fautes de processeurs et N_{bf} fautes de bus de communication.*

Hypothèse 6.5 *Les fautes de processeurs et de bus de communication sont des fautes permanentes.*

Hypothèse 6.6 *Les processeurs et les bus de communication sont à défaillances temporelles, i.e. les valeurs calculées par les processeurs sont correctes mais peuvent être délivrées à temps, trop tôt, trop tard ou infiniment tard.*

6.2.2 Données du problème de tolérance aux fautes

Les données du problème de tolérance aux fautes que nous considérons sont les suivantes :

- une architecture hétérogène *Arc* multi-bus composée d'un ensemble P de m processeurs et d'un ensemble B de n bus de communication :

$$P = \{P_1, ..., p_m\},\ B = \{b_1, ..., b_n\};$$

- un algorithme Alg composé d'un ensemble O de tâches et d'un ensemble E de dépendances de données :

$$O = \{..., \tau_i, ..., \tau_j, ...\},\ E = \{..., (\tau_i \triangleright \tau_j), ...\};$$

- des caractéristiques d'exécution Exe des composants de Alg sur les composants de Arc,
- un nombre N_{pf} de processeurs et un nombre N_{bf} de bus de communication pouvant être fautifs.
- une fonction de coût pression d'ordonnancement,

6.2.3 Notations et définitions

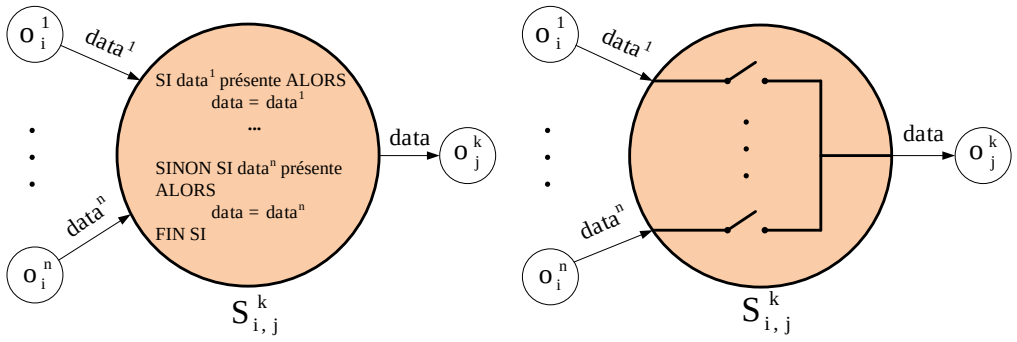
Nous présentons dans cette section quelques définitions et notations que nous utilisons par la suite.

Définition 6.1 (Tâche réplique) *La tâche $\tau_{i,k}$ représente la k^{eme} tâche réplique de τ_i. Toutes les tâches répliques d'une même tâche sont identiques, i.e. elles contiennent le même code que τ_i donc elles ont la même durée. Il faut noter que lorsqu'on réplique une tâche τ_i, on crée $N_{pf}+1$ tâches répliques $\tau_{i,k}$ sans garder la tâche τ_i.*

Attention il ne faut pas confondre la k^{eme} tâche réplique $\tau_{i,k}$ d'une tâche τ_i avec la k^{eme} instance τ_i^k (voir modèle de tâches section 1.2.1.1).

Définition 6.2 (Tâche de sélection) *Une tâche de sélection $S_{i,j}^k$ est une tâche caractérisée par ses dépendances de données d'entrées provenant de toutes les tâches répliques $\tau_{i,l}$ du graphe d'algorithme Alg, et par son unique dépendance de données de sortie $\tau_{j,k}$. Son rôle est de transmettre une des données d'entrée à sa sortie. Elle a une durée d'exécution non nulle.*

La figure 6.2 montre une représentation informatique et graphique d'une tâche de sélection $S_{i,j}^k$. La figure 6.2(a) est une représentation informatique d'un sélecteur $S_{i,j}^k$. Le principe est de vérifier la présence d'une donnée $data^k$ de la dépendance $(\tau_i \triangleright \tau_i)$. Dès qu'une donnée $data^k$ est présente alors la sortie du sélecteur prend la valeur de cette entrée : $data = data^k$. La figure 6.2(b) est une représentation graphique de ce sélecteur $S_{i,j}^k$.



(a) Représentation informatique de $S_{i,j}^k$ (b) Représentation graphique de $S_{i,j}^k$

FIGURE 6.2 – Représentations d'une tâche de sélection $S_{i,j}^k$

Définition 6.3 (Tâche exclusives) *Deux tâches $\tau_{i,k}$ et $\tau_{i,l}$ sont dites exclusives si et seulement si elles sont deux répliques identiques d'une même tâche τ_i. Elles sont*

par conséquent assignées à deux processeurs distincts. Cette relation d'exclusion est notée $\| \tau_{i,k}, \tau_{i,l} \|$.

Définition 6.4 (Dépendances exclusives) *Deux dépendances de données* $(\tau_{i,l} \triangleright \tau_{j,k})$ *et* $(\tau_{i,m} \triangleright \tau_{j,k})$ *sont exclusives si et seulement si elles sont deux répliques identiques d'une même dépendance* $(\tau_i \triangleright \tau_j)$ *et sont placées sur deux média disjoints. Les dépendances exclusives sont notées par* $\| (\tau_{i,l} \triangleright \tau_{j,k}), (\tau_{i,m} \triangleright \tau_{j,k}) \|$.

6.3 Transformation de graphe pour la tolérance aux fautes

Dans cette section nous considérons un graphe d'algorithme Alg de n tâches dépendantes et une architecture Arc de m processeurs et l bus de communication. Pour simplifier les notations nous appelons N_p le nombre de tâches répliques donné par $N_p = N_{pf} + 1$ et N_b le nombre de tâches répliques donné par $N_b = N_{bf} + 1$, avec $N_p \leq m$ et $N_b \leq l$. Lorsqu'on réplique une tâche τ_i, on crée N_p tâches répliques $\tau_{i,k}$ et on rappelle qu'on ne garde pas la tâche τ_i. Par exemple si on veut tolérer une faute de processeurs, il faut créer 2 répliques de la tâche considérée.

Nous présentons les transformations de graphe pour la tolérance aux fautes des processeurs seulement, pour les bus de communication seulement, ensuite pour les processeurs et les bus de communication. Nous illustrons toutes les transformations de graphes avec un exemple simple de graphe d'algorithme composé d'un couple de deux tâches dépendantes (figure 6.3(b)). Pour la transformation d'un graphe d'algorithme quelconque, nous suivons les mêmes étapes en raisonnant sur chaque couple de deux tâches dépendantes.

6.3.1 Tolérance aux fautes des processeurs

Dans ce cas nous supposons que les bus de communication sont fiables et seuls les processeurs peuvent être fautifs. La transformation du graphe d'algorithme Alg en un graphe transformé Alg^* a pour but de tolérer N_{pf} fautes de processeurs.

Pour cela, chaque tâche de Alg doit avoir $(N_p = N_{pf} + 1)$ tâches répliques assignées à $(N_b = N_{pf} + 1)$ processeur distincts de Arc.

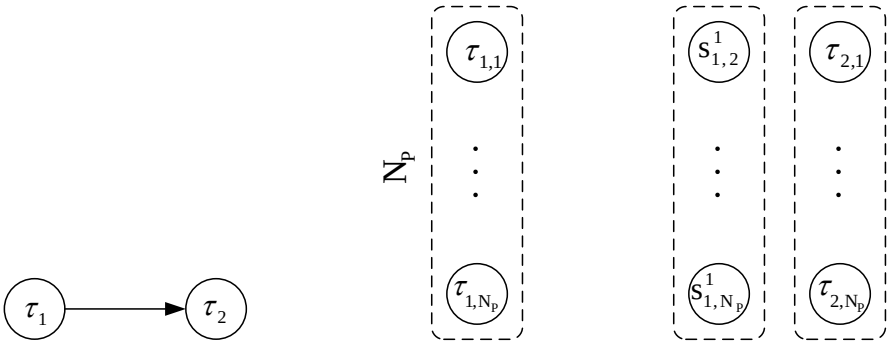
Nous procédons comme suit :

1. répliquer chaque tâche τ_i de Alg en N_p tâches répliques exclusives $\tau_{i,k}$. L'ensemble de ces tâches répliques est noté $Rep(\tau_i) = \{\tau_{i,1}, ..., \tau_{i,N_p}\}$ (figure 6.3(b)) ;

2. créer N_p tâches de sélection $S^k_{i,j}$ pour chaque couple de tâches dépendantes (τ_i, τ_j) de Alg. L'ensemble des tâches de sélection est noté $Sel(\tau_i, \tau_j) = \{S^1_{i,j}, ..., S^{N_p}_{i,j}\}$ (figure 6.3(b)) ;
3. créer une dépendance de donnée $(\tau_{i,k} \triangleright S^l_{i,j})$ entre chaque tâche réplique $\tau_{i,k}$ et chaque tâche de sélection $S^l_{i,j}$ (figure 6.3(c)),
4. créer une dépendance de donnée $(S^k_{i,j} \triangleright \tau_{j,k})$ entre chaque tâche de sélection $S^k_{i,j}$ et chaque tâche réplique $\tau_{j,k}$ (figure 6.3(c)).

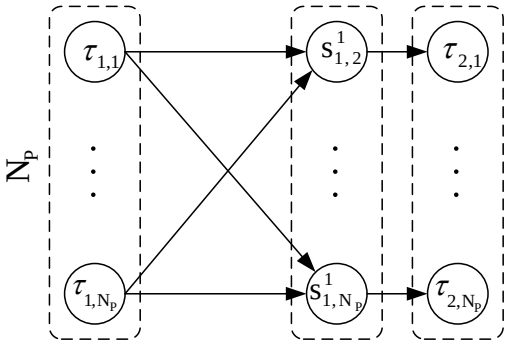
Cette transformation de graphe génère l'ensemble des exclusions entre les tâches suivant :

$$Excl_{op} = \{||\tau_{i,k}, \tau_{i,l}||,\ k \neq l\}.$$



(a) Graphe d'algorithme Alg

(b) Réplication de tâches



(c) Réplication de dépendances

FIGURE 6.3 – Transformation de graphe Alg^*

6.3.2 Tolérance aux fautes des bus de communication

Dans ce cas nous supposons que les processeurs sont fiables et seuls les bus de communication peuvent être fautifs. La transformation du graphe d'algorithme

Alg en un graphe transformé Alg^* a pour but de tolérer N_{bf} fautes de bus de communication.

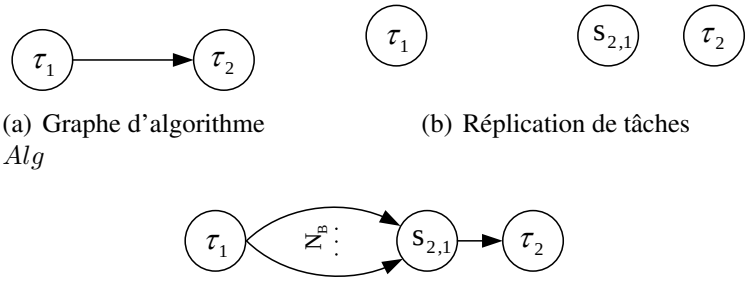
Pour cela, chaque dépendance de données $(\tau_i \triangleright \tau_j)$ de Alg doit avoir $(N_b = N_{bf}+1)$ dépendances répliques assignées à $(N_b = N_{bf}+1)$ bus de communication distincts de Arc.

Nous procédons comme suit :

1. créer une tâche de sélection $S_{i,j}$ pour chaque couple de tâches dépendantes (τ_i, τ_j) de Alg (figure 6.4(b)),
2. créer une dépendance de donnée $(S_{i,j} \triangleright \tau_j)$ entre la tâche de sélection $S_{i,j}$ et la tâche τ_j (figure 6.4(c)),
3. créer N_b dépendances de données exclusives $(\tau_i \triangleright S_{i,j})^k$ de Alg entre la tâche τ_i et la tâche de sélection $S_{i,j}$ (figure 6.4(c)).

Cette transformation de graphe génère l'ensemble des exclusions entre dépendances de données suivant :

$$Excl_{dep} = \{||(\tau_i \triangleright S_{i,j})^k, (\tau_i \triangleright S_{i,j})^l||,\ k \neq l\}.$$



(a) Graphe d'algorithme Alg

(b) Réplication de tâches

(c) Réplication de dépendances

FIGURE 6.4 – Transformation de graphe Alg^*

6.3.3 Tolérance aux fautes des processeurs et des bus de communication

Dans ce cas nous supposons que les processeurs et les bus de communication peuvent être fautifs. La transformation du graphe d'algorithme Alg en un graphe transformé Alg^* a pour but de tolérer N_{pf} fautes de processeurs et N_{bf} fautes de bus de communication.

Pour ce faire, chaque tâche de Alg doit avoir $(N_p = N_{pf} + 1)$ tâches répliques assignées à $(N_b = N_{pf} + 1)$ processeurs distincts de Arc et chaque dépendance

de données $(\tau_i \triangleright \tau_j)$ de Alg doit avoir $(N_b = N_{bf} + 1)$ dépendances répliques assignées à $(N_b = N_{bf} + 1)$ bus de communication distincts de Arc.

Nous procédons comme suit :

1. répliquer chaque tâche τ_i de Alg en N_p tâches répliques exclusives $\tau_{i,k}$. L'ensemble de ces tâches répliques est noté $Rep(\tau_i) = \{\tau_{i,1}, ..., \tau_{i,N_p}\}$ (figure 6.5(b)) ;

2. créer N_p tâches de sélection $S^k_{i,j}$ pour chaque couple de tâches dépendantes (τ_i, τ_j) de Alg. L'ensemble des tâches de sélection est noté $Sel(\tau_i, \tau_j) = \{S^1_{i,j}, ..., S^{N_p}_{i,j}\}$ (figure 6.5(c)) ;

3. créer une dépendance de donnée $(\tau_{i,k} \triangleright S^l_{i,j})$ entre chaque tâche réplique $\tau_{i,k}$ et chaque tâche de sélection $S^l_{i,j}$ (figure 6.5(c)),

4. créer une dépendance de donnée $(S^k_{i,j} \triangleright \tau_{j,k})$ entre chaque tâche de sélection $S^k_{i,j}$ et chaque tâche réplique $\tau_{j,k}$ (figure 6.5(c)),

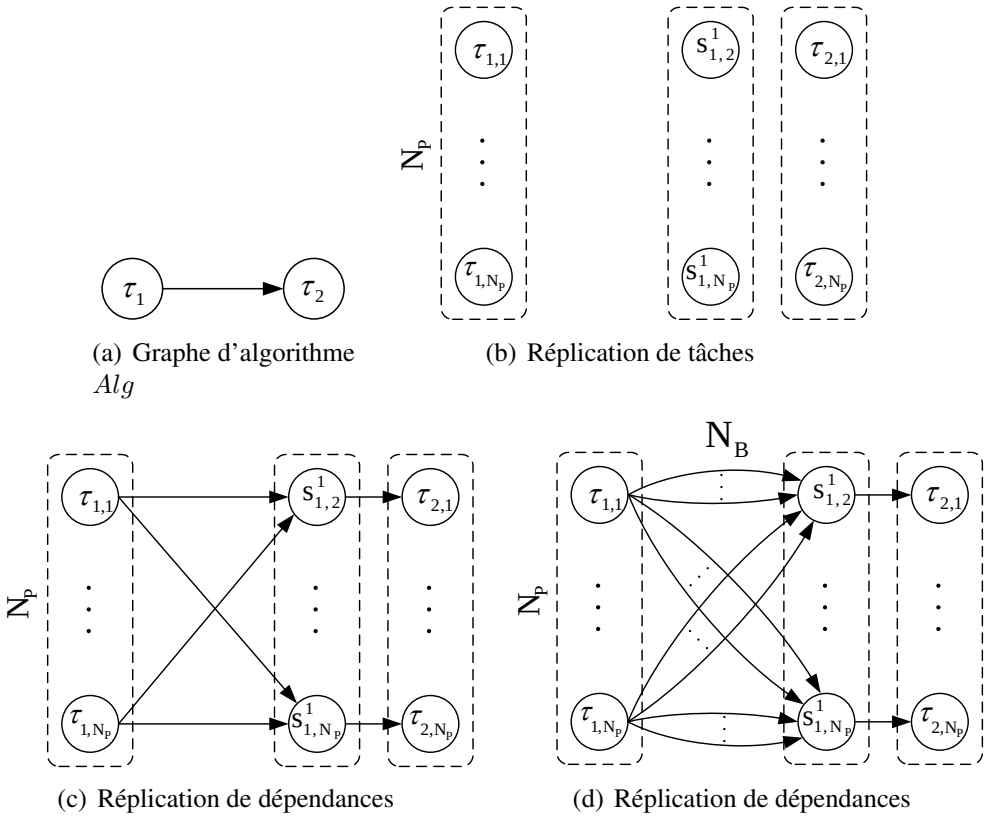
5. répliquer chaque dépendance de donnée $(\tau_{j,k} \triangleright S^k_{i,j})$ en N_b dépendances de données exclusives $(\tau_{j,k} \triangleright S^l_{i,j})^m$ (figure 6.5(d)).

Cette transformation de graphe génère l'ensemble des exclusions entre les tâches suivant :

$$Excl_{op} = \{||\tau_{i,k}, \tau_{i,l}||,\ k \neq l\}$$

et l'ensemble des exclusions entre dépendances de données suivant :

$$Excl_{dep} = \{||(\tau_{i,k} \triangleright S^l_{i,j})^m, (\tau_{i,k} \triangleright S^l_{i,j})^n||,\ m \neq n\}.$$



(a) Graphe d'algorithme Alg

(b) Réplication de tâches

(c) Réplication de dépendances

(d) Réplication de dépendances

FIGURE 6.5 – Transformation de graphe Alg^*

La transformation de graphe pour la tolérance de N_{pf} fautes de processeurs et de N_{bf} fautes de bus de communication est décrite par l'algorithme 8.

6.4 Ordonnancement tolérant aux fautes

Après avoir transformé le graphe d'algorithme Alg en Alg^* pour la tolérance aux fautes, nous nous intéressons maintenant à l'ordonnancement des tâches du nouveau graphe transformé. Comme nous l'avons montré dans le chapitre 5, l'ordonnancement des tâches d'un graphe d'algorithme est formé de trois algorithmes qui s'exécutent dans l'ordre suivant : (i) algorithme d'analyse d'ordonnançabilité, (ii) algorithme de déroulement, (iii) algorithme d'ordonnancement. L'algorithme d'ordonnancement tolérant aux fautes est celui présenté dans le chapitre 5 étendu pour prendre en compte les relations d'exclusion des tâches et les dépendances de données répliques. Par conséquent le nombre de tâches du graphe d'algorithme transformé est augmenté des tâches répliques, des tâches de sélection et des dépen-

Algorithme 8 Algorithme de transformation de graphe pour la tolérance aux fautes

1: Initialisation du graphe d'algorithme Alg et du nombre de fautes de processeurs (resp. bus de communication) tolérées N_{pf} (resp. N_{bf})
2: répliquer chaque tâche τ_i de Alg en N_p tâches répliques exclusives $\tau_{i,k}$
3: créer N_p tâches de sélection $S^k_{i,j}$ pour chaque couple de tâches dépendantes (τ_i, τ_j) de Alg
4: créer une dépendance de donnée $(\tau_{i,k} \triangleright S^l_{i,j})$ entre chaque tâche réplique $\tau_{i,k}$ et chaque tâche de sélection $S^l_{i,j}$
5: créer une dépendance de donnée $(S^k_{i,j} \triangleright \tau_{j,k})$ entre chaque tâche de sélection $S^k_{i,j}$ et chaque tâche réplique $\tau_{j,k}$
6: répliquer chaque dépendance de donnée $(\tau_{j,k} \triangleright S^k_{i,j})$ en N_b dépendances de données exclusives $(\tau_{j,k} \triangleright S^l_{i,j})^m$
7: générer l'ensemble des exclusions entre les tâches $Excl_{op}$ et l'ensemble des exclusions entre dépendances de données $Excl_{dep}$

dances correspondantes. L'algorithme d'analyse d'ordonnançabilité doit prendre en compte les relations d'exclusion pour les tâches répliques afin de ne pas assigner au même processeur deux tâches exclusives. De même l'algorithme d'ordonnancement doit prendre en compte les relations d'exclusion pour les dépendances de données afin de ne pas assigner au même bus de communication deux dépendances de données exclusives. Cependant l'algorithme de déroulement reste inchangé puisque son rôle consiste seulement à répliquer les tâches selon le rapport entre l'hyper-période et leurs périodes sans les assigner. Nous présentons dans ce qui suit les algorithmes d'analyse d'ordonnançabilité et d'ordonnancement tolérants aux fautes.

6.4.1 Algorithme d'analyse d'ordonnançabilité

Nous avons présenté dans la section 5.4 l'étude d'ordonnançabilité multiprocesseur non tolérante aux fautes. Pour étendre cette étude à la tolérance aux fautes nous devons prendre en compte les considérations suivantes :

- deux tâches exclusives ne doivent pas être assignées au même processeur,
- chaque tâche de sélection $S^k_{i,j}$ est assignée au même processeur que la tâche réplique τ^k_j.

Les relations d'exclusions des tâches répliques restreignent le nombre d'assignation de ces dernières dans la mesure où lorsqu'une tâche réplique $\tau_{i,j}$ est assignée à plusieurs processeurs, toutes les tâches répliques $\tau_{i,k}$ exclusives à $\tau_{i,j}$ ne doivent pas être assignées à ces dernier processeurs.

Par exemple si la première tâche réplique $\tau_{i,1}$ est assignée à au moins à $m - N_p + 2$ processeurs, m étant le nombre total des processeurs, alors il reste $N_p - 2$ processeurs pour assigner les $N_p - 1$ tâches répliques. Par conséquent ces tâches ne sont pas ordonnançables.

Pour ces raisons nous avons choisi d'assigner chaque tâche réplique à un seul processeur et dès que cette dernière est assignée on passe à la tâche réplique suivante, ce qui correspond maintenant à utiliser un algorithme d'analyse d'ordonnançabilité de type "First-Fit" au lieu de "First-Fit étendu" comme dans la section 5.2.

La transformation de graphe pour la tolérance aux fautes produit le graphe d'algorithme Alg^* constitué de l'ensemble Γ^* contenant les tâches répliques et les tâches de sélection et de toutes les dépendances correspondantes. L'algorithme 9 illustre les étapes de l'analyse d'ordonnançabilité.

Algorithme 9 Algorithme d'analyse d'ordonnançabilité tolérant aux fautes

Trier les tâches répliques et les tâches de sélection de Γ^* selon les périodicités de leurs périodes
pour chaque tâche réplique $\tau_{i,k}$ de Γ^* **faire**
  pour chaque processeur P_j **faire**
    si il n'existe aucune tâche réplique exclusive $\tau_{i,l}$ assignée à ce processeur et si la condition (3.12), (3.14) ou (3.16) est satisfaite **alors**
      assigner cette tâche à ce processeur
      sortir de la boucle et passer à la tâche suivante
    sinon
      passer au processeur suivant
    fin si
  fin pour
  si la tâche n'est assignée à aucun processeur on dit que le système de tâches n'est pas ordonnançable et on arrête l'algorithme
fin pour
pour chaque tâche de sélection $S^k_{i,j}$ **faire**
  pour chaque processeur P_j **faire**
    si la tâche réplique $\tau_{j,k}$ est assignée à ce processeur **alors**
      assigner $S^k_{i,j}$ à ce processeur
    sinon
      passer au processeur suivant
    fin si
  fin pour
fin pour

6.4.2 Algorithme de déroulement

L'algorithme de déroulement 6 reste le même dans le cas du déroulement tolérant aux fautes. Cependant cet algorithme ne prend pas en entrée le graphe d'algorithme Alg mais le graphe transformé Alg^*. Toutes les tâches de Alg, y compris les tâches de sélection, seront dupliquées selon le rapport entre l'hyper-période et leurs périodes respectives. Le graphe obtenu est noté Alg^{**} avec comme ensemble des tâches Γ^{**}. L'algorithme 10 d'ordonnancement présenté ci-dessous utilisera le graphe d'algorithme Alg^{**}.

6.4.3 Algorithme d'ordonnancement

L'algorithme 10 d'ordonnancement tolérant aux fautes est fondé sur l'algorithme 7 en prenant en compte les relations d'exclusion des dépendances de données. Il consiste à ordonnancer l'ensemble Γ^{**} des tâches du graphe d'algorithme répliqué et déroulé Alg^{**}, sur les m processeurs de l'architecture. On appelle $\Delta \subset \Gamma^{**}$ ensemble de tâches candidate dont toutes les tâches prédécesseures ont déjà été ordonnancées.

Tant que l'ensemble Γ^{**} n'est pas vide, à chaque itération, l'ensemble Δ est mis à jour. Comme chaque tâche n'est assignée qu'à un seul processeur, il n'y a pas lieu de faire une minimisation de la fonction de coût pression d'ordonnancement afin d'obtenir les couples (tâche, meilleur processeur). Il suffit juste de choisir parmi les tâches de l'ensemble Δ celle qui maximise la fonction de coût pression d'ordonnancement. Maintenant l'algorithme peut calculer la date r_p de réception des données des prédécesseures de la tâche candidate τ_c^k en respectant les relations d'exclusion des dépendances de données. Ensuite il calcule la date de début d'exécution de cette tâche sur le processeur P_j auquel elle est assignée.

Si la tâche candidate τ_c^k est une première répétition τ_c^0 du graphe déroulé alors l'algorithme calcule sa date de début d'exécution selon les conditions (5.1), (5.2) ou (5.3) (cf. section 5.4.2).

Si la tâche candidate τ_c^k n'est pas une première répétition alors l'algorithme vérifie que la date r_p de réception des données des prédécesseures est inférieure à sa date de début d'exécution s_c^k calculée précédemment. Dans le cas où cette condition n'est pas vérifiée la tâche n'est pas ordonnançable et par conséquent l'ensemble de tâches Γ^{**} n'est pas ordonnançable.

La tâche candidate ordonnancée est maintenant supprimée de l'ensemble de tâches Γ^{**}.

Algorithme 10 Algorithme d'ordonnancement tolérant aux fautes

initialisation de l'ensemble Γ^{**} des tâches du graphe d'algorithme déroulé
initialisation de l'ensemble P des processeurs du graphe d'architecture
tant que Γ^{**} n'est pas vide **faire**
  construire l'ensemble des tâches candidates Δ dont les prédécesseures ont été ordonnancées
  pour chaque tâche $\tau_c^k \in \Delta$ **faire**
    calculer la fonction de coût pression d'ordonnancement $\sigma(\tau_c^k, P_j)$ de la tâche τ_c^k sur le processeur P_j auquel elle a été assignée
  fin pour
  choisir parmi ces tâches celle qui maximise $\sigma(\tau_c^k, P_j)$
  calculer la date r_p de réception des données de ces tâches prédécesseures en prenant compte des relations d'exclusion des dépendances de données
  si la tâche sélectionnée est une première répétition τ_c^0 **alors**
    si cette tâche vérifie la condition (3.12) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.1)
    sinon si elle vérifie la condition (3.14) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.2)
    sinon si elle vérifie la condition (3.16) **alors**
      calculer sa date de début d'exécution s_c avec la condition (5.3)
    fin si
    ordonnancer cette tâche sur ce processeur P_j
    calculer les dates de début d'exécution des autres répétitions τ_c^k : $s_c^k = s_c + kT_c$
    enlever de l'ensemble Γ^{**} la tâche τ_c^0 qui vient d'être ordonnancée
  sinon
    si $r_p < s_c^k$, (s_c^k calculé à l'étape 19) **alors**
      ordonnancer cette tâche sur ce processeur P_j avec la date de début d'exécution s_c^k
      enlever de l'ensemble Γ^{**} la tâche τ_c^0
    sinon
      τ_c^k n'est pas ordonnançable, Γ^{**} n'est pas ordonnançable, fin de l'algorithme
    fin si
  fin si
fin tant que

6.5 Conclusion

Nous avons présenté dans ce chapitre l'étude d'ordonnancement temps réel multiprocesseur tolérant aux fautes des processeurs et de bus de communication. Nous avons commencé par présenter le modèle de fautes et les hypothèses de tolérance aux fautes que nous considérons. Puis nous avons présenté la transformation du graphe d'algorithme pour la tolérance aux fautes qui ajoute des tâches et des dépendances de données répliques, des tâches de sélection ainsi que des relations d'exclusion. Nous avons étudié séparément les problèmes de tolérance aux fautes pour des processeurs, des bus de communication et enfin des processeurs et des bus de communication. Ensuite nous avons présenté l'ordonnancement tolérant aux fautes qui est composé de trois algorithmes : l'algorithme d'analyse d'ordonnançabilité, de déroulement et d'ordonnancement. Afin de prendre en compte les relations d'exclusions générées par la transformation de graphe, nous avons modifié l'algorithme d'analyse d'ordonnançabilité et l'algorithme d'ordonnancement.

Nous présentons dans le chapitre suivant les développements logiciels, dans le logiciel SynDEx, correspondant aux travaux présentés dans les chapitres précédents.

Quatrième partie

Développements logiciels et application au CyCab

Chapitre 7

Développements du logiciel SynDEx

7.1 Méthodologie AAA et principes de SynDEx

L'appellation AAA [Sor94] (Adéquation Algorithme Architecture) regroupe un ensemble de recherches menées dans le but de développer des méthodes permettant de réaliser une implantation distribuée optimisée d'un algorithme, tout en respectant des contraintes de temps réel, de précédences, etc. La figure 7.1 montre les éléments de la méthodologie AAA : l'algorithme, l'architecture, l'adéquation qui effectue sur ces précédents éléments une analyse d'ordonnançabilité et des optimisation et produit un diagramme temporel décrivant par processeur (resp. médium de communication) les dates d'exécution des opérations (resp. des transfert de données entre opérations).

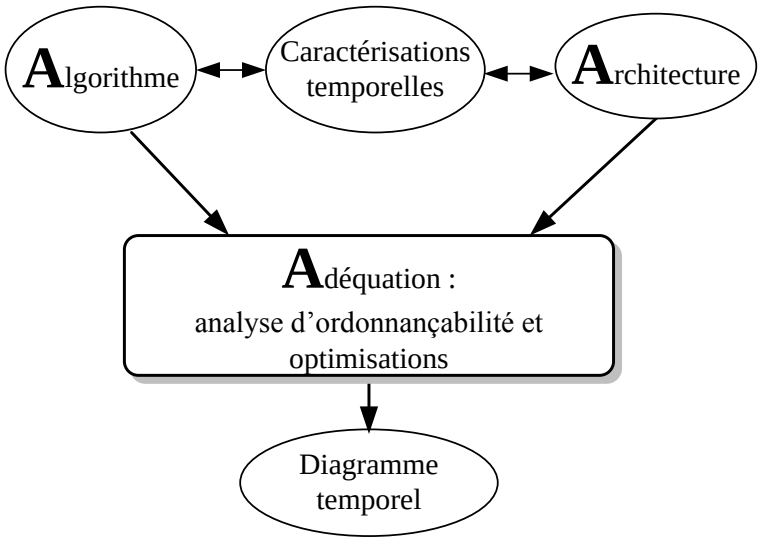


FIGURE 7.1 – Méthodologie AAA

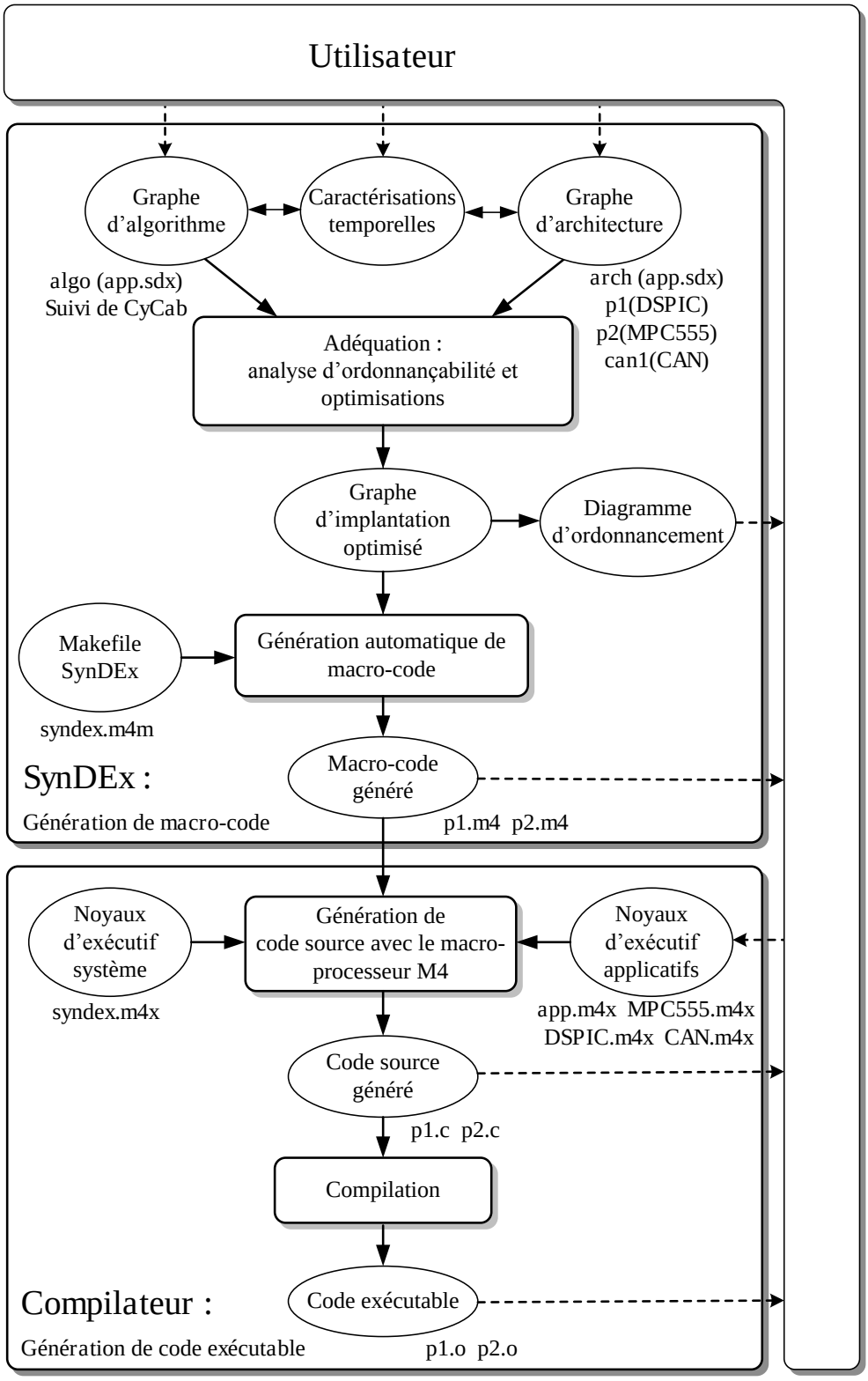


FIGURE 7.2 – Principes de SynDEx

La méthodologie AAA a été implantée au début des années 90 dans un logiciel de CAO au niveau système appelé SynDEx [GMP+90]. Le logiciel actuellement à sa version 7 est disponible gratuitement sur le site web http ://www.syndex.org Les différentes fonctionnalités offertes par SynDEx sont décrites dans la figure 7.2. SynDEx permet de spécifier l'algorithme de l'application temps réel et l'architecture sur laquelle l'application va être exécutée, ainsi que des caractéristiques temporelles. Ensuite il réalise une adéquation qui est une implantation optimisée de l'algorithme sur cette architecture, issue d'une analyse d'ordonnançabilité. Enfin il génère automatiquement un exécutif temps réel dédié à chaque processeur [Gra00, GLS99, GS03].

Dans ce manuscrit nous avons utilisé, jusque là, les termes "tâches" pour décrire un programme formé d'un ensemble d'instructions caractérisées temporellement et "processeur" pour décrire une machine d'exécution. Dans le logiciel SynDEx, le terme "opération" désigne une tâche et le terme "opérateur" désigne un processeur. C'est cette terminologie que nous utilisons à partir de ce chapitre.

7.2 Graphe d'algorithme et d'architecture

7.2.1 Algorithme

Dans la méthodologie AAA le modèle d'algorithme est un graphe de dépendance de données hiérarchique conditionné et factorisé [LS97]. Il s'agit d'un graphe orienté acyclique (DAG) [BR00], dont les sommets sont des opérations partiellement ordonnées [Pra86] par les dépendances de données inter-opérations (diffusion de données à travers des hyperarcs orientés pouvant avoir plusieurs extrémités par une seule origine). Ce graphe est décrit dans la section "#Algorithms" d'un fichier "app.sdx" (figure 7.2) "app" étant le nom de l'application temps réel considérée.

Ce graphe de dépendances est modélisé par un sous-graphe motif infiniment répété (cf. section 1.3.1). Il peut être hiérarchique c'est à dire que chaque opération du graphe peut contenir un sous-graphe permettant une spécification hiérarchique de l'algorithme jusqu'aux opérations atomiques qui sont des opérations élémentaires ne contenant que des ports d'entrée, de sortie ou d'entrée-sortie. Les opérations atomiques dans SynDEx sont :

- capteur appelé "Sensor" : il correspond aux entrées du graphe flot de données. Cette opération peut uniquement produire des données et ne contient que des ports de sorties ;
- actionneur appelé "Actuator" : il correspond aux sorties du graphe flot de données. Cette opération peut uniquement consommer des données et ne contient que des ports d'entrées ;

– constante appelée "Constant" : elle produit des données identiques lors de chaque exécution de l'algorithme,
– retard appelé "Delay" : il permet de spécifier les dépendances inter-itérations du graphe d'algorithme en les mémorisant d'une exécution à une autre.
– fonction appelée "Function" : elle peut être atomique lorsqu'elle ne contient que des ports d'entrée et de sorties, ou hiérarchique si elle contient des opérations de calcul, de conditionnement et de répétition. Une opération conditionnée permet d'exécuter de manière exclusive un de ses sous-graphes en fonction de la valeur de son entrée de conditionnement. Une opération répétée permet de répéter un sous-graphe un nombre de fois correspondant à son facteur de répétition.

La figure 7.3 montre une copie d'écran d'un graphe d'algorithme dans le logiciel SynDEx. Il contient une opération capteur $input$ qui deux fois plus rapide que les autres opérations et qui a un port de sortie o, une opération actionneur $output$ ayant deux ports d'entrée i et j, et deux fonctions $Op1$ et $Op2$ ayant chacune un port d'entrée i et un port de sortie o.

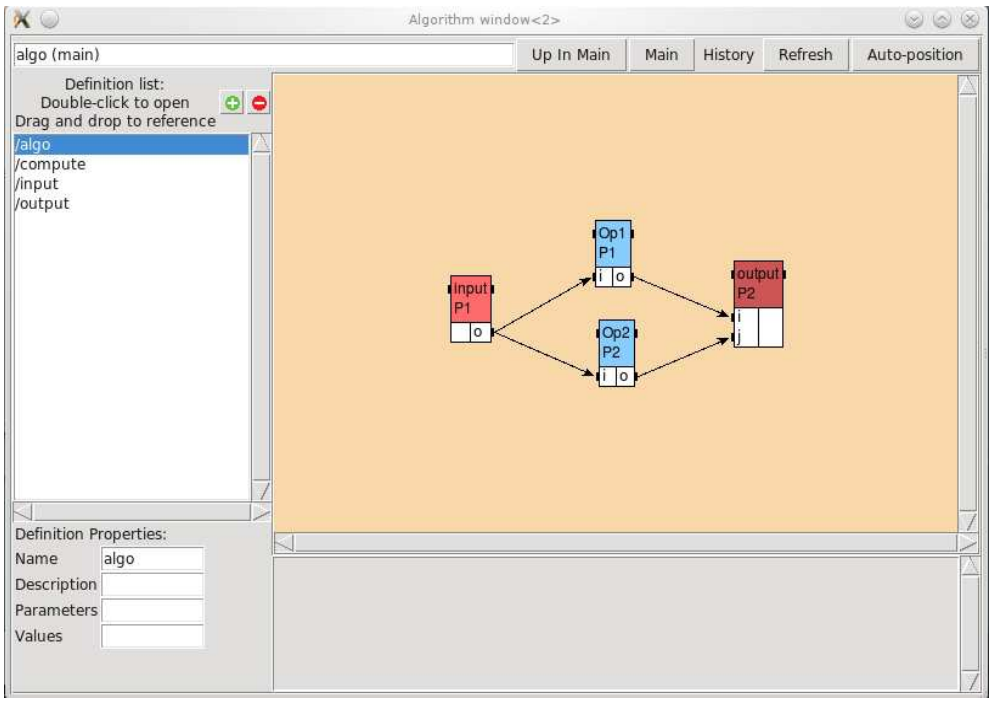


FIGURE 7.3 – Graphe d'algorithme dans SynDEx

7.2.2 Architecture

Dans SynDEx on considère des architectures hétérogènes multi-composants [Sor96, GS03]. Nous nous restreindrons dans cette thèse à des composants programmables (processeurs, microcontrôleurs, etc.). C'est à dire que l'architecture ne contient aucun composant non programmable tels que des ASIC , des FPGA, etc. Le graphe d'architecture est constitué de trois types de sommets : opérateur, mémoire partagée et multiplexeur-démultiplexeur, et d'arcs qui relient ces trois types de sommets. Un processeur est formé d'une machine séquentielle exécutant des instructions et d'autant de machines séquentielles de communication que de connexions possibles avec d'autres processeurs. Un médium de communication de type passage de messages est un sous-graphe du graphe d'architecture formé soit de deux machines séquentielles de communication (point-à-point) soit de plus de deux machines séquentielles de communication (bus) avec leurs mémoires distribuées séquentielles SAM (Sequential Access Memory) associées, connectés par des arcs. Un médium de communication est un sous-graphe du graphe d'architecture de type mémoire partagée est formé d'au moins deux machines séquentielles de communication et d'une mémoire partagée RAM (Random Access Memory), connectés par des arcs. Ce graphe est décrit dans la section "#Architectures" d'un fichier "app.sdx" (figure 7.2).

La figure 7.4 montre une copie d'écran d'un graphe d'architecture dans SynDEx composé de deux opérateurs $P1$ (resp. $P2$) de type $RTAI$ (resp. $dsPIC$) comprenant deux points d'accroches $can1$ (resp. $can2$) de type CAN représentant chacun une machine séquentielle de communication et sa mémoire distribuée associée SAM. Ces points d'accroche sont appelés dans SynDEx des "gates". Ce graphe comprend deux média de communication de type passage de message : $can1(P1)$, $can1(CAN)$, $can1(P2)$ et $can2(P1)$, $can2(CAN)$, $can2(P2)$.

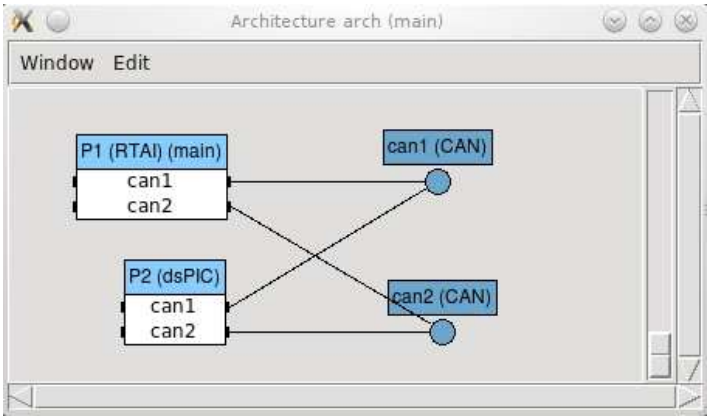


FIGURE 7.4 – Graphe d'architecture dans SynDEx

7.2.3 Caractérisations temporelles

Les sommets du graphe d'algorithme sont étiquetés par une période et une échéance égale à la période (échéance sur requête) indépendantes de l'architecture et par une durée d'exécution pire cas (WCET) dépendante de l'architecture. Toutes les périodes sont harmoniques. De la même manière les dépendances de données sont étiquetées par des caractéristiques temporelles dépendantes de l'architecture WCCT. Les périodes des communications sont déduites de celles des opérations dépendantes. Ce caractérisations temporelles sont décrites dans la section "#Extra durations" d'un fichier "app.sdx" (figure 7.2).

7.3 Mise à plat

Avant d'effectuer l'adéquation, SynDEx commence par mettre à plat le graphe d'algorithme spécifié par l'utilisateur. La mise à plat à pour objectif de supprimer la hiérarchie fonctionnelle (spécifications top-down) et la hiérarchie due aux répétitions et au conditionnement d'algorithmes en vue d'obtenir un graphe ne contenant que des opérations atomiques. De plus la mise à plat vérifie que tous les cycles contiennent un retard. Le graphe obtenu après la mise à plat contenant uniquement des opérations atomiques.

7.4 Adéquation dans SynDEx V7

L'adéquation consiste à trouver une distribution et un ordonnancement optimisé d'un graphe de dépendance factorisé conditionné donné, sur une architecture donnée. Elle est formée de trois algorithmes qui s'exécutent dans l'ordre suivant (cf. section 5.1) : (i) algorithme d'analyse d'ordonnançabilité, (ii) algorithme de déroulement, (iii) algorithme d'ordonnancement. Ces algorithmes ont été donnés par O. Kermia dans [Ker09, KS07].

7.4.1 Analyse d'ordonnançabilité

L'algorithme d'analyse d'ordonnançabilité implanté dans SynDEx est basé sur la condition d'ordonnançabilité restrictive (3.12).

7.4.2 Déroulement

L'algorithme de déroulement implanté dans SynDEx est le même que l'algorithme 6 décrit dans la section 5.3. En plus de répéter les opérations suivant le rapport entre leurs périodes et l'hyper-période. Dans le cas de deux opérations dépen-

dantes où l'opération productrice de données a une période plus petite que l'opération réceptrice, cet algorithme crée des opérations "implode" qui permettent de regrouper les $n = \frac{T_{rceptrice}}{T_{productrice}}$ données envoyées par plusieurs instances d'opérations dépendantes à leur successeur. Cette opération regroupe les données et les envoie sous forme d'un tableau à son successeur. Dans le cas de deux opérations dépendantes où l'opération productrice de données a une période plus grande que l'opération réceptrice, l'opération productrice produit $n = \frac{T_{productrice}}{T_{rceptrice}}$ la même donnée.

7.4.3 Ordonnancement

L'algorithme d'ordonnancement est initialisé par le graphe d'algorithme produit par l'algorithme de déroulement contenant les opérations "implode" rajoutées. À chaque itération de l'algorithme d'ordonnancement, la tâche candidate peut être soit une première instance o_i^1 soit une instance o_i^k avec $k \neq 1$. Si la tâche candidate est une première instance o_i^1 alors elle est ordonnancée, sans temps creux après réception des données envoyées par ses prédécesseurs, sur le processeur sur lequel elle minimise la fonction de coût pression d'ordonnancement. L'algorithme vérifie ensuite que cette tâche et toutes les tâches déjà ordonnancées satisfont la condition nécessaire et suffisante (3.6). Si cette condition n'est pas satisfaite alors cette tâche n'est pas ordonnancée et elle est donc remise dans l'ensemble des tâches candidates pour être ordonnancée ultérieurement. Si la tâche candidate est une instance quelconque o_i^k avec $k \neq 1$, alors elle est ordonnancée sur le processeur sur lequel on a ordonnancé la tâche o_i^1, avec sa date de début d'exécution $S_i^k = S_i^0 + kT_i$.

Le résultat de l'adéquation (figure 7.2) est un diagramme d'ordonnancement qui donne pour chaque processeur (resp. médium de communication) et pour chaque opération (resp. communication) son nom, sa date de début d'exécution S_i^k et sa durée d'exécution (resp. communication). La date de fin d'une opération est la somme de sa date de début d'exécution et de sa durée. Les temps creux entre la fin d'exécution d'une opération et le début d'exécution de la suivante sont nommés "wait".

La figure 7.5 montre une copie d'écran du résultat de l'adéquation du graphe d'algorithme montré dans la figure 7.4 sur le graphe d'architecture montré dans la figure 7.3. Comme l'opération "input" est deux fois plus rapide que les autres opérations, le diagramme montre deux instances "input#1" et "input#2" séparées par un "Wait_period_input", une opération "Implode_Op1" (resp. "Implode_Op2") qui regroupe les données et les envoie par "input#1" et "input#2" et les envoie à l'opération "Op1" (resp. "Op2"). L'opération "output" commence son exécution après la réception des données des opérations "Op1" et "Op2".

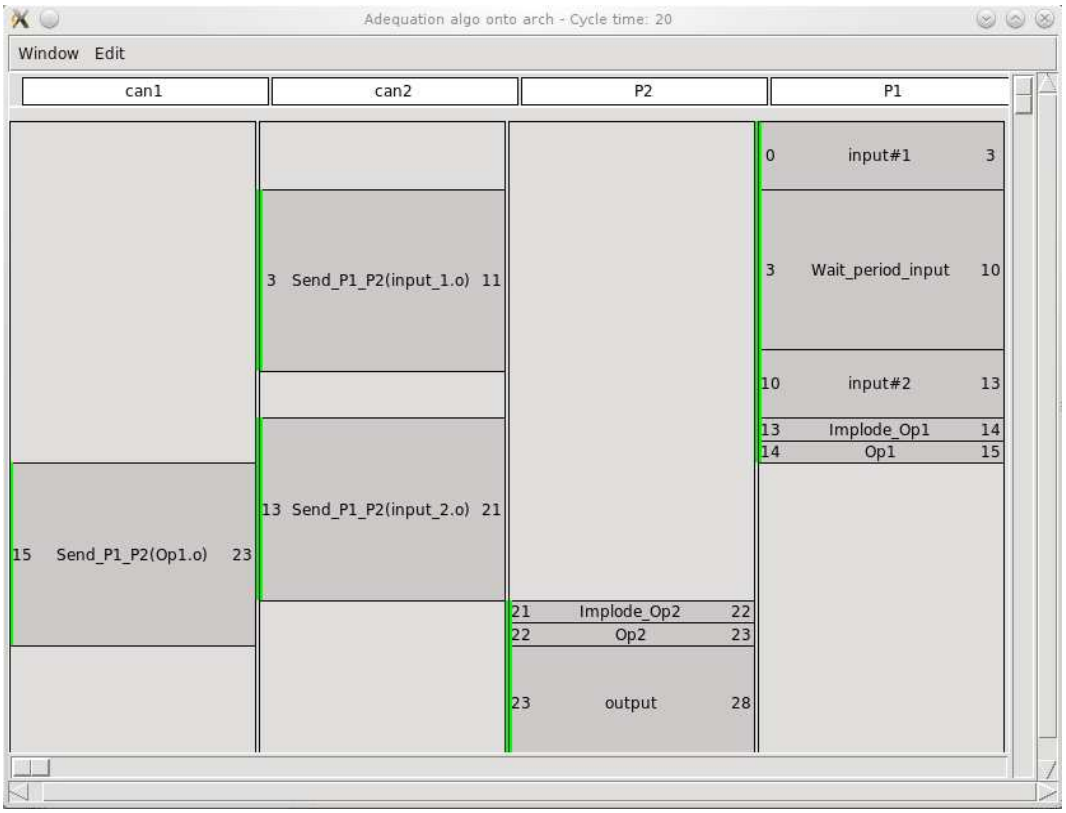


FIGURE 7.5 – Graphe temporel d'exécution de l'algorithme

7.5 Génération automatique de code

Le diagramme d'ordonnancement est utilisé pour construire une table d'ordonnancement (cf. section 3.1) contenant les dates de début d'exécutions de toutes les tâches pour chaque processeur et les dates de début de toutes les communications inter-processeurs. La génération automatique de code produit, pour chaque processeur de l'architecture, une séquence de macro-instructions dont l'ordre est conforme à celui de la table d'ordonnancement. L'aspect périodique de l'ordonnancement est imposé par une macro-instruction particulière qui effectue une boucle infinie (hyper-période) de cette séquence de macro-instructions.

Pour chaque processeur un code source est généré à partir du macro-code correspondant et des noyaux d'exécutif système contenant les macro-définitions des fonctions système et des noyaux d'exécutif applicatifs contenant les macro-définitions des opérations de l'algorithme. Enfin chaque code source est compilé pour obtenir le code temps réel exécutable (figure 7.2).

7.5.1 Génération de macro-code

SynDEx génère un "macro-code" par processeur comprenant une boucle infinie de macro-instructions exécutant les :

- opérations distribuées et ordonnancées sur ce processeur,
- communications inter-processeur : couples de (SEND, RECEIVE) envois et réceptions de messages de données pour les SAM et de (WRITE, READ) écritures et lectures de données pour les RAM,
- synchronisations intra-processeur,
- synchronisations inter-processeur.

Les synchronisations assurent que même s'il y a des variations sur les durées des opérations l'ordre dans lequel ces dernières s'exécuteront sera compatible avec l'ordre partiel du graphe d'algorithme initial.

Les synchronisations intra-processeur sont de deux types :

- intra-répétition : pour assurer l'exécution en parallèle, correcte au sens de l'ordre partiel initial, de la séquence unique de calculs et des séquences de communications à l'intérieur d'une répétition infinie,
- inter-répétition : pour assurer que les répétitions infinies se succèdent correctement. En effet il faut assurer qu'une itération infinie soit terminée avant de passer à la suivante, donc que tous les messages de données ont bien été transmis et reçus sur toutes les SAM ou que toutes les données ont bien été écrites et lues dans toutes les mémoires partagées RAM.

Les macro-instructions de synchronisation effectuent de manière atomique un "lire-modifier-écrire" d'un sémaphore et se déclinent comme suit :

- intra-répétition : `pre_full, succ_full` : signale tampon plein, attend tempon plein (figure 7.6 et 7.7),
- inter-répétition : `pre_empty, succ_empty` : signale tampon vide, attend tampon vide (figure 7.6 et 7.7).

Les synchronisations inter-processeur réalisent les synchronisations entre les boucles infinies de plusieurs processeurs. Elles sont de deux types :

- passage de messages :
  - d'aucun message pour un médium point-à-point, la FIFO est autosynchronisante,
  - d'un message SYNC envoyé à tous les processeurs pour un médium multi-point diffusant,
  - de messages SYNC envoyés uniquement aux processeurs concernés pour un médium multi-point non diffusant,
- mémoire partagée :
  - `preR_full, succR_full` : signale mémoire pleine, attend mémoire

pleine ,
– `preR_empty, succR_empty` : signale mémoire vide, attend memoire vide.

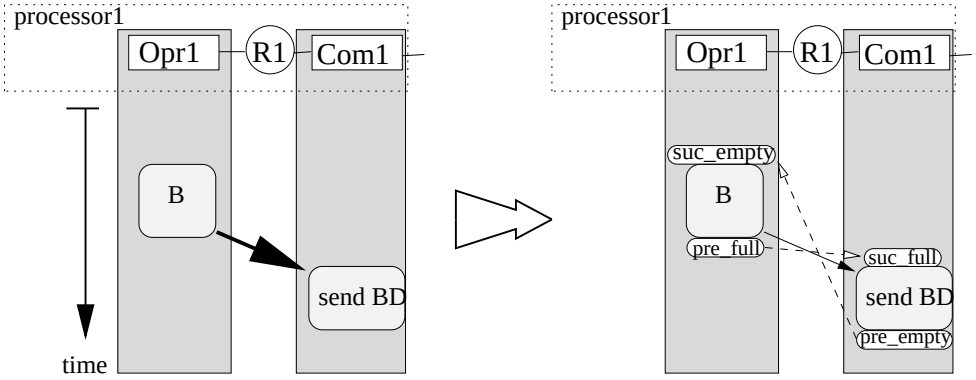


FIGURE 7.6 – Synchronisations intra-répétition et inter-répétition pour un envoi de donnée

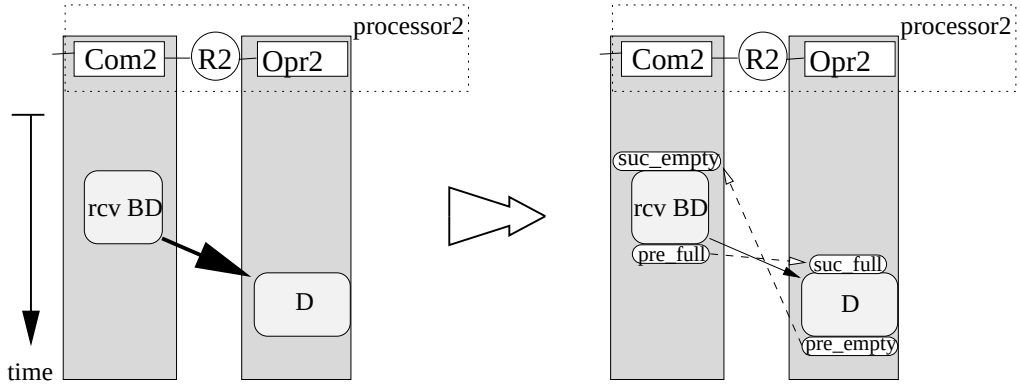


FIGURE 7.7 – Synchronisations intra-répétition et inter-répétition pour une réception de donnée

Pour chaque processeur "p" le macro-code généré est contenu dans un fichier "p.m4" (figure 7.2).

La figure 7.8 montre une copie d'écran du macro-code de l'application montrée par la copie d'écran du graphe d'algorithme (figure 7.3) et du graphe d'architecture (figure 7.4).

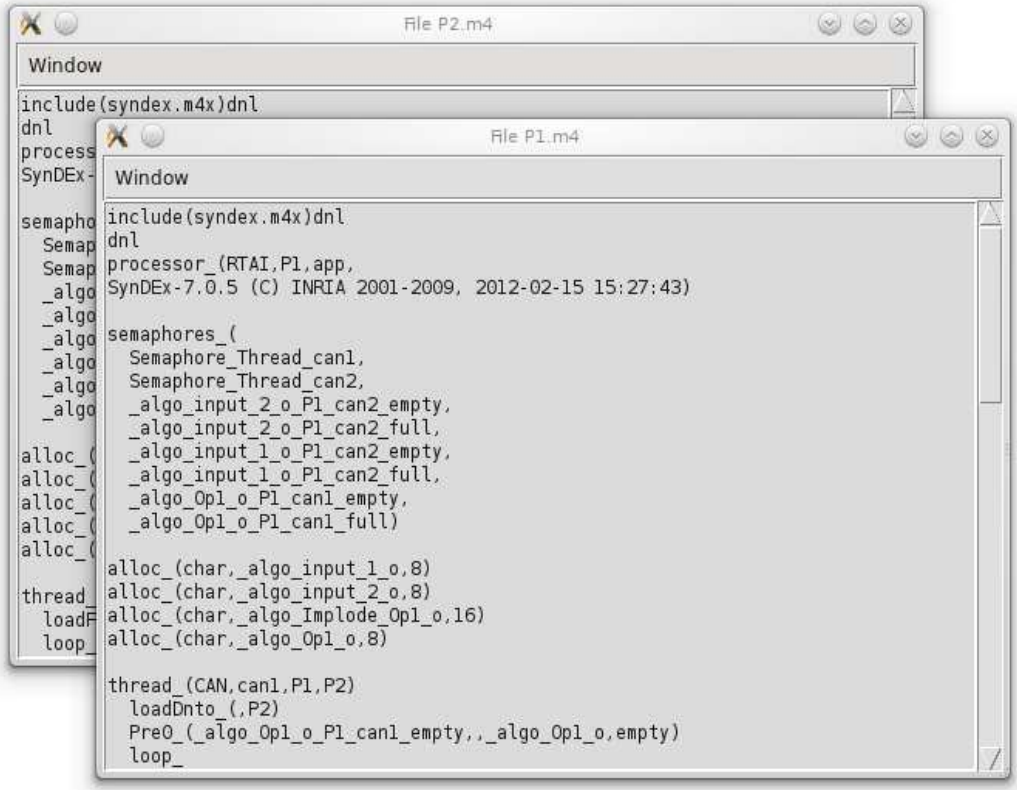


FIGURE 7.8 – Macro-code généré pour un processeur

7.5.2 Génération de code exécutable

Le macro-code de chaque processeur est macro-processé avec :

- un *noyau d'exécutif* dépendant de l'architecture et de l'éventuel exécutif résident, par exemple VxWorks, Osek, Linux/RTAI, Windows/RTX, etc. Chaque noyau d'exécutif contient les macro-définitions décrivant comment chaque macro-instruction sera traduite en du code source compilable ;
- un *noyau applicatif* dépendant de l'architecture contenant les macro-opérations correspondant aux opérations du graphe d'algorithme. Chaque noyau applicatif contient les macro-définitions décrivant comment chaque macro-opération sera traduite en du code source compilable.

Ces noyaux sont contenus dans des fichier "m4x". Les codes sources obtenus (fichiers "c", "asm", etc.) sont ensuite compilés pour produire les exécutables (fichiers "o") chargés et exécutés sur les processeurs de l'architecture (figure 7.2).

Pour tous les exécutables obtenus les synchronisations assurent que l'ordre partiel de l'algorithme est conservé par la génération automatique de code garantissant un fonctionnement en temps réel sans inter-blocage sur l'architecture distribuée.

7.6 Améliorations apportées à SynDEx

Les améliorations que nous avons apportées à SynDEx concernent d'une part les algorithmes d'analyse d'ordonnançabilité et d'ordonnancement présentés dans les chapitres 3 et 5 et d'autre part la tolérance aux fautes de processeurs et de bus de communication présentée dans le chapitre 6.

7.6.1 Adéquation

L'algorithme 5 d'analyse d'ordonnançabilité pour lequel nous avons fait une spécification logicielle, est celui proposé dans la section 5.2. L'algorithme 5 a été programmé en OCaml et testé.

Nous avons aussi fait une spécification logicielle de l'algorithme 7 d'ordonnancement qui calcule les dates de début d'exécution de chaque tâche selon l'une des conditions d'ordonnançabilité (3.12), (3.14) ou (3.16) en prenant en compte les durées de communication ce qui peut amener à avoir des temps creux entre la date de début d'exécution d'une tâche et la date de réception des données de ses prédécesseurs. Contrairement à ce que nous avons proposé, l'algorithme d'ordonnancement implanté actuellement dans SynDEx ordonnance chaque tâche, sans temps creux entre sa date de début d'exécution et la date de réception des données de ses prédécesseurs, ensuite il vérifie si cette tâche et toutes les tâches déjà ordonnancées satisfont la condition nécessaire et suffisante (3.6). Cet ordonnancement sans temps creux peut réduire l'ordonnançabilité des tâches comme nous l'avons montré à la section 5.4.1.

Nous avons aussi programmé en Matlab [mat] et testé (i) un générateur de tâches harmoniques et non harmoniques, (ii) l'algorithme 1 d'ordonnancement des tâches harmoniques, (iii) l'algorithme 2 d'ordonnancements des tâches non-harmoniques et (iiii) l'algorithme 5 d'analyse d'ordonnançabilité des tâches non-harmoniques. Nous avons utilisé ces programmes Matlab pour réaliser un simulateur permettant d'effectuer l'analyse de performances présentée dans la section 3.6.

L'algorithme 6 de déroulement n'a pas été modifié.

7.6.2 Extension de SynDEx à la tolérance aux fautes

Nous avons proposé une extension de SynDEx pour la tolérance aux fautes suivant les principes décrits dans le chapitre 6. Afin de faire une étude d'ordonnançabilité tolérante aux fautes, nous transformons le graphe d'algorithme pour que chaque opération (resp. dépendance de données) du graphe d'algorithme initial soit remplacée par des opérations (resp. dépendances de données) répliques exclusives. Pour cela nous avons fait une spécification logicielle de l'algorithme 8 de transformation de graphe qui prend en entrée un graphe d'algorithme Alg et les hypothèses de tolérance aux fautes N_{pf} et N_{pf} pour produire un graphe transformé Alg^* et des relations d'exclusions d'opérations et de dépendances de données $Excl$. Nous avons aussi fait une spécification logicielle de l'algorithme 9 d'analyse d'ordonnançabilité qui prend en compte les relations d'exclusion des opérations seulement et qui assigne chaque opération à un seul opérateur. On rappelle que l'on a fait ce choix pour améliorer l'ordonnançabilité (cf. section 6.4.1). Nous avons enfin fait une spécification logicielle de l'algorithme 10 d'ordonnancement qui prend en compte les relations d'exclusion des dépendances de données seulement.

La figure 7.9 montre l'extension de SynDEx à la tolérance aux fautes qui contient la transformation de graphe et l'adéquation. Les algorithmes colorés sont ceux que nous avons proposés ou modifiés et pour lesquels nous avons fait des spécifications logicielles ou nous les avons programmés en OCaml.

7.7 Conclusion

Nous avons présenté dans ce chapitre les améliorations apportées au logiciel SynDEx. Nous avons commencé par un rappel concernant le logiciel SynDEx puis nous avons présenté le graphe d'algorithme, le graphe d'architecture et les spécifications temporelles. Ensuite nous avons expliqué la mise à plat, l'adéquation et la génération de code. Enfin nous avons présenté les améliorations apportées à SynDEx tant sur le plan d'analyse d'ordonnançabilité et d'ordonnancement que sur le plan de la tolérance aux fautes.

Nous présentons dans le chapitre suivant une application de tout ce qui précède au suivi automatique de véhicules électriques CyCabs tolérante aux fautes.

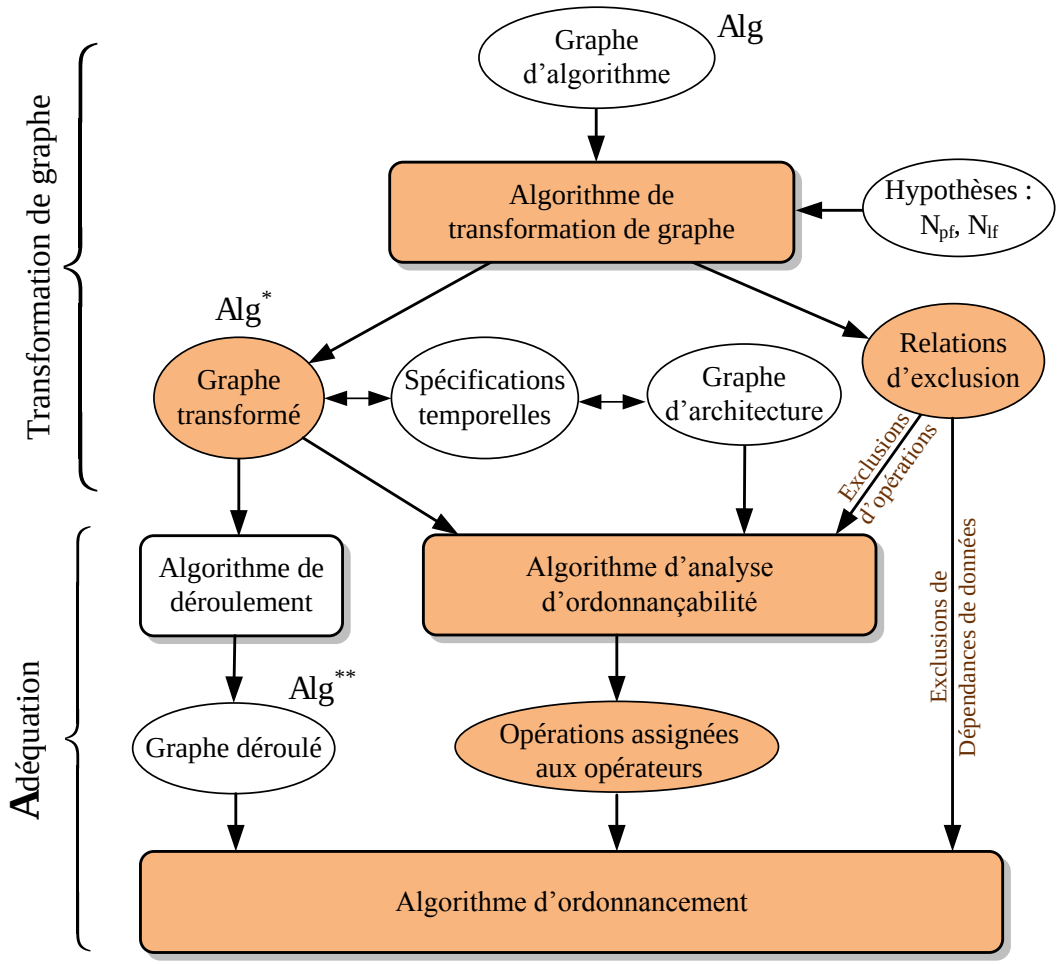


FIGURE 7.9 – SynDEx multipériode tolérant aux fautes

Chapitre 8

Application au suivi automatique de CyCabs tolérant aux fautes

8.1 Histoire de la conduite automatique à l'INRIA

Les chercheurs de l'INRIA et de l'INRETS (Institut National de Recherche sur les Transports et leur Sécurité) travaillent depuis 1991 sur de nouveaux moyens de transport intelligents pour la ville. Ils étudient en particulier le concept de la voiture en libre-service et celui de la voiture automatique. Les premiers résultats de recherche ont débouché sur le projet Praxitèle (1993-1999), qui a été mis en exploitation à Saint-Quentin-en-Yvelines. Les partenaires industriels du projet étaient CGFTE (la filiale transports publics de Vivendi), Dassault Électronique, EDF et Renault. Dans le cadre du projet Praxitèle l'INRIA a démontré la faisabilité de la conduite automatique dans certaines situations ; créneau et train de véhicule expérimenté sur un véhicule électrique Ligier instrumenté à cet effet. Pour des raisons de législation et de responsabilité ces systèmes de conduite n'ont pas pu être implémentés sur les Clio électriques de Saint-Quentin-en-Yvelines.

Dans ce contexte de la route automatisée, l'INRIA a proposé un tel système basé sur une flotte de petits véhicules électriques, appelés CyCab, spécifiquement conçus pour les zones où la circulation automobile doit être fortement restreinte. Le CyCab (contraction pour Cyber Cab) a été développé avec l'aide de l'INRETS, de EDF, de la RATP et de la société Andruet S.A. Il a été réalisé par la société Robosoft [rob] (Figure 8.1).

Le CyCab est un véhicule électrique à quatre roues motrices et directrices avec une motorisation indépendante pour chacune des roues et pour les directions. Pour contrôler et commander les 10 moteurs du CyCab (4 de traction, 2 de direction et 4 de frein), une architecture matérielle de contrôle/commande a été choisie. Elle est constituée de deux microcontrôleurs MPC555 et d'un PC embarqué Linux/RTAI

FIGURE 8.1 – CyCab

commandant les différents moteurs du CyCab et communicant par un bus de terrain CAN (Controller Area Network), très répandu dans le monde de l'automobile.

Le rôle des microcontrôleurs est d'asservir les moteurs en fonction des consignes de vitesse et de braquage, qui transitent sur le bus CAN, soit en provenance de la position du joystick, soit par un programme de planification de trajectoires. Un microcontrôleur et son électronique de puissance doit donc non seulement être capable de fournir la puissance nécessaire aux moteurs, mais aussi exécuter les boucles d'asservissement de vitesse ou/ou de position. Pour ce faire il doit prendre en compte un certain nombre d'informations en provenance des capteurs proprioceptifs : relais électriques (arrêt d'urgence, etc.), codeur incrémental (odométrie des roues, calcul de vitesse, d'accélération, etc.), codeur absolu (angle de braquage de roues), etc.

Dans le contexte de la voiture en libre-service, des trains virtuels de véhicules permettent de déplacer les véhicules d'un parking à un autre afin d'équilibrer leurs disponibilités. Un train virtuel de véhicules est constitué d'un véhicule de tête conduit par un chauffeur et d'autres véhicules automatisés, chacun suivant celui qui le précède. Ainsi le premier véhicule est suivi par le deuxième qui à son tour est suivi par le troisième, etc. Ce type d'automatisation a été pensé non seulement pour l'utilisation des voitures en libre-service, mais aussi plus tard la conduite régulée sur autoroute et sur voies périphériques. Ce procédé a l'avantage de maximiser la vitesse des véhicules ainsi que leur nombre tout en minimisant les accidents. L'application de suivi de CyCab que nous décrivons dans la suite est au cœur de ce système.

8.2 Caractéristiques générales du CyCab

La figure (8.2) montre une vue en coupe de l'architecture du CyCab qui est constituée de :

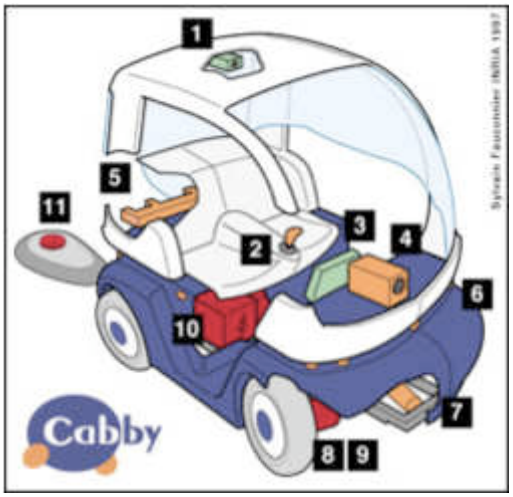


FIGURE 8.2 – Architecture matérielle d'un CyCab

– 1 ensemble de batteries avec un gestionnaire automatique de charge (10) et un bouton arrêt d'urgence qui est soit de type poussoir (2) soit de type radio-commandé (1) ;
– 2 cartes électroniques (5) et (6) d'acquisition de données comprenant chacune un microprocesseur 32 bit PowerPC (appelés MPC555). Chaque carte permet de contrôler 2 roues du CyCab.
– 1 PC embarqué au format rack (taille 2U), placé sous le siège (2), possédant un processeur Intel cadencé à 3 GHz, avec un Linux temps réel, RTAI. L'ensemble est alimenté par une tension d'entrée de -48V (350W) et non de 220V. L'écran est situé en (3) ;
– 2 bus CAN indépendants : le bus CAN 1 permet la communication entre les 2 MPC555 et le PC embarqué, alors que le bus CAN 2 permet d'ajouter d'éventuels futurs composants électronique ;
– 4 moteurs et leurs freins électriques (8) et (9) contrôlés par 4 contrôleurs de moteur appelés Curtis PMC 1227 (9) servant d'amplificateurs de puissance pour contrôler la vitesse des roues. La consigne de vitesse est donnée par une tension de 0 à 5V aux Curtis qui fourniront des signaux PWM adéquats aux moteurs. Les Curtis protègent les MPC555 des contre-courants des moteurs, quand par exemple, on les arrête brusquement ;
– 4 décodeurs incrémentaux donnant la vitesse des roues (8).
– 1 vérin de direction électrique alimenté par signal PWM (7) faisant pivoter les 4 roues ;
– 1 encodeur absolu avec sortie SPI et donnant l'angle des roues ;
– 1 joystick (2) fournissant deux courants indiquant : – la consigne de vitesse des roues, – la consigne de direction des 4 roues ;

– le CyCab possède une caméra type webcam (4) se branchant sur un port FireWire du PC embarqué.

Plus de détails concernant le PC embarqué et les MPC555 se trouvent dans l'annexe A.

8.3 Applications de commande de CyCabs

A l'heure actuelle, il existe deux branches principales d'applications CyCab : l'application manuelle et l'application automatique.

8.3.1 Application manuelle

L'application manuelle du CyCab est une application de contrôle/commande en vitesse à l'aide d'un joystick. La figure 8.3 montre le graphe d'algorithme dans SynDEx de cette application.

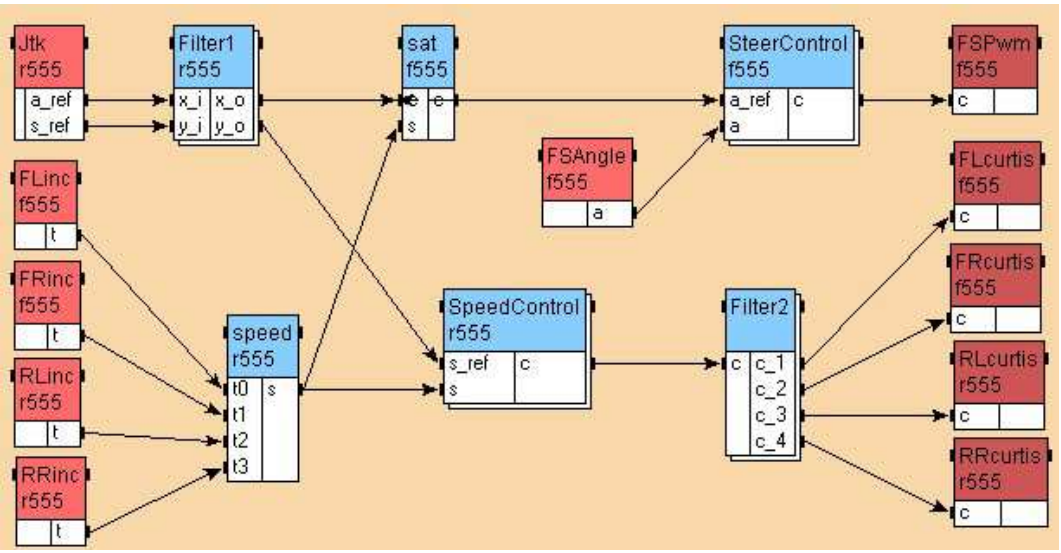


FIGURE 8.3 – Graphe d'algorithme de l'application manuelle du CyCab

Ce graphe d'algorithme est composé des opérations suivantes :

Opérations d'entrée (capteurs) :

– un joystick "Jtk" qui donne les positions relatives du joystick "x" et "y" servant relativement de référence pour d'angle de braquage des roues la vitesse longitudinale du CyCab,

– quatre encodeurs incrémentaux des roues avant droite "FRenc" (front right encoder), avant gauche "FLenc" (front left encoder), arrière droite "RRenc"

(rear right encoder), arrière gauche “RLenc” (rear left encoder), qui donnent la rotation élémentaire de chaque roues,
– un encodeur absolu du moteur de braquage avant “FAenc” (front absolute encoder) qui donne l’angle de braquage des roues avants,

Opérations de calcul :
– une opération “speed” qui calcule la vitesse longitudinale du CyCab à partir des données issues des encodeurs incrémentaux,
– un contrôleur de vitesse longitudinale “SpeedCtrl” de type proportionnel intégral, qui prend la consigne de vitesse “s_ref” issue du joystik et la vitesse du CyCab “s” issue de l’opération “speed”, et produit une commande “c” avec :

$$c = k_p(s\_ref - s) + k_i \int_0^t (s\_ref - s)dt \ ,$$

– un contrôleur d’angle de braquage “SteerCtrl” de type proportionnel intégral, qui prend la consigne d’angle de braquage “a_ref” issu du joystik et l’angle de braquage du CyCab “a” issu de l’encodeur absolu “FAenc” et produit une commande “c” avec :

$$c = k_p(a\_ref - a) + k_i \int_0^t (a\_ref - a)dt \ ,$$

– deux opérations “Filter” permettant de filtrer les sorties du joystick et celle du contrôleur de vitesse,
– une opération de saturation “sat” permettant de limiter l’angle de braquage des roues.

Opérations de sortie (actionneurs)
– quatre opérations “FLcurtis”, “FRcurtis”, “RLcurtis” et “RRcurtis” pour les quatre amplificateurs des moteurs de traction,
– un opération “FSPwm” pour l’emplificateur du moteur de braquage des roues avants.

La figure 8.4 montre le graphe d’architecture de cette application dans SynDEx. Il est composé d’un PC embarqué Linux/RTAI “root”, deux microcontrôleurs MPC555 “f555” et “r555”, et de bus CAN “CAN1”.

8.3.2 Application automatique

L’application automatique est une application de suivi de trajectoire par accrochage immatériel de type remorque (figure 8.5). Le CyCab suiveur doit suivre la trajectoire d’un CyCab suivi en maintenant une inter-distance constante $\bar{d}$ entre les deux CyCabs et en dirigeant ses roues vers le CyCab suiveur.

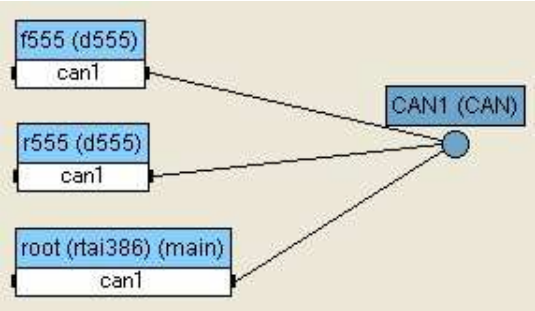


FIGURE 8.4 – Graphe d'architecture de l'application manuelle du CyCab

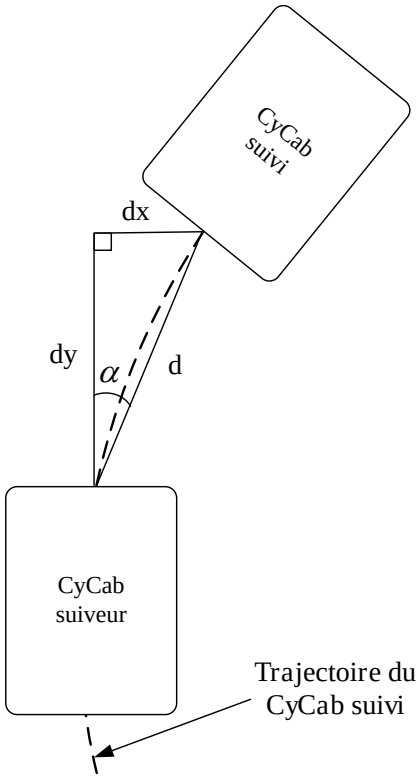


FIGURE 8.5 – Distance longitudinale dy, distance latérale dx et angle de déportation α

Un programme de traitement d'images permet d'extraire les informations utiles du CyCab suivi à partir des données issues d'une caméra à bas coût. Il procède d'abord à la détection de contours du CyCab suivi puis il calcule l'inter-distance $d = \sqrt{dx^2 + dy^2}$ et l'angle de déportation $\alpha = Arctan\left(\frac{dx}{dy}\right)$ (figure 8.5). L'inter-distance d servira pour l'asservissement longitudinal basé sur un régulateur de

distance et l'angle de déportation α pour l'asservissement latéral basé sur un régulateur d'angle de braquage des roues du CyCab suivi.

Comme le traitement d'image doit prendre le rôle du joystick alors il doit produire une vitesse et un angle de braquage de consigne. Comme le suivi est de type remorque alors l'angle de braquage des roues est égal à l'angle de déportation :

$$a\_ref = \alpha.$$

La consigne de vitesse est calculée par un contrôleur proportionnel intégral comme suit :

$$s\_ref = k_p(\bar{d} - d) + k_i \int_0^t (\bar{d} - d)dt \ .$$

La figure 8.6 montre le graphe d'algorithme dans SynDEx de cette application. Il est basé sur le graphe d'algorithme de l'application manuelle en ajoutant une opération de traitement d'images "TI" qui produit une consigne d'angle de braquage des roues avant "a_ref" et une consigne de vitesse "s_ref".

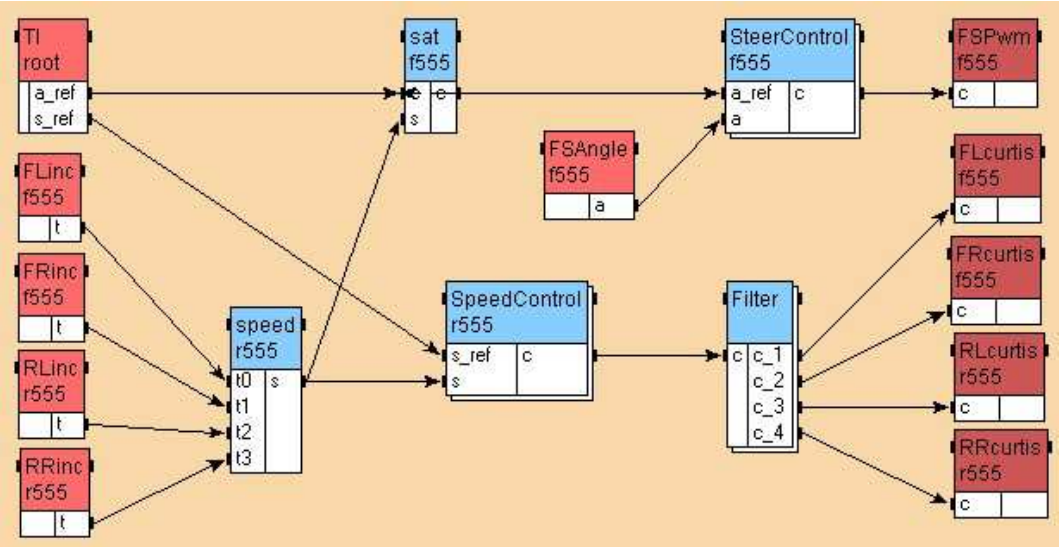


FIGURE 8.6 – Graphe d'algorithme de l'application automatique du CyCab

Le graphe d'architecture est identique à celui de l'application manuelle.

8.4 Architecture à base de dsPIC et de bus CAN

Comme nous avions choisi de ne pas traiter les fautes de capteurs et des actionneurs (hypothèses capteurs et actionneurs fiables) gérés par les microntôleurs

MPC555 dans l'architecture du CyCab, nous avons proposé de modifier cette dernière en y ajoutant des dsPICs que l'on peut, eux, faire tomber en panne. Afin de ne pas endommager les CyCabs lors des expérimentations sur la tolérance aux fautes, nous avons construit un banc de test composé de dsPICs sur lequel nous avons expérimenté nos premiers tests de détection d'erreurs et de tolérance aux fautes.

8.4.1 Banc de test

Nous avons réalisé un banc de test composé de trois microcontrôleurs dsPIC "pic33FJ128MC708" et deux bus CAN. Le bus CAN est un bus série utilisant le multiplexage pour permettre la connexion de nombreux capteurs/actionneurs et calculateurs en minimisant le câblage. C'est un bus diffusant (broadcast) qui, lorsqu'il envoie un message, celui-ci est reçu par l'ensemble des utilisateurs du bus. Ce banc de test nous a permis d'effectuer les tests de détection d'erreurs et de tolérance aux fautes de dsPICs et de bus CAN.

L'architecture du banc de test est composée de :

– trois microcontrôleurs dsPIC "pic33FJ128MC708",
– deux bus CAN.

Le protocole de communication CAN (Control Area Network) est utilisé dans le secteur de l'industrie automobiles. Il permet de connecter plusieurs calculateurs à un seul bus en minimisant le câblage. Les pic33FJ128MC708 sont des dsPICs (digital signal PICs), microcontôleurs de la société Microchip avec un architecture 16 bits. Sur ce banc, chaque dsPIC est relié à deux bus de communication CAN et une alimentation, grâce à neuf interrupteurs (figure 8.7). Ce banc permet la simulation d'erreurs de processeurs grâce à des interrupteurs reliés à l'alimentation des processeurs et des erreurs de bus grâce à des interrupteurs reliant les communicateurs aux bus.

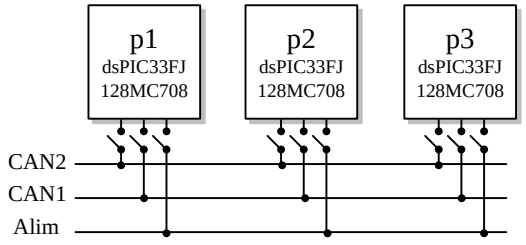


FIGURE 8.7 – Architecture du banc de test

Cette architecture simple est particulièrement utile pour la compréhension du code de communication entre les processeurs et celui de la tolérance aux fautes.

Les processeurs sont au nombre de trois pour permettre la simulation de cas généraux, ce qui est impossible avec uniquement deux dsPICs.

Plus de détails concernant les microcontrôleurs dsPIC "pic33FJ128MC708" sont données dans l'annexe A.

8.4.2 Cycab tolérant aux fautes

L'architecture du CyCab tolérante aux fautes est composées de :
- l'architecture de base du CyCab : MPC555, PC embarqué Linux/RTAI, deux bus CAN,
- le banc de test : trois dsPICs pic33FJ128MC708.

L'intégration des dspic à l'architecture de base permet d'implanter la tolérance aux fautes par redondance logicielle sur les dsPICs puisqu'on ne peut pas le faire sur les microcontrôleurs MPC555. La figure 8.8 montre l'architecture du CyCab tolérant aux fautes.

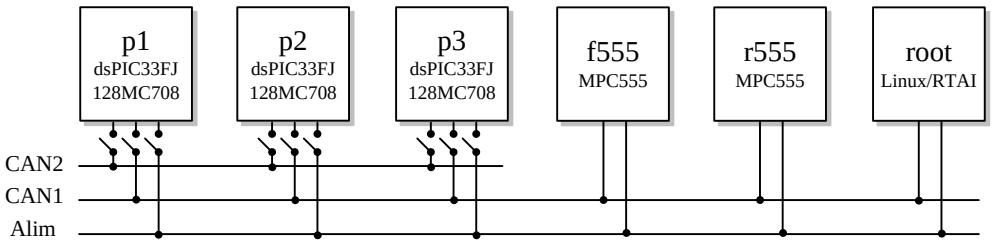


FIGURE 8.8 – Architecture du CyCab tolérant aux fautes

8.5 Code source du dsPIC

Après avoir spécifié dans SynDEx le graphe d'algorithme de l'application de suivi de CyCab et le graphe d'architecture hétérogène contenant les deux MPC555, le PC embarqué Linux/RTAI, les trois dsPICs et le bus CAN, l'adéquation génère un macro-code "pi.m4" pour chaque processeur "pi" de cette architecture. Chaque macro-code est ensuite macro-processé avec les noyaux d'exécutifs SynDEx correspondants pour générer du code source. Dans le cas des dsPICs le code source généré est du "C". Ce dernier est compilé pour obtenir des exécutables comme nous l'avons présenté dans la section 7.5 (cf. figure 7.2).

SynDEx génère autant de threads de communication CAN qu'il y a de gates dans un processeur dsPIC (cf. section 7.2.2). Un thread de communication est déclaré comme suit : `void thread_can1(volatile unsigned state)` Une variable "contexte" (`CAN_ctxt_x`) contenant toutes les informations du bus

canx auquel le gate x est connecté est associée à chaque thread de communication de bus d'indice x (thread_canx) (x=1 pour CAN1, x=2 pour CAN2). Les synchronisations intra-processeurs sont gérées par les sémaphores intra-répétition pre_full, suc_full et les sémaphores inter-répétition pre_empty, suc_empty. L'ensemble de ces sémaphores est regroupé dans un tableau de variables sem.

Si la communication dans un système à deux processeurs communiquant par un bus CAN broadcast, est simple car il n'y a qu'un envoi et une réception à faire sans besoin de se synchroniser, la communication entre plus de deux processeurs nécessite une synchronisation particulière. Lorsqu'une trame est envoyée sur le bus par un processeur, l'ensemble des autres processeurs reliés au bus la reçoit (broadcast). Le processeur destinataire récupère la trame et se charge de consommer la donnée. Pour les autres processeurs, à qui la trame n'est pas destinée, elle est reçue comme une trame de synchronisation nécessaire à l'ordonnancement des opérations mais n'est pas utilisée. Les synchronisations inter-processeurs sont gérées par l'opération SYNC d'envoi de messages (cf. section 7.5.1).

Le thread de communication est composé de plusieurs blocs de code identifiés par différentes étiquettes (labels) détaillées plus bas. L'accès aux labels se fait soit par l'appel de la fonction thread_canx(i) à partir du main, soit à partir de la routine d'interruption de communication déclenchée lors d'un envoi et d'une réception de messages sur le bus CAN. Le thread de communication contient cinq types de label :

- canx_init_0 : ce label permet d'initialiser le contexte CAN_ctxt_x du thread , de configurer la vitesse de transmission sur le CAN (baudrate), les buffers et le DMA (cf. documentation du pic33F [Inc11]), et de charger et lancer l'exécutable dans la mémoire du processeur à partir du programme de chargement et lancement (downloader),
- canx_empty_i : ce label correspond au sémaphore suc_empty permettant de libérer le sémaphore qui été bloqué pour attendre la réception de la donnée,
- canx_full_i : ce label correspond au sémaphore suc_full permettant de libérer le sémaphore qui avait été bloqué pour attendre l'envoi de la donnée,
- canx_sync_i : ce label correspond au message SYNC de la synchronisation inter-processeur permettant de faire des synchronisations dans le cas où le processeur n'est ni l'émetteur ni le récepteur d'une trame. Par conséquent on doit attendre de recevoir la trame mais la donnée n'est pas utilisée,
- canx_end_n : ce label permet de terminer la communication en désactivant le bus CAN (cf. documentation pic33F [Inc11]).

La routine d'interruption de communication s'exécute lorsque l'interruption qui signale un envoi ou une réception de trames est levée. Après avoir initialisé le

contexte de communication, le thread de communication appelle explicitement la routine d'interruption de communication chargée d'initialiser la communication :

- CAN_send_shared lors d'un envoi,
- CAN_recv_shared lors d'une réception,
- CAN_sync_shared lors d'une synchronisation.

Dans ce qui suit nous expliquons ce que nous avons dû ajouter au code C généré automatiquement par SynDEx et les compilateurs pic33f afin que ce code devienne tolérant aux fautes.

8.6 Tolérance aux fautes sur le banc de test

On présente séparément dans cette section l'implantation de la tolérance aux fautes des bus CAN et des processeurs sur le banc de test. En plus des hypothèses données dans la section 6.2.1 on fait les hypothèses suivantes :

Hypothèse 8.1 *La défaillance de chaque bus CAN est complète.*

Hypothèse 8.2 *La détection de fautes de bus CAN et de dsPICs se fait par des "watchdog" logiciels.*

Nous avons utilisé les principes suivants pour déterminer si un bus CAN et/ou un processeur sont fautifs. Les erreurs de processeurs et de bus sont détectées à l'aide d'interruptions générées par des "watchdogs" logiciels à partir de timers (matériels) (cf section timers du manuel d'utilisation du pic33f [Inc11]). On détecte l'erreur d'un bus lorsque le temps de communication d'une donnée sur ce dernier dépasse le pire temps de communication pour cette donnée. Pour cela avant chaque envoi (resp. réception) d'une donnée on initialise un watchdog, à l'aide de la fonction `initialize_timer`, avec une valeur supérieure ou égale au pire temps de communication, et on l'inhibe à l'aide fonction `inhibit_timer`, juste après avoir reçu (resp. transmis) cette donnée. Ainsi lorsqu'il y a une erreur de bus alors le temps de communication dépasse le pire temps de communication, et comme le watchdog n'a pas été inhibé, une interruption signalant une faute de bus est déclenchée.

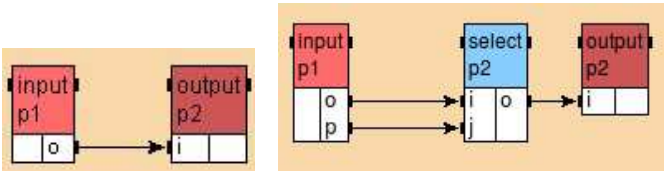
De la même manière, une erreur d'un processeur est détectée lorsque le temps de transmission de données vers/depuis un processeur dépasse le pire temps de communication. Ainsi une erreur de processeur est donc détectée lorsque le temps de communication dépasse le pire temps de communication. Notez que nous avons pris lors des expérimentations une valeur de pire temps de communication beaucoup plus grande que le temps de communication moyen mesuré sur un grand nombre de communications. On a utilisé, pour chaque processeur (resp. bus), une variable d'états `state_pi` (resp. `state_cani`) représentant l'état de chaque

processeur `pi` (resp. bus `cani`) de l'architecture : 0 si `pi` (resp. `cani`) est fautif et 1 sinon.

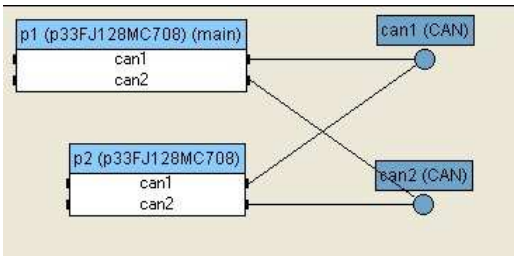
8.6.1 Tolérance aux fautes des bus CAN

Comme spécifié précédemment, une faute de bus est simulée en coupant l'interrupteur reliant le gate (communicateur) concerné à ce bus. Sur une application non tolérante aux fautes, cette opération provoque un blocage de l'application à cause des synchronisations inter-processeurs et intra-processeurs. L'objectif de la tolérance aux fautes de bus est donc de faire en sorte que si l'un des deux bus de l'architecture est fautif, l'autre bus non fautif puisse continuer de transmettre les données. Pour cela il faut d'abord détecter l'erreur de bus en identifiant le bus fautif. La détection et le traitement d'erreurs de bus CAN se font à l'aide de deux watchdogs (un watchdog par bus) chacun associé à un timer déclenchant lorsqu'il atteint sa valeur limite.

Nous illustrons la détection et le traitement d'erreurs avec une application SynDEx de base dont le graphe d'algorithme est constitué de deux opérations dépendantes "input" et "output" (figure 8.9(a)) et dont le graphe d'architecture est constitué de deux dsPICs "p1" et "p2" et deux bus CAN "can1" et "can2" (figure 8.9(b)). Afin de tolérer une faute de bus nous devons créer deux répliques de la dépendance de données $(input \triangleright output)$ et une opération de sélection "select" (figure 8.9(c)).



(a) Graphe d'algorithme initial (b) Graphe d'algorithme transformé



(c) Graphe d'architecture

FIGURE 8.9 – Graphes d'algorithme initial et transformé et graphe d'architecture pour la tolérance aux fautes de bus CAN

8.6.1.1 Détection d'erreurs

Afin de tolérer une erreur de bus CAN, nous devons modifier les codes sources des processeurs p1 et p2. Nous devons dans un premier temps initialiser les variables d'état des deux bus CAN "state_can1" et "state_can1" (cf. annexe C.1.1.1).

Ensuite nous devons encadrer la fonction de communication `thread_can1(1)` (resp. `thread_can2(1)`) par des fonctions de gestion de watchdog `initialize_timer1()` et `inhibit_timer1()` (resp. `initialize_timer2()` et `inhibit_timer2()`) (cf. annexe C.1.1.1).

Lorsqu'une erreur sur le bus est détectée, on met à jour les variables *state_canx*. Si le watchdog dépasse sa valeur limite sa routine d'interruption d'erreurs affecte la valeur 0 à la variable d'état correspondante pour indiquer que le bus est fautif (cf. annexe C.1.1.1).

8.6.1.2 Traitement des erreurs détectées

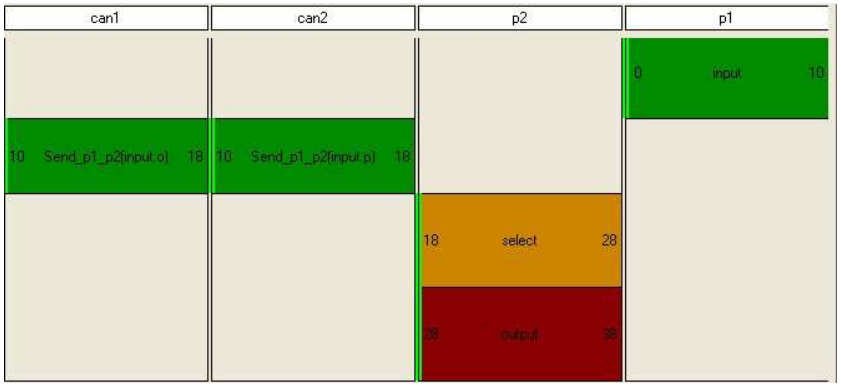
Une fois l'erreur détectée il faut débloquer dans un premier temps le sémaphore qui empêche la communication de se poursuivre, et dans un deuxième temps il faut inhiber l'opération de communication concernant le bus fautif.

Nous avons modifié la routine d'interruption pour libérer les sémaphores bloquées lors d'une faute de bus (cf. annexe C.1.1.2). Nous avons aussi ajouté un test sur la variable d'état de chaque bus de communication avant d'effectuer une les communication sur ce bus.

Le rôle du sélecteur est de choisir parmi ses entrées selon l'état du bus. Le code du sélecteur est donnée en annexe C.1.1.2.

La figure 8.10(b) représente les données transmises sur les bus CAN1 et CAN2. Lorsqu'on coupe le bus CAN1 à la douzième seconde, on remarque que le bus CAN1 continue à envoyer ses données et l'application continue à fonctionner en faisant clignoter des LEDs sur les cartes électroniques comportant de p1 et p2. Les données du CAN ont été récupérées à l'aide de l'analyseur de CAN Peak-CAN et du logiciel PCAN-Explorer [pca].

Remarque 8.1 *Pour tolérer les fautes d'un bus il est indispensable de modifier le code de tous les processeurs connectés à ce bus. En effet si un bus est fautif chaque processeur connecté à ce bus reste en attente d'une donnée qu'il va soit consommer soit considérer comme message de synchronisation. Si le code d'un processeur n'est pas modifié pour prendre en compte cette faute alors ce processeur reste bloqué ainsi que tous les autres processeurs connectés à ce bus fautif.*



(a) Adéquation

```
 6002.1  Rx        02AA  0
 7202.2  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
 8402.2  Rx        02AA  0
 8402.4  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
 9602.4  Rx        02AA  0
 9602.5  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
10802.5  Rx        02AA  0
10802.6  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
13641.1  Rx        02AA  0
13641.3  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
14841.4  Rx        02AA  0
14841.5  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
16041.5  Rx        02AA  0
16041.7  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
17241.7  Rx        02AA  0
17241.8  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
18441.9  Rx        02AA  0
18442.0  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
19642.1  Rx        02AA  0
19642.2  Rx        02A9  8  0B 0B 0B 0B 0B 0B 0B 0B
```

```
 7207.4  Rx        02AA  0
 8407.2  Rx        02A9  8  01 01 01 01 01 01 01 01
 9607.3  Rx        02AA  0
 9607.4  Rx        02A9  8  01 01 01 01 01 01 01 01
10807.6  Rx        02AA  0
10807.7  Rx        02A9  8  01 01 01 01 01 01 01 01
12007.9  Rx        02AA  0
```

(b) Trames CAN

FIGURE 8.10 – Adéquation et trames CAN lors d'erreur sur CAN2

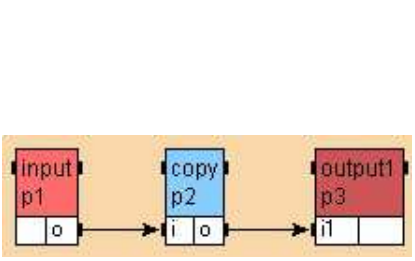
8.6.2 Tolérance aux fautes des dsPICs

De la même manière que pour les fautes de bus, les fautes de processeurs, sur cette architecture, sont simulées en coupant un des interrupteurs reliés à l'alimentation. Sur une application non tolérante aux fautes, cette opération provoque un blocage de l'application à cause des synchronisations inter-processeurs et intra-processeurs.

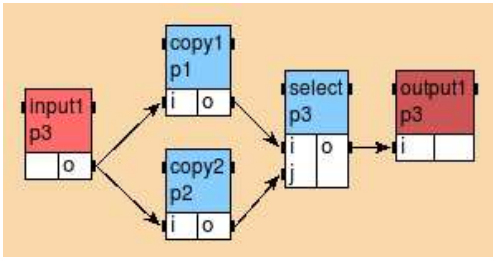
Afin de tolérer $N_{pf} = 1$ faute de processeur, il faut avoir au moins 2 processeurs et un seul bus CAN suffit. Comme on a plus de deux processeurs, la difficulté consiste donc à assurer leur synchronisation quelle que soit le nombre de processeurs non fautifs au cours de l'exécution (cf. section 8.5). Nous avons donc fait les modifications suivantes.

Nous illustrons la détection et le traitement d'erreurs avec une application SynDEx de base dont le graphe d'algorithme est constitué de trois opérations dépen-

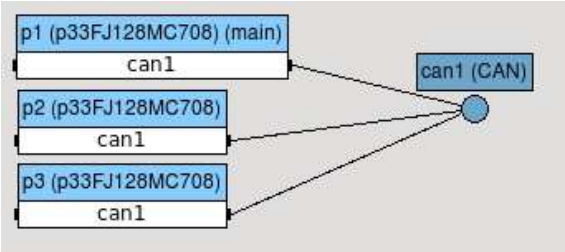
dantes "input", "copy" et "output" (figure 8.11(a)) et dont le graphe d'architecture est constitué de trois dsPICs "p1", "p2" et "p3" et d'un bus CAN "can1" (figure 8.11(b)). Afin de tolérer une faute de bus nous devons créer deux opérations répliques "copy1" et "copy2" une opération de sélection "select" (figure 8.11(c)).



(a) Graphe d'algorithme initial



(b) Graphe d'algorithme transformé



(c) Graphe d'architecture

FIGURE 8.11 – Graphes d'algorithme initial et transformé et graphe d'architecture pour la tolérance aux fautes des dsPICs

8.6.2.1 Détection d'erreurs

La détection d'erreurs de processeurs utilise le même principe (watchdog) que celle des erreurs de bus, en encadrant certaines parties du code C généré par les fonctions `initialize_timer` et `inhibit_timer`.

Dans le cas des processeurs, encadrer l'appel du thread ne suffit pas, car celui-ci contient l'ensemble des communications impliquant des processeurs différents. Il faut donc se concentrer, à l'intérieur du thread de communication du processeur considéré, sur la communication avec un autre processeur pouvant être fautif.

Afin de tolérer une erreur de dsPIC, nous devons modifier les codes sources de tous les dsPICs. Nous devons dans un premier temps initialiser les variables d'état des deux dsPICs "state_p1", "state_p2" et "state_p3" sur chaque processeur (cf. annexe C.1.2.1).

Lorsqu'une erreur sur un dsPIC est détectée, on met à jour les variables *state_px*. Si le watchdog dépasse sa valeur limite sa routine d'interruption d'erreurs affecte

la valeur 0 à la variable d'état correspondante pour indiquer que le dsPICs est fautif (cf. annexe C.1.2.1).

8.6.2.2 Traitement des erreurs détectées

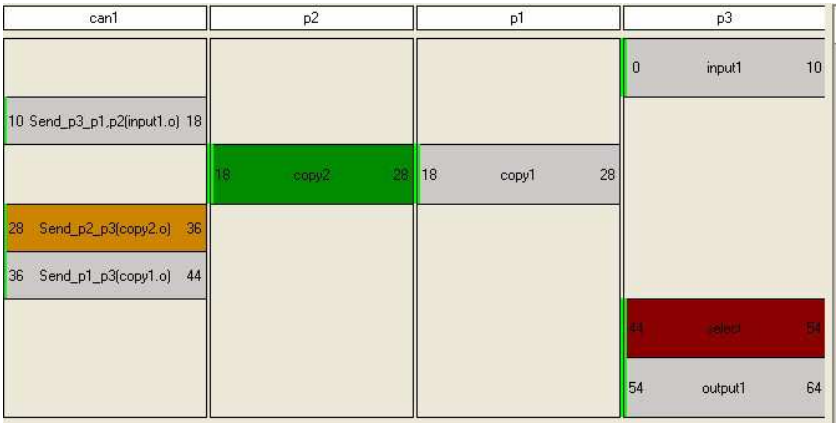
Comme nous visons une application de suivi sur l'architecture hétérogène du CyCab (MPC555, PC Linux/RTAI, dsPIC), nous devrions modifier le code source de chacun processeur afin d'en tolérer des erreurs. Cependant il s'est avéré très complexe de modifier les codes sources des MPC555 et du PC embarqué car ils sont écrits en assembleur. Nous avons donc modifié uniquement le code C des dsPICs.

On commence par initialiser l'état de tous les processeurs à 0 (state_px = 0), ensuite on intègre les routines d'interruptions d'erreurs. Lorsque le processeur p1 (resp. p2) envoie une donnée au processeur p3, le processeur p2 (resp. p1) reçoit une trame de synchronisation sans utiliser la donnée. Si cette synchronisation échoue sur le processeur p1 (resp. p2) alors le processeur p2 (resp. p1) est fautif. Pour résoudre ce problème nous avons proposé la solution suivante. La gestion de la faute se fait en envoyant la donnée une deuxième fois à partir du processeur non fautif pour assurer les synchronisations pour les autres processeurs de l'architecture (p3 dans notre cas) (cf. annexe).

Remarque 8.2 *Nous avons proposé cette solution pour pouvoir contourner le problème de tolérance aux faute de processeurs sans avoir à modifier le code source de tous les processeurs de l'architecture. En effet s'il y a un processeur fautif qui doit envoyer une donnée à un autre processeur, alors ce dernier reste en attente de cette donnée et les autre processeurs, qui ne consomment pas cette données, en ont besoin pour assurer la bonne synchronisation. Mais lorsque cette donnée est envoyée par un autre processeur (non fautif), alors elle est reçue par le processeur qui la consomme, et elle servira de message de synchronisation pour les autres processeurs. Ces derniers ne s'aperçoivent même pas qu'il y a un processeur fautif dans l'architecture.*

8.7 Tolérance aux fautes sur le CyCab : application de suivi

Nous considérons dans cette partie l'architecture du CyCab tolérant aux fautes constituée de trois dsPICs, de deux MPC555 et d'un PC embarqué (figure 8.8). Nous avons présenté dans la section précédente une solution permettant de tolérer les fautes de processeurs sans avoir à modifier le code source de tous les processeurs (remarque 8.2). Cependant pour tolérer les fautes de bus il est indispensable



(a) Adéquation

15:37:59.9531	Rx	2A8h	0	
15:37:59.9532	Rx	2A9h	0	
15:38:00.0732	Rx	2AAh	8	00 01 02 03 04 05 06 07
15:38:00.0733	Rx	2AAh	0	
15:38:00.1932	Rx	2A9h	8	00 01 02 03 04 05 06 07
15:38:00.1932	Rx	2AAh	0	
15:38:00.1934	Rx	2A8h	8	00 01 02 03 04 05 06 07
15:38:00.1934	Rx	2A8h	0	
15:38:00.1935	Rx	2A9h	0	
15:38:00.4335	Rx	2AAh	8	00 01 02 03 04 05 06 07
15:38:00.4336	Rx	2AAh	0	
15:38:02.0723	Rx	2AAh	0	
15:38:02.0725	Rx	2A8h	8	00 01 02 03 04 05 06 07
15:38:02.0725	Rx	2AAh	0	
15:38:02.0727	Rx	2A8h	8	00 01 02 03 04 05 06 07
15:38:02.0728	Rx	2A8h	0	
15:38:02.3128	Rx	2AAh	8	00 01 02 03 04 05 06 07
15:38:02.3129	Rx	2AAh	0	
15:38:02.4330	Rx	2A8h	8	00 01 02 03 04 05 06 07
15:38:02.4331	Rx	2AAh	0	
15:38:02.4332	Rx	2A8h	8	00 01 02 03 04 05 06 07

(b) Trames CAN

FIGURE 8.12 – Adéquation et trames CAN lors d'erreur sur p3

de modifier le code de tous les processeurs (remarque 8.1), cela reviendrait à refaire tous les travaux effectués en C sur les dsPICs pour les codes assembleurs des MPC555 et le PC embarqué. Comme ces modifications sont très complexes, on se limite uniquement à la tolérance aux fautes des dsPICs.

Pour cela, il faut dans un premier temps, assurer la communication entre les dsPICs et les MPC555 afin que l'échange de données se fasse correctement. Dans un deuxième temps, on élimine les conflits d'identification des processeurs avec PCANExplorer afin de rendre plus claire la visualisation des trames échangées. Enfin, on intègre les redondances d'opérations dans les applications monopériode

et multipériode.

8.7.1 Communication entre dsPICs et MPC555

Pour parvenir à faire communiquer correctement un dsPIC et un MPC555, il faut prendre en compte le fait que le premier possède une architecture 16bits, et le deuxième possède une architecture 32bits. Cette différence d'architecture est significative au niveau de la définition des types sur les deux opérateurs : un *int* sera défini sur 2 octets sur un dsPIC, et sur 4 octets sur un MPC555. Concrètement, si un MPC555 envoi une donnée stockée dans un *int* à un dsPIC, ce dernier ne récupérera que la moitié de la donnée.

Pour éviter la confusion des types entre les processeurs d'architecture différente, on définit un type partagé intermédiaire *longint*, codé sur 4 octets (équivalent d'un *int* sur un MPC555, et l'équivalent d'un *long* sur un dsPIC). Ainsi, les variables des applications de base du CyCab restent inchangées et celles des dsPICs sont codées en *longint*. Il faut ensuite définir des fonctions *int2longint* et *longint2int* qui prennent un *int* (resp. *longint*) en entrée et un *longint*(resp. *int*) en sortie et qui servent d'intermédiaire pour toute communication entre un dsPIC et un MPC555 ou Linux/RTAI. Elles permettent de contourner la contrainte de SynDEx qui n'autorise pas deux ports de types différents à communiquer. Les modifications à apporter aux fichiers existants sont données dans l'annexe C.2.1.

Au cours des premiers tests sur le CyCab, PCANExplorer présentait des conflits au niveau des valeurs d'identification : plusieurs processeurs possédaient le même identifiant(CANId), rendant l'analyse des trames compliquée. On a résolu en attribuant aux processeurs un identifiant différent pour le bootloading et pour la communication. La résolution de ce problème est présentée dans l'annexe C.2.1.

Pour des raisons pratiques, les tests sur le CyCab ont d'abord été effectués en monopériode avant de passer au multipériode. Le principe des tests sur CyCab est de reprendre une application CyCab existante fonctionnelle, d'intégrer les dsPICs dans l'architecture initiale et de refaire l'adéquation de manière à ce que le code soit distribué sur les dsPICs. Dans les applications qui suivent, les fonctions attribuées aux dsPICs constistent simplement à copier une donnée reçue et à l'envoyer vers l'opérateur suivant. L'objectif de ces tests est de parvenir à faire fonctionner l'application de suivi du CyCab même en présence de fautes de dsPIC quelles que soient les fonctions que ceux-ci réalisent (simples ou compliquées). On pose les hypothèses qu'un seul dsPIC peut être fautif $N_{pf} = 1$.

8.7.2 Application SynDEx de base de suivi automatique de CyCabs

Afin de tester la tolérance aux fautes sur le CyCab, nous avons modifié le graphe d'algorithme du suivi automatique du CyCab présenté dans la figure 8.6 et le graphe d'architecture présenté dans la figure 8.4.

Pour cela nous avons codé l'opération "SpeedControl", initialement codée pour le mictrocontrôleur MPC555, pour le dsPICs. Nous avons ajouté des opérations de conversions de données "conv1", "conv2" et "conv3", comme le montre la figure 8.13, car les MPC555 et les dsPICs n'utilisent pas le même codage de données. Les opérations "TI" et "conv1" sont allouées à l'opérateur "root", l'opérations "SpeedControl" est allouée à l'opérateur "dspic1" et le reste des opérations sont allouées aux opérateurs "f555" et "r555".

La durée d'exécution du traitement d'images varie entre $70ms$ et $100ms$ alors que celle de l'opération de conversion "conv1" est négligeable devant le traitement d'images. La somme de ces deux durées d'exécutions est du même ordre que celle du traitement d'images. Afin que ces deux opérations puissent être ordonnancées sur le même opérateur "root" ils faut qu'elles satisfassent la condition (3.12) : $C_1 + C_2 \leq PGCD(T_1, T_2)$. Comme $C_1 + C_2 \simeq C_1 \leq 100ms$ et $PGCD(T_1, T_2) = 200ms$ alors la condition (3.12) est satisfaite et ces deux opérations sont ordonnançables sur le même processeur.

Le reste des opérations de contrôle ont toutes une période de $10ms$ et sont ordonnancées sur le reste des processeurs.

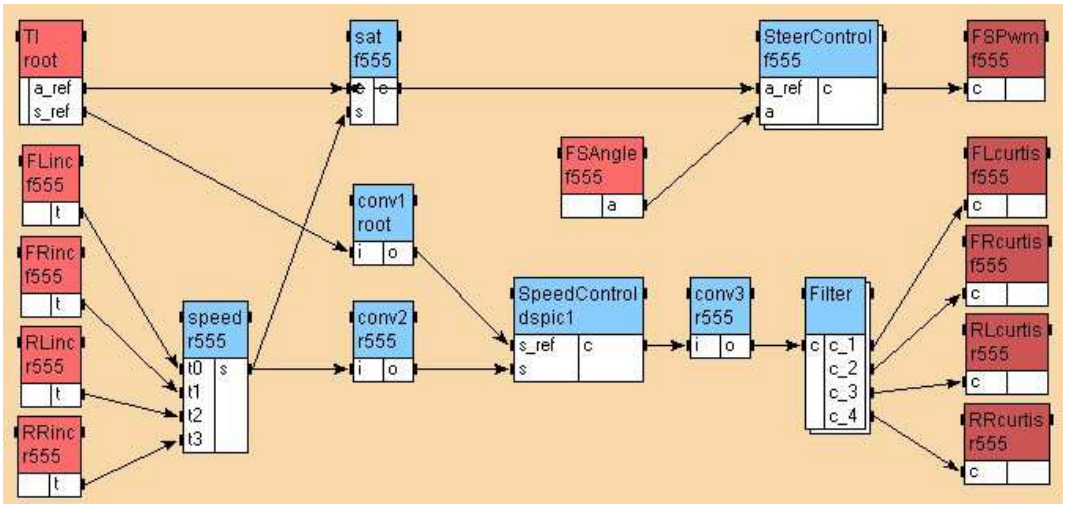


FIGURE 8.13 – Graphe d'algorithme de l'application de suivi automatique multipériode de base

Le graphe d'architecture de cette application est montré par la figure 8.14.

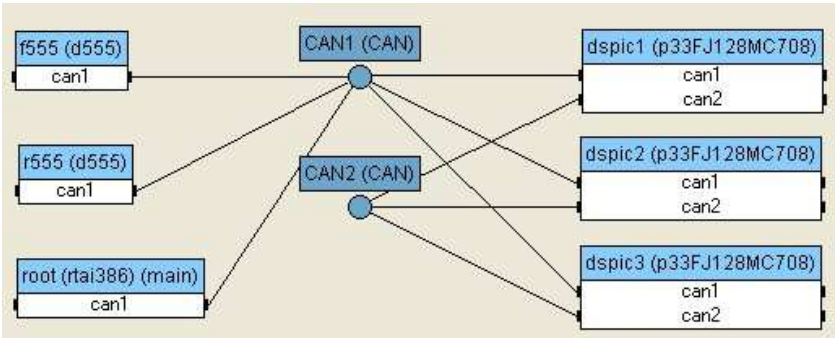


FIGURE 8.14 – Graphe d'architecture de l'application de suivi automatique multipériode de base

8.7.3 Tolérance aux fautes des bus CAN

Nous avons utilisé les mêmes principes que ceux présentés dans la section 8.6.1 pour tester la tolérance aux fautes des bus CAN. Afin de tolérer une faute du bus CAN1 ou une faute du bus CAN2, nous avons répliqué une fois la dépendance de données ("SpeedControl" ▷ "conv3") apparaissant sur la figure 8.13, en utilisant une opération "copy". Cette opération doit être allouée au même opérateur "dspic1" que celui sur lequel est allouée l'opération "SpeedControl". Nous avons ensuite utilisé un sélecteur "Sélect" alloué à l'opérateur "dspic2". Cette transformation apparaît sur la figure 8.15.

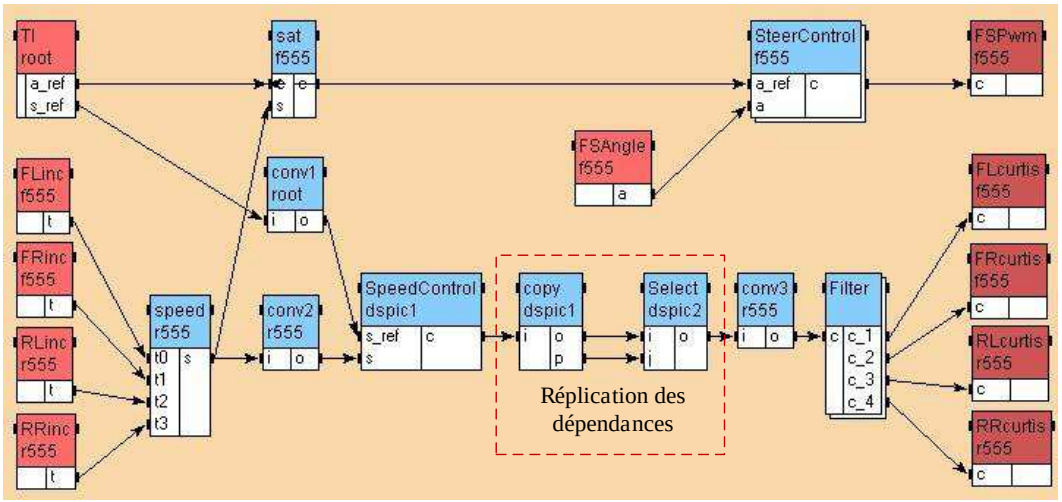


FIGURE 8.15 – Graphe d'algorithme de l'application de suivi automatique multipériode tolérante aux fautes de bus CAN

Le graphe d'adéquation est donné par la figure 8.17. Il montre que le traite-

ment d'images alloué sur le processeur "root" s'exécute une fois lorsque toute les autres opérations de contrôle/commande et les communications reliant le traitement d'images et ces dernières, s'exécutent vingt fois sur les autres microcontrôleurs. Cela est dû à la période du traitement d'images qui est de $200ms$ alors que celle des autres opérations est de $10ms$. Le respect des périodes est assuré par des "Wait" débutant à la fin dé l'exécution de chaque opération et ayant comme durée la différence entre sa période et sa durée d'exécution. La communication de la deuxième dépendance de données ("copy" $\triangleright$ "Select") est allouée au bus CAN2 (voir les vingt communications de la colonne CAN2 de la figure 8.17). C'est cette communication qui permet d'assurer la tolérance aux fautes d'un des deux bus CAN.

8.7.4 Tolérance aux fautes des dsPICs

Nous avons utilisé les mêmes principes que ceux présentés dans la section 8.6.2 pour tester la tolérance aux fautes des dsPICs. Nous supposons que l'un des opérateurs "dspic1" ou "dspic2" peut être fautif. Pour tolérer une faute de dsPIC nous avons répliqué l'opération "SpeedControl". Cela a conduit à deux opérations "SpeedControl1" et "SpeedControl2" allouées respectivement aux opérateurs "dspic1" et "dspic2". Des opérations "copy1" et "copy2" allouées à l'opérateur "dspic3" ont été ajoutées pour renvoyer une deuxième fois la donnée qui n'a pas été effectivement envoyée par l'opérateur fautif (cf. section 8.6.2.2). Une opération de sélection allouée à "dspic3" permet de sélectionner entre les sorties des opérations "SpeedControl1" et "SpeedControl2". Le graphe d'algorithme transformé pour la tolérance aux fautes est donné par la figure 8.16. Cette transformation de graphe nous permet de tolérer une faute de processeur "dspic1" ou "dspic2".

Le graphe d'adéquation est donné par la figure 8.18. Comme pour la tolérance aux fautes des bus CAN, cette figure montre que le traitement d'images alloué sur le processeur "root" s'exécute une fois lorsque toute les autres opérations de contrôle/commande et les communications reliant le traitement d'images et ces dernières, s'exécutent vingt fois sur les autres microcontrôleurs. L'opérateur "dspic1" (resp. "dspic2") exécute vingt fois l'opération "SpeedControl1" (resp. "SpeedControl2") que l'on peut voir sur la colonne dspic1 (resp. dspic2) de la figure 8.18. Cela permet d'assurer la tolérance aux fautes d'un des processeurs dsPICs.

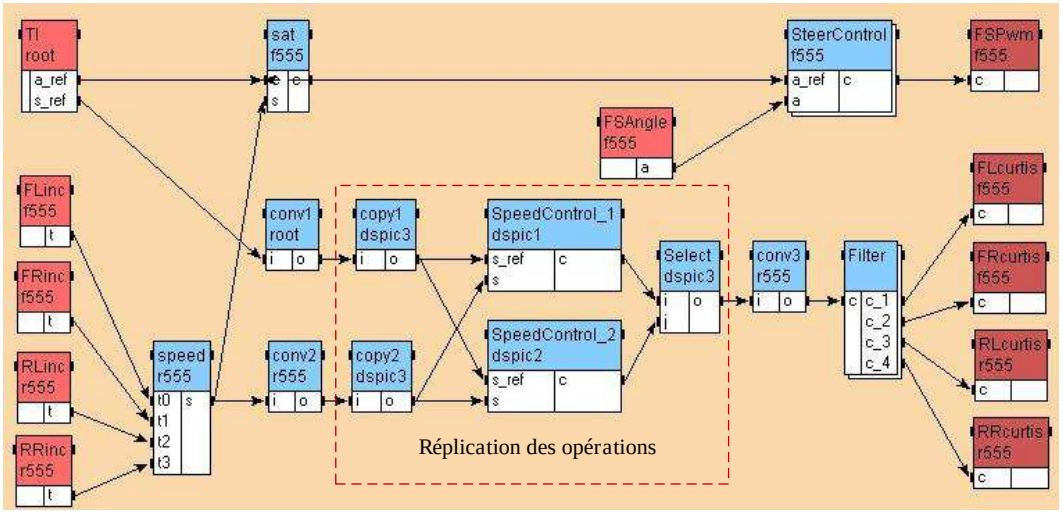


FIGURE 8.16 – Graphe d'algorithme de l'application de suivi automatique multi-période tolérante aux fautes de dsPICs

8.8 Conclusion

Nous avons présenté dans ce chapitre l'application de suivi de CyCabs tolérante aux fautes. Nous avons d'abord décrit le banc de test à base de dsPICs que nous avons réalisé et sur lequel nous avons testé la tolérance aux fautes des dsPICs et de bus CAN. Nous avons ensuite modifié l'architecture des CyCabs en y intégrant notre banc de test et nous avons testé la tolérance aux fautes des dsPICs et des bus CAN sur une application de suivi de CyCab comportant un algorithme de traitement d'images effectuant la détection du CyCab suivi, et des lois de commande permettant de maintenir une distance minimale entre le CyCab suivi et le CyCab suiveur.

FIGURE 8.17 – Adéquation de l'application de suivi automatique multipériode tolérante aux fautes de bus CAN

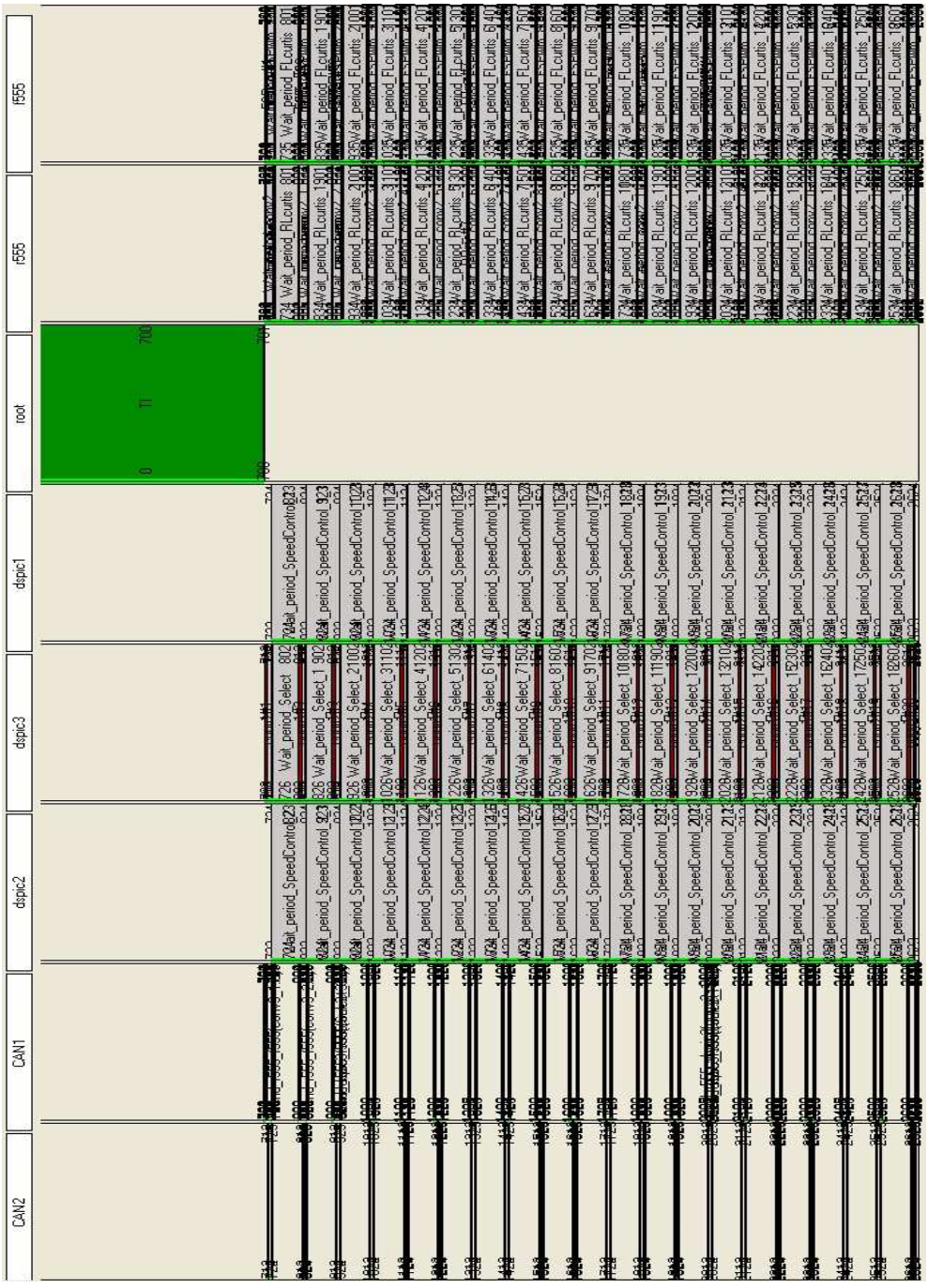


FIGURE 8.18 – Adéquation de l'application de suivi automatique multipériode tolérante aux fautes de dsPICs

Conclusion générale et perspectives

Conclusion générale

Nous avons d'abord présenté un état de l'art sur les systèmes temps réel embarqués. Nous avons commencé par donner des définitions concernant les architectures des systèmes embarqués ainsi que les contraintes temporelles et matérielles, puis nous avons présenté le modèle de tâches temps réels classique et celui de tâches temps réel NPPS. Comme nous nous intéressons aux applications de contrôle/commande temps réel critiques, les traitements de capteurs/actionneurs et les traitements de commande de procédés ne doivent pas avoir de gigue sur les entrées issues des capteurs et sur les sorties fournies aux actionneurs. Pour ces raisons nous avons considéré les tâches NPPS. Nous avons ensuite présenté les différentes conditions d'ordonnançabilité et les algorithmes d'ordonnancement associés, monoprocesseur et multiprocesseur. Enfin nous avons présenté le graphe d'algorithme flot de données décrivant le système de tâches temps réel dépendantes et le graphe d'architecture distribuée que nous allons utiliser dans la suite. Ensuite nous avons présenté les terminologies et les principes de la tolérance aux fautes. Comme nous nous intéressons qu'aux fautes matérielles (processeurs et média de communication), nous avons présenté deux types de redondances pour la tolérance aux fautes matérielles : redondance logicielle et redondance matérielle. Pour la redondance logicielle nous avons présenté trois types de redondances : active, passive et hybride que nous avons comparées. Enfin nous avons présenté les principes de la redondance matérielle basée sur la redondance modulaire.

À partir de cet état de l'art nous avons proposé une analyse d'ordonnançabilité de tâches NPPS dans le cas monoprocesseur. Nous avons présenté dans un premier temps une stratégie d'ordonnancement qui consiste à ordonnancer une tâches candidate avec un ensemble de tâches déjà ordonnancées. Nous avons utilisé cette même stratégie dans les deux cas de tâches harmoniques et non harmoniques. Puis nous avons donné un théorème permettant de calculer la phase transitoire et la phase permanente d'un ordonnancement de tâches NPPS. Ensuite nous avons distingué deux sous-cas des tâches harmoniques : des tâches ayant toutes des périodes distinctes, et des tâches ayant certaines périodes identiques ;

le second cas est plus compliqué à étudier que le premier. Nous avons proposé un théorème qui donne une condition d'ordonnançabilité nécessaire et suffisante dans le premier cas, et un théorème qui donne une condition d'ordonnançabilité suffisante dans le second cas. Finalement pour le cas général qui est le cas des tâches non harmoniques, nous avons présenté une étude d'ordonnançabilité pour plus de deux tâches en proposant un théorème qui donne une condition d'ordonnançabilité moins restrictive que la condition d'ordonnançabilité existante. Nous avons aussi présenté une étude d'ordonnançabilité d'une combinaison de tâches NPPS et de tâches PP, ayant toutes des priorités fixes. Les tâches NPPS ont toutes la même plus haute priorités, et les tâches PP ont des priorités croissantes en fonction de leurs échéances relatives (algorithme d'ordonnancement Deadline Monotonic). Nous avons d'abord montré comment construire l'ensemble des instants critiques correspondant aux dates de début d'exécution des tâches NPPS, puis nous avons proposé deux lemmes permettant de ne considérer que les instants critiques appartenant à la phase permanente, i.e. de ne pas considérer les instants critiques de la phase transitoire, et de ne garder que le premier instant critique lorsque plusieurs instances s'exécutent sans temps creux dans la phase permanente. Ensuite nous avons donné un théorème qui permet de calculer le temps demandé au processeur pour une tâche PP combinée avec les tâches plus prioritaires (périodiques strictes et non strictes). Finalement nous avons proposé un théorème qui donne une condition d'ordonnançabilité nécessaire et suffisante, calculant itérativement les pires temps de réponse pour chaque tâche PP et les comparant à leurs échéances relatives.

Nous avons ensuite proposé une analyse d'ordonnançabilité multiprocesseur avec une approche d'ordonnancement partitionnée qui transforme le problème d'ordonnancement multiprocesseur en un problème d'ordonnancement monoprocesseur via les trois algorithmes suivants. L'algorithme d'analyse d'ordonnançabilité permet de faire une étude d'ordonnançabilité monoprocesseur et d'assigner chaque tâche sur éventuellement plusieurs processeurs sur lesquels cette dernière est ordonnançable. L'étude d'ordonnançabilité a été présentée précédemment. L'algorithme de déroulement permet de transformer le graphe d'algorithme en un graphe déroulé où chaque tâche est répétée un nombre de fois égal au rapport entre l'hyper-période et sa période. L'algorithme d'ordonnancement exploite les résultats des deux algorithmes précédents pour choisir sur quel processeur ordonnancer une tâche assignée à plusieurs processeurs et calculer sa date de début d'exécution en prenant en compte les durées de communications inter-processeurs.

Nous avons ensuite proposé d'étendre l'étude d'ordonnançabilité temps réel multiprocesseur précédente pour qu'elle soit tolérante aux fautes des processeurs et de bus de communication. Nous avons commencé par présenter le modèle de fautes et les hypothèses de tolérance aux fautes que nous considérons. Puis nous

avons présenté la transformation du graphe d'algorithme pour la tolérance aux fautes qui ajoute des tâches et des dépendances de données répliques, des tâches de sélection permettant de choisir la réplique de tâches allouée à un processeur non fautif ainsi que des relations d'exclusion permettant de ne pas allouer deux tâches (resp. dépendances) répliques au même processeur (resp. bus de communication). Nous avons étudié séparément les problèmes de tolérance aux fautes pour des processeurs, des bus de communication et enfin des processeur et des bus de communication. Ensuite nous avons présenté l'ordonnancement tolérant aux fautes qui est composé de trois algorithmes : l'algorithme d'analyse d'ordonnançabilité, de déroulement et d'ordonnancement. Afin de prendre en compte les relations d'exclusions générées par la transformation de graphe, nous avons modifié profondément l'algorithme d'analyse d'ordonnançabilité et l'algorithme d'ordonnancement existants en y intégrant les conditions d'ordonnançabilité et les calculs de dates de début d'exécution. En revanche nous n'avons pas eu à modifier l'algorithme de déroulement existant.

Nous avons ensuite présenté les améliorations apportées au logiciel SynDEx programmé en OCaml tant sur le plan de l'analyse d'ordonnançabilité et de l'ordonnancement que sur le plan de la tolérance aux fautes.

Finalement nous avons présenté les travaux expérimentaux concernant l'application de suivi de CyCabs. Nous avons réalisé un banc de test à base de dsPICs sur lequel nous avons testé la tolérance aux fautes des dsPICs et de bus CAN. Nous avons ensuite modifié l'architecture des CyCab en y intégrant notre banc de test et nous avons testé les tolérances aux fautes des dsPICs et du bus CAN sur une application de suivi de CyCab comportant un algorithme de traitement d'images effectuant la détection du CyCab suivi et des lois de commande permettant de maintenir une distance minimale entre le CyCab suivi et le CyCab suiveur.

Perspectives

On peut envisager plusieurs perspectives pour les travaux présentés dans cette thèse, tant sur le plan de l'ordonnancement que sur celui de la tolérance aux fautes.

Sur le plan de l'ordonnancement monoprocesseur de tâches NPPS, un moyen intéressant d'améliorer le taux de succès de l'ordonnancement serait de relâcher la contrainte de périodicité stricte pour certaines tâches ne traitant pas les entrées/sorties, tout en restant non préemptif. Il serait aussi intéressant d'étudier le taux de succès de l'ordonnancement des combinaisons possibles de tâches (non) préemptives (non) périodiques (non) strictes.

Dans le cas multiprocesseur, il serait intéressant de fusionner les deux algorithmes d'analyse d'ordonnançabilité et d'ordonnancement de telle sorte à assigner chaque tâche candidate aux processeurs sur lesquels elles est ordonnançable,

puis à l'ordonnancer sur celui où elle maximise la fonction de coût et la désassigner des autres processeurs. L'inconvénient de cette fusion est qu'elle ralentit l'exécution lorsque le système de tâches n'est pas ordonnançable. Pour améliorer le taux de succès de l'ordonnancement des tâches NPPS, il serait intéressant d'étudier une version semi-partitionnée des travaux présentés, en faisant migrer certaines instances de tâches qui se chevauchaient avec d'autres instances déjà ordonnnancées sur un processeur vers d'autres processeurs où elles pourrons être ordonnancées. Cela pourrait aussi être étendu dans le cas où l'on combine des tâches préemptives et non préemptives. Alors que nos travaux sur l'ordonnancement multiprocesseurs ont été fait dans le cas où le nombre de processeur est fixé, il serait intéressant de traiter le problème à nombre de processeurs variable en cherchant à le minimiser. Ce problème peut être résolu par dichotomie en commençant par un nombre de processeur égal au nombre de tâches.

Sur le plan de la tolérance aux fautes, il serait intéressant de prendre en compte les fautes des capteurs, soit à l'aide de la redondance matérielle de capteurs, soit par la reconstitution des données du capteur fautif à l'aide d'autres capteurs (fusion de données). Il serait aussi intéressant de prendre en compte la redondance passive en ayant des tâches primaires et des tâches de sauvegarde. Cela permettrait de diminuer la charge des processeurs mais nécessiterait des mécanismes d'activation des tâches de sauvegarde complexes. Il serait aussi intéressant de prendre en compte les fautes partielles du bus CAN afin de profiter au maximum des parties non fautives du bus. Notre approche d'ordonnancement tolérante aux fautes reste valable dans le cas des fautes intermittentes. Cependant cette approche reste à tester sur l'application de suivi de CyCabs.

Enfin il serait utile de tester nos travaux sur des applications robotiques plus complexes que le suivi de CyCabs utilisé dans le cadre de cette thèse.

Cinquième partie

Annexes

Annexe A

Caractéristiques générales du CyCab : PC embarqué Linux/RTAI, microcontrôleurs MPC555 et dsPIC33

A.1 PC embarqué

Le CyCab contient un PC embarqué qui communique avec les 2 MPC555 via le bus CAN 0. Le PC des CyCabs de l'équipe IMARA possédait un processeur Intel à 233 MHz, sous Linux temps réel (kernel 2.4, RTAI 2.4), ce qui était suffisant pour faire tourner l'application de conduite manuelle. Sa vitesse était trop faible pour faire du traitement de l'image, et donc par conséquent, de faire du suivi automatique basé sur une caméra.

Par conséquent, le PC embarqué du CyCab de l'équipe AOSTE a été mis à jour par un stagiaire. Il est désormais constitué de :

- le même châssis alimenté par une tension -48 V,
- une carte mère contenant un Pentium 4 cadencé à 3 GHz et 512 Mo de mémoire vive,
- une carte de fond de panier, référence PCI-5SDA-RS-R30 à 2 slots PCI et 4 ISA. Une nouvelle version contient 4 slots PCI,
- une carte SPI pour la communication CAN,
- une carte SPI pour la communication IEEE 1394,
- une carte graphique PCI NVidia GX 5200, la puce graphique incluse dans la carte mère est suffisante pour un affichage confortable.
- un disque dur IDE à 20 Go.

Le système d'exploitation du PC est un Linux temps réel dont la distribution

est une Debian 4.0 release 0. Le noyau Linux installé par défaut par Debian est le 2.6.18. Un noyau 2.6.18.52 a été compilé et patché pour le temps réel avec RTAI 3.4.

A.2 Microcontrôleur MPC555

Les MPC555 du CyCab sont constitués de quatre parties démontables dont nous allons expliquer l'utilité :

- une carte mère,
- une carte fille s'emboîtant sur la carte mère,
- un micro-contrôleur MPC555,
- une petite boîte de couleur noire.

Les entrées-sorties de la carte mère (à gauche, figure A.1), sont :

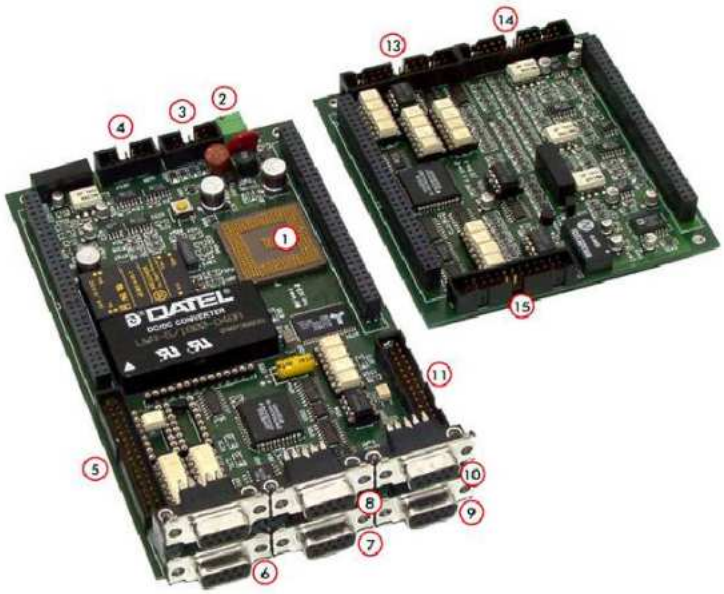


FIGURE A.1 – Carte mère (à gauche) et fille (à droite)

1. l'emplacement du MPC555. Par convention avec les autres CyCab de l'équipe IMARA, on donnera l'identifiant $0x4000$ au processeur du MPC555 arrière et l'identifiant $0x4001$ au processeur du MPC555 avant,
2. l'emplacement de l'alimentation. Selon l'ancienneté du CyCab, la tension délivrée par l'alimentation est soit 24 Vdc, soit de 48 Vdc,

3. l'est l'entrée de la boîte de couleur noire. Sans cette boîte le CyCab ne peut démarrer. Selon les dires de Robosoft, elle permet de remettre à zéro le MPC555 lorsque le bouton arrêt d'urgence est enfoncé,
4. l'entrée du joystick (uniquement pour le MPC555 arrière du CyCab),
5. les entrées-sorties numériques,
6. l'entrée/sortie sérielle de type SPI vers l'encodeur absolu. Seul le connecteur du bas utilisé,
7. ne sert pas,
8. ne sert pas,
9. l'entrée/sortie CAN 0 permettant la communication avec l'autre MPC555 et le PC embarqué. Par convention avec les autres CyCab de l'équipe IMARA, la vitesse du bus CAN est de 800 Kbit/s,
10. l'entrée/sortie CAN 1 permettant de brancher des équipements externes au CyCab (PC portable, par exemple).

Figure label	Description
1	MPC555
2	Power supply : 18-60V DC (from batteries)
3	BDM Interface (Basic Debug Interface)
4	7 analog inputs
5	16 logical inputs and 20 logical outputs
6	Synchronous serial line (SPI)
7	Asynchronous serial lines (port 0)
8	Asynchronous serial lines (port 1)
9	CAN bus (port 0)
10	CAN bus (port 1)
from 11 to 14	4 connectors dedicated to axis control (including 1 analog output per axis)

TABLE A.1 – Principales caractéristiques du MPC555

A.3 Microcontrôleur dsPIC33FJ128MC708

Le tableau A.3 regroupe les principales caractéristiques du microcontrôleur dsPIC33FJ128MC708.

Paramètre	Description
Architecture	16-bit
CPU Speed (MIPS)	40
Memory Type	Flash
Program Memory (KB)	128
RAM Bytes	16,384
Temperature Range C	-40 to 125
Operating Voltage Range (V)	3 to 3.6
I/O Pins	69
Pin Count	80
System Management Features	PBOR
POR	Yes
WDT	Yes
Internal Oscillator	7.37 MHz, 512 kHz
nanoWatt Features	Fast Wake/Fast Control
Digital Communication Peripherals	2-UART, 2-SPI, 2-I2C
Codec Interface	NO
Analog Peripherals	2-A/D 18x10-bit 1(ksps)
Op Amp	NO
CAN (#, type)	2 ECAN
Capture/Compare/PWM Peripherals	8/8
PWM Resolution bits	16
Motor Control PWM Channels	8
Quadrature Encoder Interface (QEI)	1
Timers	9 x 16-bit 4 x 32-bit
Parallel Port	GPIO
Hardware RTCC	No
DMA	8
XLP	NO

TABLE A.2 – Principales caractéristiques du dsPIC33FJ128MC708

La figure A.2 montre le banc de test à base de trois dsPIC33FJ128MC70.

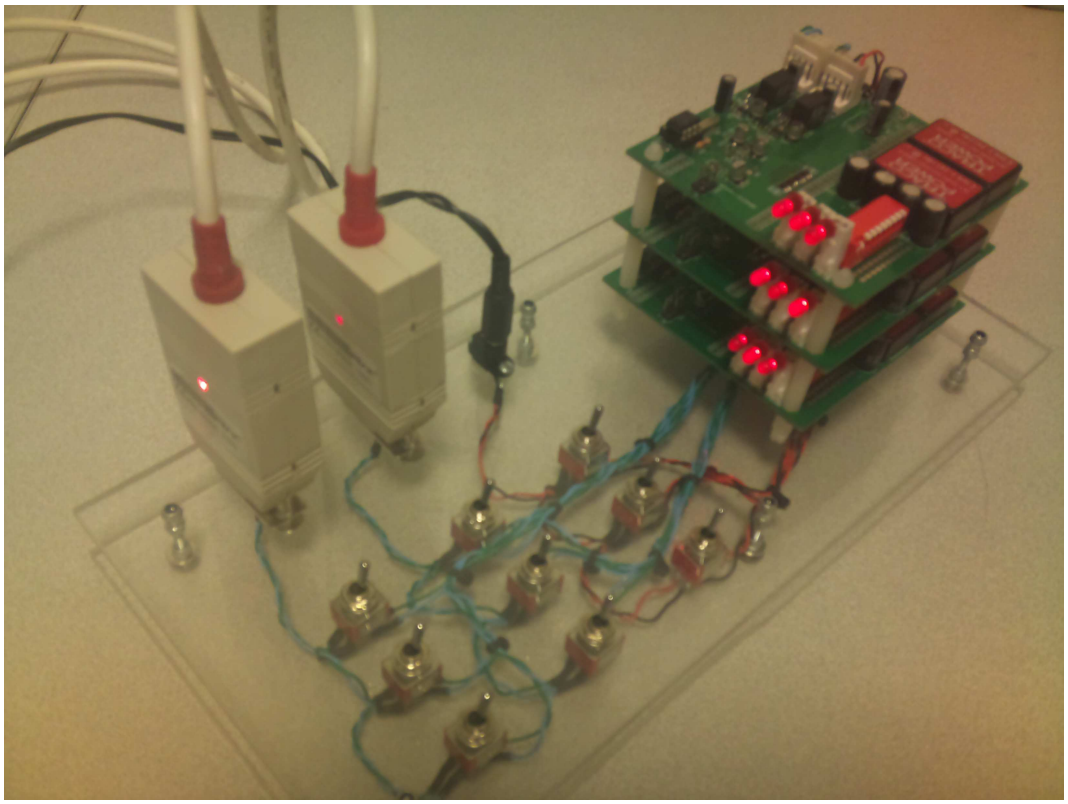


FIGURE A.2 – Banc de test à base de trois dsPIC33FJ128MC708

Annexe B

Code source des dsPIC pour les communications inter-processeurs

Le thread de communication du bus d'indice x (thread_canx) est associée à un **contexte** (CAN_ctxt_x) qui permet de caractériser la communication CAN à tout moment. Ci-dessous le code source généré par SynDEx.

```
typedef struct CAN_ctxt_s CAN_ctxt_;
struct CAN_ctxt_s {
//nom du thread auquel le contexte est associé
void (*thread)(volatile unsigned);
//indice du prochain label auquel accéder
volatile unsigned iptr;
//pointeur sur le handler d'interruption en cours d'utilisation
void (*stat)(volatile CAN_ctxt_*);
//adresse de la base de la DMA
volatile unsigned* base;
//pointeur sur la mémoire allouée dans la DMA
volatile unsigned* msgBuf;
//nombre de trames de synchronisation attendues par le processeur
volatile unsigned nsyn;
//taille de la donnée en cours de traitement
volatile int size;
//pointeur sur le buffer de stockage de la donnée traitée
volatile uchar *addr;
};
```

Le contexte CAN_ctxt_x du thread est initialisé comme suit :

```
CAN_ctxt_1.thread = thread_can1;
```

```
CAN_ctxt_1.msgBuf = CAN_buf_1;
CAN_ctxt_1.base = (uint *)CAN_base_1;
CAN_ctxt_1.nsyn = 0;
CAN_ctxt_1.stat = CAN_init_state;
```

La partie du code qui définit les routines d'interruption est fixe quelle que soit l'application implémentée sur le processeur. Elle est composée des définition des fonctions suivantes :

```
void CAN_init_state(volatile CAN_ctxt_* ctxt);
void CAN_send_shared(volatile CAN_ctxt_* ctxt);
void CAN_send_waitingSynch_state(volatile CAN_ctxt_* ctxt);
void CAN_send_allSynchReceived(volatile CAN_ctxt_* ctxt);
void CAN_sending_state(volatile CAN_ctxt_* ctxt);
void CAN_recv_shared(volatile CAN_ctxt_* ctxt);
void CAN_recv_waitingSynch_state(volatile CAN_ctxt_* ctxt);
void CAN_recving_state(volatile CAN_ctxt_* ctxt);
void CAN_recv_brev2(uchar *, volatile unsigned);
void CAN_recv_brev4(uchar *, volatile unsigned);
void CAN_recv_brev8(uchar *, volatile unsigned);
void CAN_sync_shared(volatile CAN_ctxt_* ctxt);
void CAN_sync_state(volatile CAN_ctxt_* ctxt);
void CAN_resume_state(volatile CAN_ctxt_* ctxt);
```

En revanche la routine d'interruption de communication du bus CAN peut varier d'une application à une autre. Ci-dessous est le code d'une routine d'interruption de communication d'un bus CAN1 :

```
void __attribute__((interrupt, no_auto_psv)) _C1Interrupt(void)
{
if(C1INTFbits.TBIF) {
CAN_ctxt_1.stat(&CAN_ctxt_1);
C1INTFbits.TBIF = 0;
}
if(C1INTFbits.RBIF) {
if(C1RXFUL1bits.RXFUL1) {
CAN_ctxt_1.stat(&CAN_ctxt_1);
C1RXFUL1bits.RXFUL1 = 0;
}
C1INTFbits.RBIF = 0;
}
IFS2bits.C1IF = 0;
}
```

Annexe C

Programmation des dsPICs pour la tolérance aux fautes

C.1 Tolérance aux fautes sur le banc de test

Nous présentons dans cette section les programmes C pour la tolérance aux fautes des bus CAN et des dsPICs sur le banc de test.

C.1.1 Tolérance aux fautes des bus CAN

C.1.1.1 Détection d'erreurs

L'initialisation des variables "state_canx" est donnée par :

```
int state_can1 = 1;
int state_can2 = 1;
```

Les fonctions `initialize_timer1()` et `inhibit_timer1()` sont définies comme suit :

```
//Initialize Timer 1 for Period Interrupts
void initialize_timer1()
{
Count1 = 0;
T1CON = 0;            // Timer reset
IFS0bits.T1IF = 0;    // Reset Timer1 interrupt flag
IPC0bits.T1IP = 6;    // Timer1 Interrupt priority level=4
IEC0bits.T1IE = 1;    // Enable Timer1 interrupt
TMR1=  0x0000;
PR1 = 0xFFFF;         // Timer1 period register
```

```
T1CONbits.TON = 1;    // Enable Timer1 and start the counter
}

//inhibit timer interruption
void inhibit_timer1()
{
T1CON = 0;            // Timer reset
IFS0bits.T1IF = 0;    // Reset Timer1 interrupt flag
T1CONbits.TON = 0;    // Disable Timer1
Count1 = 0;
}
```

La routine d'interruption qui se charge de modifier les variables d'état est la suivante :

```
// ISR ROUTINE FOR THE TIMER1 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T1Interrupt( void )
{
    Count1++;
if(Count1 == 1000)
{
Count1 = 0;
state_can1 = 0;
LATBbits.LATB10 = 1;
   }
    TMR1 = 0;
// reset Timer 1 interrupt flag
    IFS0bits.T1IF = 0;
}

// ISR ROUTINE FOR THE TIMER2 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T2Interrupt( void )
{
    Count2++;
if(Count2 == 1000)
{
Count2 = 0;
state_can2 = 0;
LATBbits.LATB11 = 1;
   }
    TMR2 = 0;
```

```
// reset Timer 1 interrupt flag
    IFS0bits.T2IF = 0;
}
```

C.1.1.2 Traitement des erreurs détectées

La routines d'interruption d'erreurs a été modifiée pour pouvoir libérer les sémaphores en cas de fautes sur un bus CAN. Ci-dessous est le code C correspondant :

```
// ISR ROUTINE FOR THE TIMER1 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T1Interrupt( void )
{
    Count1++;
if(Count1 == 1000)
{
    Count1 = 0;
state_can1 = 0;
LATBbits.LATB10 = 1;
/* Release Pre0_CAN1 semaphore */
asm volatile("bset.b %0, #%1": "=T"(sem[0]): "i"(1));
//**allow thread to end
asm volatile("bset.b %0, #%1": "=T"(sem[0]): "i"(5));
// inhibit_timer1();
IFS0bits.T1IF = 0;      // Reset Timer1 interrupt flag
T1CONbits.TON = 0;      // Enable Timer1 and start the counter
T1CON = 0;              // Timer reset
   }
    TMR1 = 0;
// reset Timer 1 interrupt flag
    IFS0bits.T1IF = 0;
}

// ISR ROUTINE FOR THE TIMER2 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T2Interrupt( void )
{
    Count2++;
if(Count2 == 1000)
{
    Count2 = 0;
```

```
state_can2 = 0;
LATBbits.LATB11 = 1;
/* Release Pre0_CAN2 semaphore */
asm volatile("bset.b %0, #%1": "=T"(sem[0]): "i"(3));
//**allow thread to end
asm volatile("bset.b %0, #%1": "=T"(sem[0]): "i"(4));
// inhibit_timer2();
IFS0bits.T2IF = 0;  // Reset Timer2 interrupt flag
T2CONbits.TON = 0;  // Enable Timer2 and start the counter
T2CON = 0;          // Timer reset
   }
    TMR2 = 0;
// reset Timer 1 interrupt flag
    IFS0bits.T2IF = 0;
}
```

Enfin le code du sélecteur est le suivant :

```
/* 'select(_algo_input_i,_algo_input_j,_algo_select_o)' */
if(state_can1 == 1){
    int i;
    for(i=0;i<8;i++)
    _algo_select_o[i] = _algo_input_i[i];
    }
else if ((state_can1 == 0) &&(state_can2 == 1)){
    int i;
    for(i=0;i<8;i++)
    _algo_select_o[i] = _algo_input_j[i];
}
```

C.1.2 Tolérance aux fautes des dsPICs

C.1.2.1 Détection d'erreurs

L'initialisation des variables "state_px" sur le processeur "p1" est donnée par :

```
int state_p2 = 1;
int state_p3 = 1;
```

Il y a une routine d'interruption pour chaque processeur. Par exemple pour la détection d'erreur de "p2" la routine d'interruption est la suivante :

```
// ISR ROUTINE FOR THE TIMER1 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T1Interrupt( void )
{
    Count1++;
if(Count1 == 1000)
{
    Count1 = 0;
    state_p2 = 0;
   }
    TMR1 = 0;
    IFS0bits.T1IF = 0;  // reset Timer 1 interrupt flag
}
```

C.1.2.2 Traitement des erreurs détectées

Si "p1" est fautif alors "p2" envoi la donnée qui devrait être envoyée par "p1" via la ligne de fonction `thread_can1(4)`.

```
// ISR ROUTINE FOR THE TIMER1 INTERRUPT
void __attribute__((interrupt,no_auto_psv)) _T1Interrupt( void )
{
    Count1++;
if(Count1 == 1000)
{
    Count1 = 0;
state_p1 = 0;
LATBbits.LATB10 = 1;
// asm volatile("bset.b %0, #%1": "=T"(sem[0]): "i"(1));

// inhibit_timer1();
IFS0bits.T1IF = 0;      // Reset Timer1 interrupt flag
T1CONbits.TON = 0;      // Enable Timer1 and start the counter
T1CON = 0;              // Timer reset

CAN_ctxt_1.nsyn = 0;
thread_can1(4);   //** 4 renvoi immédiat //**
   }
    TMR1 = 0;
    IFS0bits.T1IF = 0;      // reset Timer 1 interrupt flag
}
```

C.2 Tolérance aux fautes sur le CyCab

Le conflit de communication entre les dsPICs et les MPC555s cité dans la section 8.7.1 a été résolu comme suit.

C.2.1 Communication entre dsPICs et MPC555

Il faut apporter les modifications suivantes :
Définition d'un type partagé longint code sur 4 octets dans d555.m4x :

```
############
# DATA TYPES

# --------
# typedef_(name, size) defines a new type with
# its size in address units
typedef_(‘bool’,  1) # MUST be defined
typedef_(‘char’,  1) # SHOULD be defined
typedef_(‘int’,   4) # SHOULD be defined
typedef_(‘float’, 4) # SHOULD be defined
typedef_(‘double’,8) # SHOULD be defined
typedef_(‘longint’,4)
```

Définition d'un type partagé longint code sur 4 octets dans rtai386.m4x :

```
############
# DATA TYPES

# --------
# typedef_(name, size) defines a new type with
# its size in address units
typedef_(‘bool’,  1) # MUST be defined
typedef_(‘char’,  1) # SHOULD be defined
typedef_(‘int’,   4) # SHOULD be defined
typedef_(‘float’, 4) # SHOULD be defined
typedef_(‘double’,8) # SHOULD be defined
typedef_(‘longint’,4)
```

Définition d'un type partagé longint code sur 4 octets dans PIC33.m4x :

```
############
# DATA TYPES
```

```
# --------
# typedef_(name, size) defines a new type with
# its size in address units
typedef_(‘prec_synchro’, 0) # MUST be defined
typedef_(‘bool’, 1) # MUST be defined
typedef_(‘char’, 1)
typedef_(‘int’,  2)
typedef_(‘fractional’, 2)
typedef_(‘long’, 4)
typedef_(‘longlong’,   8)
typedef_(‘longint’,   4)

...

###############
# MAIN sequence

...

typedef long longint; // définition du type C
typedef unsigned char bool;
typedef unsigned char uchar;
typedef unsigned int uint;
typedef long long longlong;
typedef union {struct {uint l; uint h;}w;unsigned long l;}ulong;‘’dnl
divert(-1)
```

Définition de fonctions de conversion d'un int vers un longint et vice versa dans le fichier app.m4x :

```
define(‘longint2int’,‘copy_($2,$1)’)
define(‘int2longint’,‘copy_($2,$1)’)
```

Au cours des premiers tests sur le CyCab, PCANExplorer présentait des conflits au niveau des valeurs d'identification : plusieurs processeurs possédaient le même identifiant(CANId), rendant l'analyse des trames compliquée. Ce problème a été résolu en attribuant aux processeurs un identifiant différent pour le bootloading et pour la communication. Des modifications ont été apportées aux fichiers suivants :

Dans PIC33CAN.m4x, ajouter

```
########################
# COMMUNICATION SEQUENCE

# -------
# NodeId_(procrName) returns node identifier from
# processor identifier (in $(A).m4m)
define(‘NodeId_’,$1‘_ID’)
# -------
# BootId_(procrName) returns Node Identifier used
# for download
define(‘BootId_’,‘0x’’eval(NodeId_($1) | 0x4000, 16, 4)’)
# -------
# CANSId_(mediaName) returns downloader reserved CAN ID
define(‘CANSId_’,‘eval($1_DWNLD_ID & 0x7ff)’)
# -------
# CANId_(procrName) returns ID formated for CAN bus
# configuration register
define(‘CANId_’,‘eval(NodeId_($1)+CANSId_(mediaName_),16,3)’)

# -------
# NodeBId_(procrName) returns node identifier from
# processor identifier (in $(A).m4m)
define(‘NodeBId_’,$1‘_BID’)
# -------
# CANBId_(procrName) returns ID formated for CAN bus
# configuration register
define(‘CANBId_’,‘eval(NodeBId_($1)+CANSId_(mediaName_),16,3)’)
```

Remplacer

```
‘#’define ASYNC_CAN_ID 0x’’CANId_(processorName_)
```

par

```
‘#’define ASYNC_CAN_ID 0x’’CANBId_(processorName_)
```

Remplacer

```
_(‘CAN_buf_’CPid_‘[0*8+0] = 0x’CANId_(processorName_)‘ << 2;’)dnl
```

par

```
_(‘CAN_buf_’CPid_‘[0*8+0] = (0x’eval(CANSId_(mediaName_), 16, 3)
‘ | 0x’eval(lindex(processorName_,shift($@)),16,4)‘) << 2;’)dnl
```

Dans PIC33.m4m, remplacer :

```
define(‘PROCID’,procr_‘_ID’)dnl defined in file $(A).m4m
```

par

```
define(‘PROCID’,procr_‘_BID’)dnl defined in file $(A).m4m
```

Dans app.m4m des dsPICs, remplacer :

```
define(‘p1_ID’, 0x03) # first allowed processor identifier!
define(‘p2_ID’, 0x04)
define(‘p3_ID’, 0x05)
```

par

```
define(‘p1_BID’, 0x03) # first allowed processor identifier!
define(‘p2_BID’, 0x04)
define(‘p3_BID’, 0x05)
```

Bibliographie

[ABB08] B. Andersson, K. Bletsas, and S. Baruah.
Scheduling arbitrary-deadline sporadic task systems on multiprocessors.
In *Real-Time Systems Symposium, 2008*, pages 385–394, 30 2008-dec. 3 2008.

[AL81] T. Anderson and P.A. Lee.
Fault tolerance, principles and practice.
Prentice/Hall International, 1981.

[And00] J. H. Anderson.
Pfair scheduling : Beyond periodic task systems.
In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[ASBH10] Ahmad Al Sheikh, Olivier Brun, and Pierre-Emmanuel Hladik.
Partition Scheduling on an IMA Platform with Strict Periodicity and Communication Delays.
In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 179–188, Toulouse, France, November 2010.
Rapport LAAS n˚10364 Rapport LAAS n˚10364.

[AT06] B. Andersson and E. Tovar.
Multiprocessor scheduling with few preemptions.
In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 322–334, 0-0 2006.

[Avi67] A. Avizienis.
Design of fault-tolerant computers.
Fall Joint Computer, pages 733–743, 1967.

[BA97] A. Burns and A.Wellings.
Real-time systems and programming languages.
Addison-Wesley, 1997.

[Bar98] S.K. Baruah.

A general model for recurring real-time tasks.
In *Proceedings of Real-Time Systems Symposium, 1998*, pages 114–122, dec 1998.

[Bar06] S. K. Baruah.
The non-preemptive scheduling of periodic tasks upon multiprocessors.
Real-Time Systems, 32(1-2) :9–20, 2006.

[BB05] Enrico Bini and Giorgio C. Buttazzo.
Measuring the performance of schedulability tests.
Real-Time Syst., 30 :129–154, May 2005.

[BC06] S.K. Baruah and S. Chakraborty.
Schedulability analysis of non-preemptive recurring real-time tasks.
Parallel and Distributed Processing Symposium, International, 0 :149, 2006.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel.
Proportionate progress : A notion of fairness in resource allocation.
Algorithmica, 15 :600–625, 1996.

[BCS69] W. G. Bouricius, W. C. Carter, and P. R. Schneider.
Reliability modeling techniques for self-repairing computer systems.
In *IEEE Transactions on Computers*, Yorktown Heights, New York, 1969. IBM Watson Research Center.

[BGP95] S.K. Baruah, J.E. Gehrke, and C.G. Plaxton.
Fast scheduling of periodic tasks on multiple resources.
In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 280 –288, apr 1995.

[BI07] Moris Behnam and Damir Isovic.
Real-time control design for flexible scheduling using jitter margin.
Technical Report ISSN 1404-3041 ISRN MDH-MRTC-214/2007-1-SE, Målardalen University, June 2007.

[Bim07] F. Bimbard.
Dimensionnement Temporel de Systèmes Embarqués : Application à OSEK.
PhD thesis, Université Pierre et Marie Curie, Paris 6, 2007.

[BLOS95] A. Burchard, J. Liebeherr, Yingfeng Oh, and S.H. Son.
New strategies for assigning real-time tasks to multiprocessor systems.
Computers, IEEE Transactions on, 44(12) :1429–1442, dec 1995.

[BR00] R. Balakrishnan and K. Ranganathan.
A Textbook of Graph Theory.

Springer, New York, 2000.

[BRC08] Patricia Balbastre, Ismael Ripoll, and Alfons Crespo.
Minimum deadline calculation for periodic real-time tasks in dynamic priority systems.
IEEE Trans. Comput., 57 :96–109, January 2008.

[BWM92] V. Bobin, S. Whitaker, and G. Maki.
Links between n-modular redundancy and the theory of error-correct ing codes.
In *The 1992 4th NASA SERC Symposium on VLSI Design*. 4th NASA Symposium on VLSI Design 1992 i, 1992.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri.
Ordonnancement temps réel - Cours et exercices corrigés.
Hermès, janvier 2000.

[CGG04] Annie Choquet-Geniet and Emmanuel Grolleau.
Minimal schedulability interval for real-time systems of periodic tasks with offsets.
Theor. Comput. Sci., 310(1-3) :117–134, January 2004.

[CJG+98] E.G. Coffman, Jr., G. Galambos, S. Martello, and Daniele Vigo.
Bin packing approximation algorithms : Combinatorial analysis, 1998.

[CKS02] L. Cucu, R. Kocik, and Y. Sorel.
Real-time scheduling for systems with precedence, periodicity and latency constraints.
In *Proceedings of 10th Real-Time Systems Conference, RTS'02*, Paris, France, March 2002.

[CP99] P. Chevochot and I. Puaut.
Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies.
In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 356 –363, 1999.

[CPS07] L. Cucu, N. Pernet, and Y. Sorel.
Periodic real-time scheduling : from deadline-based model to latency-based model.
Annals of Operations Research, 2007.

[CS03] L. Cucu and Y. Sorel.
Schedulability condition for systems with precedence and periodicity constraints without preemption.
In *Proceedings of 11th Real-Time Systems Conference, RTS'03*, Paris, March 2003.

[CSC05] K. Chaaban, M. Shawky, and P. Crubille.

A distributed framework for real-time in-vehicle applications.
In *Proceedings of Intelligent Transportation Systems, 2005*, pages 925 – 929, sept. 2005.

[Cuc04] L. Cucu.
Ordonnancement non préemptif et condition d'ordonnançabilité pour système embarqués à contraintes temps réel.
PhD thesis, Université de Paris Sud, Spécialité Électronique, 28/05/2004.

[DB05] R.I. Davis and A. Burns.
Hierarchical fixed priority pre-emptive scheduling.
In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp. –398, dec. 2005.

[DB10] Robert I. Davis and Alan Burns.
A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems, 2010.

[Der74] M. Dertouzos.
Control Robotics : the procedural control of physical processors.
Proceedings of the IFIP congress, pp. 807-813, 1974.

[DGLS01] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel.
Off-line real-time fault-tolerant scheduling.
In *Euromicro Workshop on Parallel and Distributed Processing*, pages 410–417, Mantova, Italy, February 2001.

[DK08] T.J. Dysart and P.M. Kogge.
System reliabilities when using triple modular redundancy in quantum-dot cellular automata.
In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on*, pages 72 –80, oct. 2008.

[DM89] M.L. Dertouzos and A.K. Mok.
Multiprocessor online scheduling of hard-real-time tasks.
Software Engineering, IEEE Transactions on, 15(12) :1497–1506, dec 1989.

[DP06] Klaus Danne and Marco Platzner.
An edf schedulability test for periodic tasks on reconfigurable hardware devices.
SIGPLAN Not., 41(7) :93–102, June 2006.

[EHN+10a] Friedrich Eisenbrand, Nicolai Hahnle, Martin Niemeier, Martin Skutella, José Verschae, and Andreas Wiese.
The periodic maintenance problem.
Technical report, EPF Lausanne and TU Berlin, February 2010.

[EHN+10b] Friedrich Eisenbrand, Nicolai Hahnle, Martin Niemeier, Martin Skutella, José Verschae, and Andreas Wiese.
Scheduling periodic tasks in a hard real-time environment.
In *37th International Colloquium on Automata, Languages and Programming (ICALP2010)*, volume 37, pages 299–311. Springer-Verlag, 2010.

[Gar05] A. B. Abril Garcia.
Estimation et optimisation de la consommation dans les description architecturales des systèmes integrés complexes.
PhD thesis, Université de Paris 6, 2005.

[GDF97] Joel Goossens, Raymond Devillers, and R. Devillers Fjgoosens.
The non-optimality of the monotonic priority assignments for hard real-time offset free systems, 1997.

[GGN06] Mathieu Grenier, Joël Goossens, and Nicolas Navet.
Near-Optimal Fixed Priority Preemptive Scheduling of Offset Free Systems.
In *14th International Conference on Real-Time and Networks Systems (RTNS'06)*, pages 35–42, Poitiers/France, May 2006.

[GJ90] Michael R. Garey and David S. Johnson.
Computers and Intractability ; A Guide to the Theory of NP-Completeness.
W. H. Freeman & Co., New York, NY, USA, 1990.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel.
Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors.
In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[GMP+90] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel, and M. Sorine.
SynDEx, un environnement de programmation pour multiprocesseur de traitement du signal, mécanismes de communication.
Rapport de Recherche 1236, INRIA, June 1990.

[Gra00] T. Grandpierre.
Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.
PhD thesis, Université de Paris Sud, Spécialité électronique, 30/11/2000.

[GRS96] L. George, N. Rivierre, and M. Spuri.
Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling.

Research Report RR-2966, INRIA, 1996.
Projet REFLECS.

[GS96] R. Guerraoui and A. Schiper.
Fault-tolerance by replication in distributed systems.
Proceeding Conference on Reliable Software Technologies, pages 38—-57, 1996.

[GS97] Rachid Guerraoui and André Schiper.
Software-based replication for fault tolerance.
Computer, 30 :68–74, April 1997.

[GS03] T. Grandpierre and Y. Sorel.
From algorithm and architecture specification to automatic generation of distributed real-time executives : a seamless flow of graphs transformations.
In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[HL92a] K.S. Hong and J.Y.-T. Leung.
On-line scheduling of real-time tasks.
IEEE Transactions on Computers, 41 :1326–1331, 1992.

[HL92b] Kwang S. Hong and Joseph Y.-T. Leung.
On-line scheduling of real-time tasks.
IEEE Trans. Comput., 41(10) :1326–1331, October 1992.

[Inc11] Microchip Technology Incorporated.
33F Reference Manual Part 1, 2011.

[IPEP05] V. Izosimov, P. Pop, P. Eles, and Z. Peng.
Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems.
In *Proceedings of Design, Automation and Test in Europe, 2005*, pages 864 – 869 Vol. 2, march 2005.

[Jac55] J. R. Jackson.
Scheduling a production line to minimize maximum tardiness.
Research Report Number 43, Management Science Research Project, UCLA, 1955.

[Jos85] Mathai Joseph.
On a problem in real-time computing.
Information Processing Letters, 20(4) :173 – 177, 1985.

[JP86] Mathai Joseph and Paritosh K. Pandya.
Finding response times in a real-time system.
Comput. J., 29(5) :390–395, 1986.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel.
On non-preemptive scheduling of period and sporadic tasks.
In *Proceedings of the 12 th IEEE Symposium on Real-Time Systems*, pages 129–139, December 1991.

[KAL96] Jan H. M. Korst, Emile H. L. Aarts, and Jan Karel Lenstra.
Scheduling periodic tasks.
INFORMS Journal on Computing, 8(4) :428–435, 1996.

[Kal04] H. Kalla.
Génération automatique de distributions/ordonnancements temps réel fiables et tolérant les fautes.
PhD thesis, Institut National Polytechnique de Grenoble, Spécialité Systèmes et Logiciel, 17/12/2004.

[KALW91] Jan H. M. Korst, Emile H. L. Aarts, Jan Karel Lenstra, and Jaap Wessels.
Periodic multiprocessor scheduling.
In *PARLE (1)*, pages 166–178, 1991.

[Ker09] O. Kermia.
Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence.
PhD thesis, Université de Paris Sud, Spécialité Physique, 19/03/2009.

[KK07] I. Koren and C. Mani Krishna.
Fault-Tolerant Systems.
Morgan Kaufmann, San Francisco, USA, 2007.

[Koc00] R. Kocik.
Sur l'optimisation des systèmes distribués temps réel embarqués : application au prototypage rapide d'un véhicule électrique semi-autonome.
PhD thesis, Université de Rouen, Spécialité informatique industrielle, 22/03/2000.

[KS07] O. Kermia and Y. Sorel.
A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor.
In *Proceedings of ISCA 20th International Conference on Parallel and Distributed Computing Systems, PDCS'07*, Las Vegas, Nevada, USA, September 2007.

[KS08] O. Kermia and Y. Sorel.
Schedulability analysis for non-preemptive tasks under strict periodicity constraints.

In *Proceedings of 14th International Conference on Real-Time Computing Systems and Applications, RTCSA'08*, Kaohsiung, Taiwan, August 2008.

[KY07] S. Kato and N. Yamasaki.
Real-time scheduling with task splitting on multiprocessors.
In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 441–450, aug. 2007.

[KY08a] Shinpei Kato and Nobuyuki Yamasaki.
Portioned edf-based scheduling on multiprocessors.
In *EMSOFT*, pages 139–148, 2008.

[KY08b] Shinpei Kato and Nobuyuki Yamasaki.
Portioned static-priority scheduling on multiprocessors.
In *IPDPS*, pages 1–12, 2008.

[Lap92] J. C. Laprie.
Dependability : Basic Concepts and Terminology : in English, French, German, Italian and Japanese.
Springer-Verlag, Wien, New York, USA, 1992.

[LBOS95] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son.
New strategies for assigning real-time tasks to multiprocessor systems.
IEEE Trans. Comput., 44(12) :1429–1442, December 1995.

[LDG01] J. M. Lopez, J. L. Diaz, and D. F. Garcia.
Minimum and maximum utilization bounds for multiprocessor rm scheduling.
In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, page 67, Washington, DC, USA, 2001. IEEE Computer Society.

[LGDG00] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia.
Worst-case utilization bound for edf scheduling on real-time multiprocessor systems.
ecrts, 00 :25, 2000.

[Lin02] B. Lincoln.
Jitter compensation in digital control systems.
In *American Control Conference, 2002. Proceedings of the 2002*, volume 4, pages 2985 – 2990 vol.4, 2002.

[Liu69] C. L. Liu.
Scheduling algorithms for multiprocessors in a hard real-time environment.

In *JPL Space Programs Summary 37-60*, volume 2, pages 28–37, November 1969.

[LL73] C. L. Liu and James W. Layland.
Scheduling algorithms for multiprogramming in a hard-real-time environment.
J. ACM, 20(1) :46–61, January 1973.

[LL08] Chien-Hung Lin and Ching-Jong Liao.
Makespan minimization for multiple uniform machines.
Computers amp ; Industrial Engineering, 54(4) :983 – 992, 2008.

[LM80] Joseph Y.-T. Leung and M. L. Merrill.
A note on preemptive scheduling of periodic, real-time tasks.
Inf. Process. Lett., 11(3) :115–118, 1980.

[LPAHP01] F. Lombardi, N. Park, M. Al-Hashimi, and H. H. Pu.
Modeling the dependability of n-modular redundancy on demand under malicious agreement.
In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, PRDC '01, pages 68–, Washington, DC, USA, 2001. IEEE Computer Society.

[LRL09] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky.
Partitioned fixed-priority preemptive scheduling for multi-core processors.
Real-Time Systems, Euromicro Conference on, 0 :239–248, 2009.

[LS97] C. Lavarenne and Y. Sorel.
Modèle unifié pour la conception conjointe logiciel-matériel.
Traitement du Signal, 14(6), 1997.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead.
On the complexity of fixed-priority scheduling of periodic, real-time tasks.
Performance Evaluation, 2(4) :237 – 250, 1982.

[mat] http ://www.mathworks.fr/products/matlab.

[MBFSV07] Leonardo Mangeruca, Massimo Baleani, Alberto Ferrari, and Alberto L. Sangiovanni-Vincentelli.
Uniprocessor scheduling under precedence constraints for embedded systems design.
ACM Trans. Embedded Comput. Syst., 7(1), 2007.

[MD78] A. K. Mok and M. L. Detouzos.
Multiprocessor scheduling in a hard real-time environment.
In *7th IEEE Texas Conf. on Computing Systems*, USA, 1978. IEEE.

[MFFR01] P. Marti, J.M. Fuertes, G. Fohler, and K. Ramamritham.

Jitter compensation for real-time control systems.
In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 39 – 48, dec. 2001.

[ML03] Donald C. Mayer and Ronald C. Lacoe.
Designing integrated circuits to withstand space radiation, 2003.

[Mok83a] A. K. Mok.
FUNDAMENTAL DESIGN PROBLEMS OF DISTRIBUTED SYSTEMS FOR THE HARD-REAL-TIME ENVIRONMENT.
PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1983.

[Mok83b] A.K. Mok.
Fundamental Design Problems for the Hard Real-Time Environments.
PhD thesis, MIT, 1983.

[MS10] M. Marouf and Y. Sorel.
Schedulability conditions for non-preemptive hard real-time tasks with strict period.
In *Proceedings of 18th International Conference on Real-Time and Network Systems, RTNS'10*, Toulouse, France, November 2010.

[OS95] Yingfeng Oh and Sang H. Son.
Allocating fixed-priority periodic tasks on multiprocessor systems.
Real-Time Syst., 9 :207–239, November 1995.

[Par02] F. Parain.
Ordonnancement sous contraintes énergétiques d'applications multimédia sur une plate-forme multiprocesseur hétérogène.
PhD thesis, Université de Rennes, 2002.

[pca] http ://www.peak-system.com.

[PIEP09] P. Pop, V. Izosimov, P. Eles, and Zebo Peng.
Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication.
Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 17(3) :389 –402, march 2009.

[Pra86] V. Pratt.
Modeling concurrency with partial orders.
Int. J. Parallel Program., 15(1), 1986.

[PSTW97] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein.
Optimal time-critical scheduling via resource augmentation (extended abstract).

In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 140–149, New York, NY, USA, 1997. ACM.

[QHPL00] X. Qin, Z. Han, L. Pang, and S. Li.
Real-time fault-tolerant scheduling in heterogeneous distributed systems.
Proceedings of Cluster Computing Technologies, Environments, and Applications(CC-TEA), June 2000.

[rob] http ://www.robosoft.com.

[Sor94] Y. Sorel.
Massively parallel systems with real time constraints, the algorithm architecture adequation methodology.
In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, May 1994.

[Sor96] Y. Sorel.
Real-time embedded image processing applications using the algorithm architecture adequation methodology.
In *Proceedings of IEEE International Conference on Image Processing, ICIP'96*, Lausanne, Switzerland, September 1996.

[SS83] Richard D. Schlichting and Fred B. Schneider.
Fail-stop processors : an approach to designing fault-tolerant computing systems.
ACM Trans. Comput. Syst., 1(3) :222–238, August 1983.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky.
Aperiodic task scheduling for hard-real-time systems.
Real-Time Systems, 1(1) :27–60, 1989.

[SVC98] S. Saez, J. Vila, and A. Crespo.
Using exact feasibility tests for allocating real-time tasks in multiprocessor systems.
In *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 53–60, jun 1998.

[SY91] T.J. Smith and J.N. Yelverton.
Processor architectures for fault tolerant avionic systems.
In *Proceedings of Digital Avionics Systems Conference, 1991*, pages 213 –219, oct 1991.

[Ven06] N. Ventroux.
Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués : élaboration et validation d'une architecture.
PhD thesis, Université de Rennes, 2006.

[Vic99] A. Vicard.

Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués.
PhD thesis, Université de Paris Nord, Spécialité informatique, 5/07/1999.

[WS99] Yun Wang and Manas Saksena.
Scheduling fixed-priority tasks with preemption threshold.
Real-Time Computing Systems and Applications, International Workshop on, 0 :328, 1999.

[xZbX09] Jian xiao Zou and Hong bing Xu.
Design and reliability analysis of emergency trip system with triple modular redundancy.
In *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, pages 1006 –1009, july 2009.

[YGSdR09] P. Meumeu Yomsi, L. George, Y. Sorel, and D. de Rauglaudre.
Improving the quality of control of periodic tasks scheduled by fp with an asynchronous approach.
International Journal on Advances in Systems and Measurements, 2(2), 2009.

[Yom09] P. Meumeu Yomsi.
Prise en compte du coût exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples.
PhD thesis, Université de Paris Sud, Spécialité Physique, 02/04/2009.

[ZGD11] Haitao Zhu, S. Goddard, and M.B. Dwyer.
Response time analysis of hierarchical scheduling : The synchronized deferrable servers approach.
In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 239 –248, 29 2011-dec. 2 2011.

Ordonnancement temps réel dur multiprocesseur et tolérant aux fautes appliqué à la robotique mobile

Résumé :

Nous nous sommes intéressés dans cette thèse au problème d'ordonnancement temps réel dur multiprocesseur tolérant aux fautes pour des tâches non préemptives périodiques strictes pouvant être combinées avec des tâches préemptives. Nous avons proposé des solutions à ce problème et les avons implantées dans le logiciel SynDEx puis nous les avons testées sur une application de suivi de véhicules électriques CyCabs.

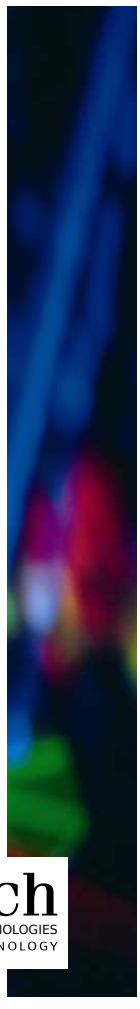
Nous avons d'abord présenté un état de l'art sur les systèmes temps réel embarqués et plus précisément sur l'ordonnancement classique monoprocesseur et multiprocesseur de tâches préemptives périodiques. Comme nous nous intéressons aux applications de contrôle/-commande temps réel critiques, les traitements de capteurs/actionneurs et les traitements de commande de procédés ne doivent pas avoir de gigue. Pour ces raisons nous avons aussi présenté un état de l'art sur l'ordonnancement des tâches non-préemptives périodiques strictes. Par ailleurs nous avons présenté un état de l'art sur la tolérance aux fautes. Comme nous nous sommes intéressés aux fautes matérielles, nous avons présenté les deux types de redondances : logicielle et matérielle.

Les analyses d'ordonnançabilité existantes de tâches non préemptives périodiques strictes dans le cas monoprocesseur ayant de faibles taux de succès d'ordonnancement, nous avons proposé une nouvelle analyse d'ordonnançabilité. Nous avons présenté une stratégie d'ordonnancement qui consiste à ordonnancer une tâche candidate avec un ensemble de tâches déjà ordonnancée. Nous avons utilisé cette stratégie pour ordonnancer des tâches harmoniques et non harmoniques, et nous avons proposé des nouvelles conditions d'ordonnançabilité. Afin d'améliorer le taux de succès d'ordonnancement de tâches non préemptives périodiques strictes, nous avons proposé de garder certaines tâches non préemptives périodiques strictes et d'y ajouter des tâches préemptives périodiques non strictes ne traitant ni les entrées/sorties ni le contrôle/commande.

Nous avons ensuite étudié le problème d'ordonnancement multiprocesseur selon une approche partitionnée. Ce problème est résolu en utilisant trois algorithmes. Le premier algorithme effectue une analyse d'ordonnançabilité monoprocesseur et assigne chaque tâche sur éventuellement plusieurs processeurs. Le deuxième algorithme transforme le graphe de tâches dépendantes en un graphe déroulé où chaque tâche est répétée un nombre de fois égal au rapport entre le PPCM des autres périodes et sa période. Le troisième algorithme exploite les résultats des deux algorithmes précédents pour choisir sur quel processeur ordonnancer une tâche et calculer sa date de début d'exécution.

Nous avons ensuite proposé d'étendre l'étude d'ordonnançabilité temps réel multiprocesseur précédente pour qu'elle soit tolérante aux fautes de processeurs et de bus de communication. Nous avons proposé un algorithme qui permet de transformer le graphe de tâches dépendantes en y ajoutant des tâches et des dépendances de données répliques et des tâches de sélection permettant de choisir la réplique de tâches allouée à un processeur non fautif. Nous avons étudié séparément les problèmes de tolérance aux fautes pour des processeurs, des bus de communication, et enfin des processeur et des bus de communication. Finalement nous avons étendu les trois algorithmes vus précédemment d'analyse d'ordonnançabilité, de déroulement et d'ordonnancement afin qu'ils soient tolérants aux fautes.

Nous avons ensuite présenté les améliorations apportées au logiciel SynDEx tant sur le plan de l'analyse d'ordonnançabilité et l'algorithme d'ordonnancement, que sur le plan de la tolérance aux fautes. Finalement nous avons présenté les travaux expérimentaux concernant l'application de suivi de CyCabs. Nous avons modifié l'architecture des CyCabs en y intégrant des microcontrôleurs dsPICs et nous avons testé la tolérance aux fautes de dsPICs et du bus CAN sur une application de suivi de CyCab.



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

Mots clés : systèmes temps réel, analyse d'ordonnançabilité, ordonnancement non préemptif, ordonnancement mixte préemptif/non préemptif, période stricte, ordonnancement monoprocesseur/multiprocesseur, tolérance aux fautes, redondance logicielle, application de robotique mobile

Fault-tolerant multiprocessor hard real-time scheduling for mobile robotics

Abstract:

In this thesis, we studied the fault-tolerant multiprocessor hard real-time scheduling of non-preemptive strict periodic tasks which could be combined with preemptive tasks. We proposed solutions that we implemented into the SynDEx software, then we tested these solutions on an electric vehicle following.

First, we present a state of the art on real-time embedded systems and more specificaly on the classical uniprocesseur and multiprocessor scheduling of preemptive periodic tasks. Since we were interested in critical real-time control applications, sensor/actuators computations and processes control must not have jitter. For these reasons, we also presented a state of the art of the scheduling of non-preemptive strict periodic tasks. Also, we presented a state of the art on fault-tolerance. As we were interested in hardware faults, we presented two types of redundancies: software and hardware.

Presently, existing schedulability analyses of non-preemptive strict periodic tasks have low schedulability success ratios, thus we proposed a new schedulability analysis. We first presented a scheduling strategy which consists in scheduling a candidate task whereas a task set is already scheduled. We used this strategy to solve the problem of scheduling harmonic and non-harmonic tasks, and we proposed new schedulability conditions. In order to improve the scheduling success ratio of non-preemptive strict periodic tasks, we proposed to keep some non preemptive strict periodic tasks and to add preemptive periodic tasks which are neither dedicated to input/output nor to control.

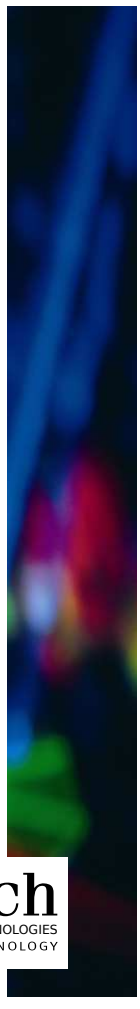
Then, we studied the multiprocessor scheduling problem using the partitioned approach. In order to solve this problem we proposed three algorithms. The first algorithm performs a uniprocessor schedulability analysis and assigns each task according to a schedulability condition to possibly several processors. The second algorithm transforms the dependent task graph into an unrolled graph where each task is repeated a number of times equal to the ratio between the LCM of all tasks periods and its period. The third algorithm exploits the two precedent algorithms to choose, with a cost function, on which processor it will schedule a task previously assigned to several processors, and it computes the first start times of each task.

Then, we extended the multiprocessor schedulability analysis to be tolerant to processor and bus media faults. We proposed an algorithm which transforms the dependent task graph by adding redundant tasks, redundant dependencies, and selecting tasks. The latter allow to choose the redundant task allocated to non faulty processors. We studied separately the processor fault-tolerance problem, the bus fault-tolerant problem, and finally both processor and bus fault-tolerant problem. Finally, we extended the schedulability analysis algorithms, the unrolling algorithm and the scheduling algorithm to be fault-tolerant.

Then, we presented the improvements provided to the SynDEx software for the schedulability analysis algorithm, the scheduling algorithm and the fault-tolerance algorithm.

Finally, we conducted some experiments on the electric vehicle following called CyCab. We modified the hardware architecture of the CyCab to integrate dsPICs microcontrolers, and we tested dsPICs and CAN buses fault-tolerant on the CyCabs following.

Keywords: real-time systems, schedulability analysis, non-preemptive scheduling, mixed preemptive/non-preemptive scheduling, strict period, uniprocessor/multiprocesseor scheduling, fault-tolerance, software redundancy, mobile robotics application



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY