Une heuristique d'ordonnancement et de distribution tolérante aux pannes pour systèmes temps-réel embarqués

Alain Girault* — **Hamoudi Kalla*** — **Yves Sorel****

* *Projet POP ART (INRIA Rhône-Alpes)*
ZIRST – 655, avenue de l'Europe Montbonnot
38334 Saint-Ismier cedex, FRANCE
Alaint.Girault,Hamoudi.Kalla@inrialpes.fr
Tel : 04 76 61 53 51 - Fax : 04 76 61 52 52
** *Projet OSTRE (INRIA Rocquencourt)*
B.P.105 - 78153 Le Chesnay Cedex, FRANCE

RÉSUMÉ. *Nous présentons dans cet article une nouvelle heuristique d'ordonnancement de tâches sur des architectures multiprocesseurs distribuées, qui permet de générer un ordonnancement statique, distribué, tolérant aux pannes des processeurs et minimisant la longueur totale de l'ordonnancement généré. L'heuristique que nous proposons est basée sur la réplication active des tâches, où chaque tâche est répliquée sur au moins $\mathcal{N}pf$ + 1 processeurs afin de tolérer $\mathcal{N}pf$ pannes. À travers un exemple détaillé, nous illustrons les techniques utilisées pour minimiser la longueur de l'ordonnancement et tolérer les pannes des processeurs. Grâce à des simulations, nous montrons l'efficacité de notre méthode par rapport à d'autres heuristiques trouvées dans la littérature.*

ABSTRACT. *In this paper, we present a new task scheduling heuristic on distributed multiprocessor architectures, which allows to generate a static distributed fault tolerant schedule, minimising the length of the whole generated schedule. The heuristic that we propose is based on the active replication of the tasks, where each task is replicated at least on $\mathcal{N}pf$ + 1 different processors to tolerate $\mathcal{N}pf$ failures. Through a detailed example, we show the techniques used to minimise the length of the schedule and to tolerate the failures of processors. A study by simulations shows the efficiency of our method compared to other heuristics found in the literature.*

MOTS-CLÉS : *Systèmes critiques, systèmes temps-réel embarqués, architectures distribuées, systèmes tolérants aux pannes, heuristique d'ordonnancement statique.*

KEYWORDS: *Safety critical systems, embedded real-time systems, distributed architectures, fault-tolerant systems, static scheduling heuristics.*

1. Introduction

Les systèmes distribués embarqués sont de plus en plus utilisés dans plusieurs domaines applicatifs importants comme les véhicules (automobiles, robots, avions, satellites, bateaux), et les systèmes de contrôle de processus industriels. Une des caractéristiques importantes de ces systèmes est d'être réactif. Un système réactif est un système qui réagit continûment avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs, des entrées provenant de l'environnement, appelés stimuli, réagit à tous ces stimuli en effectuant un certain nombre d'opérations et produit, grâce à des actionneurs, des sorties utilisables par l'environnement, appelés réactions. D'une façon générale, ce sont des systèmes périodiques échantillonnés, dont le mode d'exécution est le suivant :

```
à chaque top faire
  lire les entrées
  calculer
  écrire les sorties
fin faire
```

Ici, `top` est une horloge dont le rythme doit être fixé en fonction des caractéristiques de l'environnement, c'est-à-dire du rythme d'évolution des grandeurs physiques. Pour valider ce modèle, il faut borner le temps de calcul afin de vérifier qu'il est plus petit que la période de l'horloge.

Ces systèmes réalisent des tâches complexes critiques [RUS 94] et sont soumis à des fortes contraintes en terme de temps et de fiabilité. En effet, au vu des conséquences catastrophique (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une panne, ces systèmes doivent être extrêmement fiables. Pour cette raison, ils doivent mettre en œuvre des mécanismes de tolérance aux pannes comme la redondance matérielle/logicielle [GUE 96, HIL 98, CHE 99, TOR 00].

La majorité des méthodes existant dans la littérature pour la conception des systèmes tolérants aux pannes se concentrent sur les problèmes qui résultent des défaillances de matériel (processeurs et liens de communications). Ils supposent que le programme (l'ensemble des tâches) est correct et validé, que ce soit par des outils de « model-checking » ou des assistants de preuve. Ces méthodes sont basées sur la réplication passive ou active des tâches sur des processeurs distincts afin de tolérer les pannes. La réplication active [CHE 99] consiste à exécuter la même tâche en parallèle sur n processeurs distincts. La réplication passive [OH 97, AHN 97, QIN 00] consiste à répliquer chaque tâche en n exemplaires, mais une seule des n répliques effectue le calcul. Les $n-1$ autres répliques sont passives et ne prennent la relève que si la réplique active est défaillante.

Le domaine de nos recherches est centré sur les systèmes distribués embarqués tolérants aux pannes. L'objectif général est de générer automatiquement un exécutif distribué (appelés aussi système d'exploitation), tolérant aux pannes des processeurs, à partir d'une spécification d'architecture et d'algorithme (figure 1).

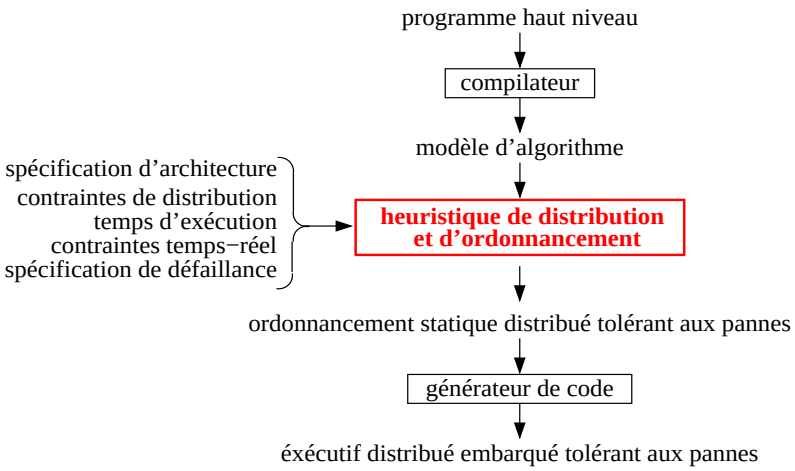

Figure 1. *Notre méthodologie de génération d'exécutif tolérant aux pannes.*

Dans cet article, nous concentrons notre étude sur le sous-problème de génération automatique d'un ordonnancement statique tolérant aux pannes de processeurs, à partir d'une spécification d'architecture ($\mathcal{A}rc$), d'une spécification d'algorithme ($\mathcal{A}lg$), des contraintes de distributions ($\mathcal{D}is$), des temps d'exécutions ($\mathcal{E}xe$) de $\mathcal{A}lg$ sur $\mathcal{A}rc$, des contraintes temps-réel ($\mathcal{R}tc$) à satisfaire et un nombre ($\mathcal{N}pf$) de processeurs à tolérer.

Le problème de placement/ordonnancement dans un système temps réel à contraintes strictes est connu pour être un problème NP-complet [GAR 79]. C'est pourquoi, nous proposons une heuristique de placement/ordonnancement qui cherche une solution approchée, et qui vérifie les contraintes temporelles, de distributions et de défaillances. Notre approche de tolérance aux pannes est basée sur une heuristique *statique* et une technique de *réplication logicielle* pour deux principales raisons :

– *Un ordonnancement statique* (hors-ligne) [AHM 95] signifie que la séquence de l'ordonnancement est prédéterminée à l'avance, ce qui nous permet d'analyser hors-ligne le système en l'absence et en présence de pannes, *avant* de le déployer.

– *Les systèmes embarqués* doivent respecter des contraintes physiques et financières fortes. La redondance logicielle permet de tolérer les pannes des processeurs tout en satisfaisant ces contraintes.

La méthode que nous proposons est basée sur la *redondance active* des tâches, où au moins $\mathcal{N}pf+1$ copies de chaque tâche sont exécutées en parallèle dans des processeurs distincts, afin de tolérer $\mathcal{N}pf$ pannes de processeurs.

Le reste de cet article est organisé comme suit : la section 2 présente le modèle d'architecture, d'algorithme et de défaillance que nous utilisons dans notre approche.

La section 3 présente l'algorithme de placement/ordonnancement tolérant aux pannes que nous proposons. Un exemple illustratif est présenté dans la section 4. Enfin, la section 5 présente les résultats d'un ensemble de simulations. Nous concluons dans la section 6.

2. Modèles du système

2.1. *Modèle d'algorithme*

L'algorithme ($\mathcal{A}lg$) est modélisé par un graphe flot de donnée, où les nœuds sont les opérations de calcul de l'algorithme et les arcs sont les dépendances de données entre opérations. Une opération ne peut s'exécuter que lorsque ses données d'entrée sont présentes. Elle consomme ses données d'entrée et produit des données en sortie, qui sont ensuite utilisées par ses successeurs. Une opération sans prédécesseur (resp. sans successeur) représente une interface d'entrée, c'est-à-dire un capteur, (resp. une interface de sortie, c'est-à-dire un actionneur) avec l'environnement. Les opérations d'entrée/sortie sont appelées `extios`; toutes les autres opérations de calcul sont appelées `comps`; enfin les opérations de communication sont appelées `comms`. Ce graphe spécifie les trois phases `"lire les entrées","calculer" et "écrire les sorties"` de la boucle périodique d'exécution (voir section 1). Il est infiniment itéré pour prendre en compte les aspects réactifs temps-réel. La figure 2.1(a) est un exemple de graphe algorithme, avec neuf opérations : I et O sont des `extios`, A–G sont des `comps`.

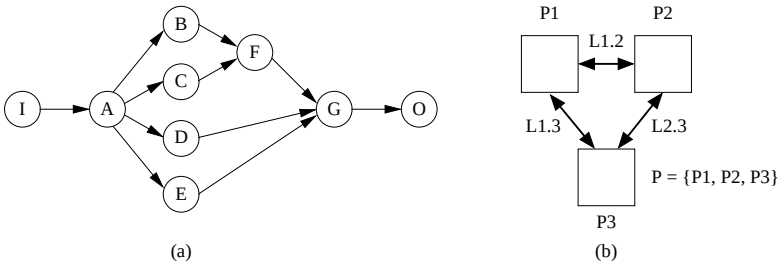

Figure 2. *(a) Graphe algorithme $\mathcal{A}lg$; (b) Graphe architecture $\mathcal{A}rc$.*

2.2. *Modèle d'architecture*

L'architecture ($\mathcal{A}rc$) est modélisée par un graphe non orienté, où les nœuds désignent les processeurs composant l'architecture et les arêtes désignent les liaisons physiques entre processeurs. Chaque processeur est composé d'une unité de calcul, d'une mémoire locale, et d'une ou plusieurs unités de communication pour communi-

quer avec les périphériques ou d'autres processeurs. La figure 2.1(b) est un exemple de graphe architecture, composé de trois processeurs et de trois liens de communication.

2.3. *Modèle de défaillance*

Dans cet article, nous ne considérons que les pannes permanentes et intermittentes des processeurs. Nous supposons que chaque processeur est de type silence sur défaillance, c'est-à-dire qu'en cas de panne il ne produit strictement rien en sortie. Nous supposons aussi que les actionneurs et les capteurs sont fiables, et que le réseau de communication physique n'est jamais partitionné.

2.4. *Contraintes de distribution, temps d'exécution et contraintes temps-réel*

Les temps d'exécutions $\mathcal{E}xe$ de chaque `comp` et `extio` de $\mathcal{A}lg$ sont représentés par un tableau, où chaque case $\langle o, p\rangle$ représente le temps d'exécution de l'opération o sur le processeur p. Comme l'architecture est hétérogène, le temps d'exécution de chaque opération peut varier d'un processeur à un autre.

Les temps d'exécutions $\mathcal{E}xe$ de chaque `comm` de $\mathcal{A}lg$ sont représentés par un tableau, où chaque case $\langle d, l\rangle$ représente le temps de communication de la dépendance de données d à travers le lien de communication l.

Les contraintes de distribution $\mathcal{D}is$ sont données dans le tableau $\mathcal{E}xe$ par association de la valeur "∞" à certaines cases $\langle o, p\rangle$, qui signifie que l'opération o ne peut pas être implantée sur le processeur p.

Par exemple, les tableaux 1 et 2 donnent les contraintes $\mathcal{D}is$ et $\mathcal{E}xe$ pour les graphes $\mathcal{A}lg$ et $\mathcal{A}rc$ de la figure 2.1.

		opération								
	temps d'exécution	I	A	B	C	D	E	F	G	O
proc.	P1	1	2	3	2	3	1	2	1.4	1.4
	P2	1.3	1.5	1	3	1.7	1.2	2.5	1	∞
	P3	∞	1	1.5	1	3	2	1	1.5	1.8

Tableau 1. *$\mathcal{D}is$ et $\mathcal{E}xe$ pour les opérations de calcul.*

Enfin, les contraintes temps réel $\mathcal{R}tc$ peuvent être de différents types, par exemple l'échéance de $\mathcal{A}lg$, c'est-à-dire la date à laquelle l'exécution de $\mathcal{A}lg$ doit être terminée. Pour notre exemple, $\mathcal{R}tc = 16$, ce qui signifie que la longueur de l'ordonnancement tolérant aux pannes obtenu doit être inférieure à 16 unités de temps.

		dépendance de donnée					
	temps d'exécution	I ▷ A	A ▷ B	A ▷ C	A ▷ D	A ▷ E	B ▷ F
lien	L1.2	1.75	1	1	1.5	1	1
	L2.3	1.25	0.5	0.5	1	0.5	0.5
	L1.3	1.25	0.5	0.5	1	0.5	0.5

		dépendance de donnée				
	temps d'exécution	C ▷ F	D ▷ G	E ▷ G	F ▷ G	G ▷ O
lien	L1.2	1.3	1.9	1.3	1	1.1
	L2.3	0.8	1.4	0.8	0.5	0.6
	L1.3	0.8	1.4	0.8	0.5	0.6

Tableau 2. *$\mathcal{E}xe$ pour les opérations de communication.*

3. La méthode proposée

Dans cette section, nous présentons dans un premier temps les bases techniques de notre méthode, puis dans un deuxième temps, nous présentons notre algorithme de placement/ordonnancement tolérant aux pannes. L'algorithme que nous proposons est un algorithme d'ordonnancement de liste basé sur la technique de la *réplication active* d'opérations [CHE 99], où chaque opération est répliquée en au moins $\mathcal{N}pf+1$ exemplaires sur des processeurs distincts, qui sont exécutées en parallèle afin de tolérer $\mathcal{N}pf$ pannes de processeurs.

3.1. *Principe de la méthode*

La méthode que nous proposons utilise la redondance logicielle des `comps`/`extios` et `comms`. Chaque opération `comp` ou `extio` de $\mathcal{A}lg$ est répliquée sur $\mathcal{R}ep$ processeurs distincts, où $\mathcal{R}ep$ doit être supérieur à $\mathcal{N}pf+1$. Chaque réplique d'une opération envoie ses résultats en parallèle à tous les répliques de toutes ses opérations successeurs dans $\mathcal{A}lg$. Ainsi, chaque opérations reçoit ses données d'entrée en $\mathcal{R}ep$ exemplaires ; dès qu'elle reçoit le premier exemplaire de ses données d'entrées, l'opération commence son exécution et ignore les autres exemplaires. Cependant, dans certains cas, une opération réplique peut recevoir ses données d'entrées une seule fois ; ce qui est le cas dans une communication intra-processeurs. Par exemple, supposons qu'une opération X ait une seule entrée, et que Y soit son prédécesseur (figure 3(a)).

Dans l'exemple de la figure 3(a), supposons qu'une réplique de X est assignée au processeur P. Deux cas distincts peuvent se présenter : ou bien une réplique de Y est assignée à P, ou bien toutes les répliques sont assignées à des processeurs différents de P. Dans le premier cas, la communication `comm` entre Y et X n'est pas répliquée et elle est implantée comme une opération de communication *intra-processeur* (figure 3(b)). Dans le deuxième cas, la communication `comm` entre X et Y est répliquée en $\mathcal{N}pf+1$ exemplaires, chaque `comm` est implantée comme une opération de communication

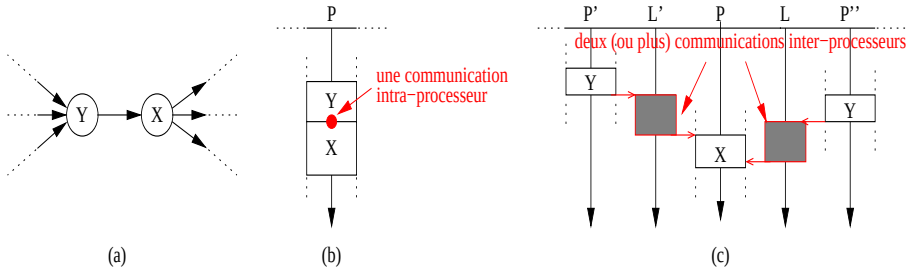

Figure 3. *(a) Sous-graphe algorithme ; (b) Ordonnancement avec une réplique de Y sur P ; (c) Ordonnancement sans réplique de Y sur P. Dans ces deux ordonnancements, les opérations sont représentées par des boîtes dont la hauteur est proportionnelle à leur durée d'exécution ou de communication.*

inter-processeur (figure 3(c)). Les données provenant de P' arrivent en premier, donc X s'exécute et ignore les données provenant de P".

Dans la mesure où le coût d'une communication intra-processeur est négligeable, la réplication en plus de $\mathcal{N}pf+1$ exemplaires de certaines opérations de calcul peut réduire le temps global de communication, en remplaçant des communications inter-processeurs par des communications intra-processeur. Par exemple, dans la figure 4(a) : si Y est répliqué en plus sur P, la longueur de l'ordonnancement est réduit, à la fois en l'absence de pannes et en présence de pannes, comme cela est illustré sur la figure 4(b).

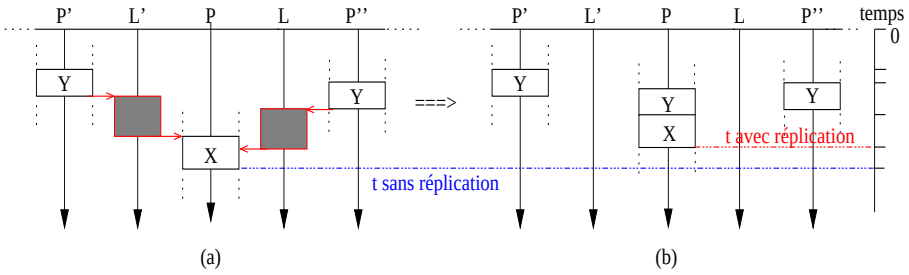

Figure 4. *Ordonnancement de plus de $\mathcal{N}pf+1$ répliques de l'opération Y.*

3.2. *Heuristique d'ordonnancement*

L'heuristique implantant cette méthode est une heuristique glouton de type ordonnancement de liste [YAN 93], appelée HTBR (« Heuristique Tolérante aux pannes

Basée sur la Réplication active »). Le principe général de l'heuristique est donné dans la figure 5. L'exposant entre parenthèses désigne l'étape de l'heuristique.

Avant de présenter l'heuristique, nous définissons les notations suivantes qui sont employées dans le reste de cet article :

- $O_{cand}^{(n)}$: La liste des opérations *candidates*. Une opération devient candidate si tous ses prédécesseurs sont déjà implantés.
- $O_{impl}^{(n)}$: La liste des opérations déjà *implantées*.
- $R^{(n)}$: La longueur de l'ordonnancement à l'étape n de l'heuristique.
- $pred(o_i)$: Les prédécesseurs de l'opération o_i.
- $succ(o_i)$: Les successeurs de l'opération o_i.
- $E_{exc}^{(n)}(o_i, p_j)$: La date de fin d'exécution de o_i implantée sur p_j.
- $E_{com}^{(n)}(o_i, o_j)$: La date de fin de communication entre o_i et o_j.
- $\overline{\Delta}(o_i)$: Le temps moyen d'exécution de o_i sur les processeurs.
- $\overline{S}^{(n)}(o_i)$: La date de début au-plus-tard de o_i depuis la fin. Elle est calculée comme suit :
$$\overline{S}^{(n)}(o_i) = \overline{\Delta}(o_i) + \max_{o_j \in succ(o_i)} \overline{S}^{(n)}(o_j)$$
- $S_{best}^{(n)}(o_i, p_l)$: La date de début au-plus-tôt de l'opération o_i sur le processeur p_l. Elle est calculée comme suit :
$$S_{best}^{(n)}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \min_{k=1}^{Npf+1} E_{com}^{(n)}(o_j^k, o_i) \right\}$$
- $S_{worst}^{(n)}(o_i, p_l)$: La date de début au-plus-tard de l'opération o_i sur le processeur p_l, prenant en compte tous les répliques de ses prédécesseurs. Elle est calculée comme suit :
$$S_{worst}^{(n)}(o_i, p_l) = \max_{o_j \in pred(o_i)} \left\{ \max_{k=1}^{Npf+1} E_{com}^{(n)}(o_j^k, o_i) \right\}$$
où o_j^k est la k^{eme} réplique de o_j. Si o_i et o_j sont implantées sur le même processeur p_l alors : $E_{com}^{(n)}(o_j^k, o_i) = E_{exc}^{(n)}(o_j, p_l)$.

À chaque étape de l'heuristique, une liste d'opérations implantables auxquelles nous avons assigné des priorités est établie. Cette liste est ordonnée par priorité décroissante. La règle de priorité choisie repose sur une fonction de coût, la pression d'ordonnancement [GRA 99]. La pression d'ordonnancement, notée par $\sigma(o_i, p_j)$ est calculée pour chaque opération $o_i \in O_{cand}^{(n)}$ et chaque processeur p_j par la formule suivante :
$$\sigma(o_i, p_j) = S_{worst}^{(n)}(o_i, p_j) + \overline{S}^{(n)}(o_i) - R^{(n-1)}$$

La pression d'ordonnancement vise à minimiser la longueur du chemin critique et à exploiter la marge d'ordonnancement de chaque opération.

Initialement, l'ensemble des opérations candidates à l'étape S0 de l'heuristique est l'ensemble des opérations de calcul sans prédécesseur (les `extios` d'entrées).

Dans la macro-étape mSn.1 de la figure 5, nous calculons pour chaque opération o_i autant de pressions d'ordonnancement qu'il y a de processeurs sur lesquels cette opération peut s'exécuter. Nous conservons pour chaque opération o_i uniquement les $\mathcal{N}pf + 1$ pressions d'ordonnancement $\sigma_{best}^{\mathcal{N}pf+1}(o_i)$ les plus petites, qui constituent les priorités de l'opération, et nous conservons également les $\mathcal{N}pf + 1$ processeurs $\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_i)$ correspondants. Puis, dans la macro-étape mSn.2, nous selectionnons parmi toutes les opérations candidates $O_{cand}^{(n)}$ la meilleure opération candidate o_{best} qui a la pression d'ordonnancement $\sigma_{urgent}(o_{best})$ maximale.

Algorithme HTBR :
BEBUT
S0. Initialiser la liste des candidats et la liste des opérations implantées :

$O_{cand}^{(0)} := \{o \in O \mid pred(o) = \emptyset\}$;

$O_{impl}^{(0)} := \emptyset$;

Sn. TANT QUE ($O_{cand}^{(n)} \neq \emptyset$) FAIRE

mSn.1 Calculer la pression d'ordonnancement pour chaque opération o_i de $O_{cand}^{(n)}$ sur chaque processeur p_j en utilisant $S_{worst}^{(n)}(o_i, p_j)$ et conserver pour chaque opération les $\mathcal{N}pf$+1 résultats les plus petits dans $\sigma_{best}^{\mathcal{N}pf+1}(o_i)$ et leurs processeurs associés dans $\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_i)$:

$$\forall o_i \in O_{cand}^{(n)}, \quad \sigma_{best}^{\mathcal{N}pf+1}(o_i) := \min_{p_j \in P}^{\mathcal{N}pf+1} \sigma(o_i, p_j)$$
$$\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_i) := \{p_j \mid p_j \in \sigma_{best}^{\mathcal{N}pf+1}(o_i)\}$$

mSn.2 Sélectionner la meilleure opération candidate o_{best}, telle que :

$$\sigma_{urgent}(o_{best}) = \max_{o_i \in O_{cand}^{(n)}} \sigma_{best}^{\mathcal{N}pf+1}(o_i)$$

mSn.3 Implanter l'opération o_{best} sur ses $\mathcal{N}pf$+1 processeurs de $\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_{best})$ conservés par l'étape mSn.1 en utilisant la fonction : « *Minimiser_la_date_de_début* » de la figure 6 ;

mSn.4 Mettre à jour la liste des opérations candidates et implantées :

$O_{cand}^{(n+1)} := O_{cand}^{(n)} - \{o_{best}\} \cup \{o' \in succ(o_{best}) \mid pred(o') \subseteq O_{impl}^{(n)}\}$;

$O_{impl}^{(n)} := O_{impl}^{(n-1)} \cup \{o_{best}\}$;

FIN TANT QUE
FIN

Figure 5. *L'heuristique HTBR de distribution et d'ordonnancement tolérante aux pannes de processeurs.*

Ensuite, dans la macro-étape mSn.3, l'opération o_{best} la plus prioritaire est ordonnancée sur ses processeurs correspondants $\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_{best})$. Dans cette macro-étape, la date de début de l'opération sélectionnée peut être réduite par la réplication de certains de ses prédécesseurs sur ses processeurs correspondants $\mathcal{P}_{opts}^{\mathcal{N}pf+1}(o_{best})$ en utilisant la

fonction « *Minimiser_la_date_de_début* » proposée par Ahmad et al. dans [AHM 98] et présentée dans la figure 6.

Minimiser_la_date_de_début(o,p) :

1. Calculer $S^{(n)}_{worst}(o,p)$;
2. SI $S^{(n)}_{worst}(o,p)$ est indéfini ALORS Retour (o ne peut être implantée sur p) ; FIN SI
3. Chercher le Plus Tard Prédécesseur (PTP) de o ;
4. Minimiser la date de début de ce PTP en appelant récursivement la fonction *Minimiser_la_date_de_début(PTP,p)* ;
5. Calculer le $nouveau\ S^{(n)}_{worst}(o,p)$;
6. SI ($nouveau\ S^{(n)}_{worst}(o,p) \geq S^{(n)}_{worst}(o,p)$) ALORS
   a. Annuler toutes les réplications effectuées par l'étape 4 ;
   b. Implanter o sur p à $S^{(n)}_{best}(o,p)$;
   c. Implanter les `comms` induites par l'étape b ;

   SINON Chercher le nouveau PTP de o et aller à l'étape 4 ;
   FIN SI

Figure 6. *Minimiser la date du début de l'opération o implantée sur le processeur p.*

Pour chaque paire $\langle$prédécesseur, copie d'opération$\rangle$, les `comms` sont implantées si et seulement si toutes les copies de l'opération prédécesseur sont implantées sur des processeurs différents de celui de la copie d'opération (voir la figure 3).

4. Exemple

Nous avons implanté l'heuristique dans l'outil SYNDEX [1] [GRA 99], qui est un environnement graphique interactif de développement pour applications temps réel. Nous avons appliqué notre heuristique sur l'exemple de la figure 2.1 pour tolérer une seule panne de processeur $\mathcal{N}pf = 1$. Après deux étapes, nous obtenons l'ordonnancement de la figure 7. Les opérations sont représentées par des boîtes dont la hauteur est proportionnelle à leur durée d'exécution (pour les `comps` et `extios`) ou de communication (pour les `comms`).

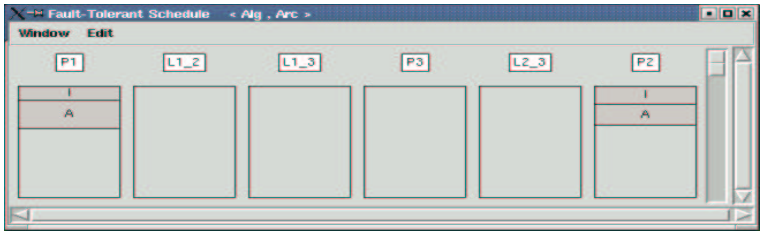

Figure 7. *Ordonnancement après l'étape 2.*

1. SYNDEX : http ://www-rocq.inria.fr/syndex

Dans l'étape suivante, parmi les opérations candidates, l'opération C est sélectionnée avec une pression d'ordonnancement égale à 9.73, 10.53 et 9.23 respectivement sur $P1$, $P2$ et $P3$. Cependant, si A est répliquée sur $P3$, la pression d'ordonnancement de C sur $P3$ devient 5.73, ce qui signifie que la date de début d'exécution de C sur $P3$ est réduite. Donc, une copie de A est placée sur $P3$ et deux copies de C sont placées sur $P3$ et $P1$ (voir figure 8). Comme cela est montré sur la figure 8, l'opération A reçoit ses données d'entrée en deux exemplaires. Après le placement de C, nous obtenons l'ordonnancement de la figure 8, où les communications induites ont été également implantées.

Figure 8. *Ordonnancement après l'étape 3.*

L'ordonnancement final est montré sur la figure 9, où chaque opération est répliquée en au moins deux exemplaires. On constate que la contrainte temps-réel est satisfaite : en effet, la longueur totale de l'ordonnancement est égale à 15.05 qui est inférieure à $\mathcal{R}tc$.

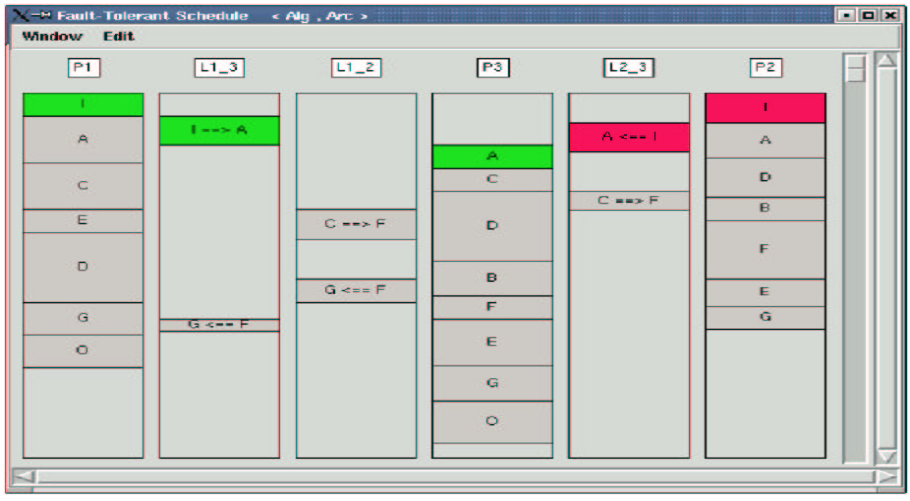

Figure 9. *L'ordonnancement final.*

La figure 10 montre l'ordonnancement obtenu quand le processeur $P1$ tombe en panne. Remarquez que sur P3, A doit attendre l'arrivée des données en provenance de P2 qui arrivent plus tard que celles produites par P1 dans l'ordonnancement de la figure 9. La longueur totale de l'ordonnancement obtenu quand un des processeur $P1$, $P2$ ou $P3$ tombe en panne est 15.35, 15.05 et 12.6 respectivement. Donc la contrainte temps réel est toujours satisfaite puisque ces trois valeurs sont inférieures à $\mathcal{R}tc$.

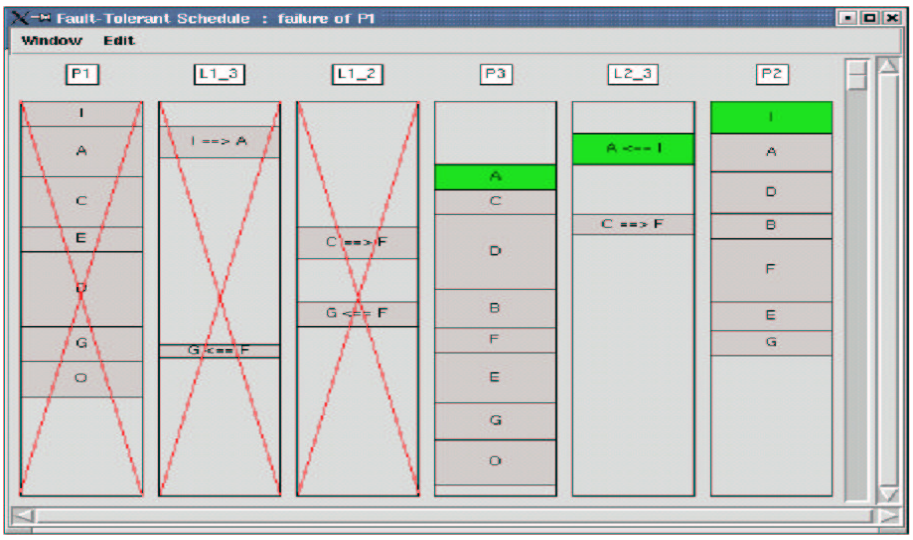

Figure 10. *L'ordonnancement obtenu quand $P1$ tombe en panne.*

4.1. *Comportement d'exécution*

L'heuristique proposée dans cet article peut tolérer $\mathcal{N}pf$ pannes de processeurs par la réplication active de chaque opération sur au moins $\mathcal{N}pf$+1 processeurs distincts.

Si aucune panne se produit dans le système, chaque opération reçoit ces données d'entrée en $\mathcal{N}pf$+1 exemplaires ; dès qu'elle reçoit les premières répliques de données d'entrées, l'opération commence son exécution et ignore les autres répliques.

Si k pannes (avec $k \leq \mathcal{N}pf$) se produisent dans le système, chaque opération reçoit ses données d'entrée en ($\mathcal{N}pf$+1)-k exemplaires ; là aussi, dès qu'elle reçoit les premières réplique de données d'entrées, l'opération commence son exécution et ignore les autres répliques.

La méthode que nous proposons est adaptée a deux catégories de pannes :

❶ *Les pannes permanentes des processeurs* : pour tolérer ce type de panne où chaque panne de processeur persiste indéfiniment, nous proposons un mécanisme de détection de pannes basé sur un tableau d'états où chaque processeur sauvegarde l'état (actif ou en panne) des autres processeurs. A chaque panne

d'un processeur p, les processeurs actifs changent l'état de p dans leurs tableaux d'états. Dans ce cas, chaque processeur n'envoi ses données que vers les processeurs actifs dans son tableau d'état.

❷ *Les pannes intermittentes des processeurs* : ces pannes de processeurs se reproduisent sporadiquement et chaque processeur en panne peut revenir a son état actif. Par conséquent, dans notre méthode, ce type de pannes ne nécessite pas un mécanisme spécial de détection de pannes, puisque chaque opération o reçoit ses données d'entrées en $\mathcal{N}pf$+1 exemplaires de $\mathcal{N}pf$+1 processeurs distincts en parallèle. Dans ce cas, tout processeur qui redevient actif après une panne intermittente continue à rececoir ses données et à envoyer ses résultats vers les autres processeurs (actif ou en panne).

5. Évaluation des performances

Afin d'évaluer l'algorithme HTBR, nous avons comparé ses performances avec l'algorithme proposé par Hashimoto et al. dans [HAS 02], appelé HBP (« Height-Based Partitioning »), qui est le plus proche du notre. HBP est conçu pour les architectures homogènes, ne considère que la redondance logicielle des opérations de calcul et pas des communications, et ne peut tolérer qu'une seule panne de processeur. Nous avons appliqué les même hypothèses sur notre algorithme HTBR dans cette simulation. L'objectif général de cette simulation est de comparer le surcoût dans la longueur de l'ordonnancement introduit par HTBR et HBP, en l'absence et en présence de pannes.

Nous avons appliqué HTBR et HBP à un ensemble de graphes algorithmes générés aléatoirement et un graphe architecture de $P = 4$ processeurs. Pour générer ces graphes algorithmes aléatoires, nous avons fait varier deux paramètres : le nombre d'opérations $N = 10, 20, ..., 80$ et $CCR = 0.1, 0.5, 1, 2, 5, 10$ qui est le rapport entre le temps moyen de communication des `comms` et le temps moyen d'exécution des `comps` et `extios`.

Le surcoût dans la longueur de l'ordonnancement est calculé comme suit :

$$\text{Surcoût} = \frac{\text{longueur(HTBR ou HBP)} - \text{longueur(heuristique sans tolérance)}}{\text{longueur(heuristique sans tolérance)}} * 100$$

où la longueur de l'heuristique sans tolérance est la longueur totale de l'ordonnancement généré par HTBR avec $\mathcal{N}pf$=0.

Nous avons tracé dans la figure 11 et 12 la moyenne du surcoût de 60 graphes aléatoires pour chaque N et chaque CCR, en l'absence de panne (figure 11(a) et 12(a)) et en présence d'une panne de processeur (figure 11(b) et 12(b)).

La figure 11 montre que les performances de HTBR et HBP décroissent lorsque le nombre d'opérations croît. Ceci s'explique par le fait que chaque opération est répli-

quée en deux exemplaires. Dans la plus part des cas, les résultats montrent surtout que HTBR fonctionne mieux que HBP.

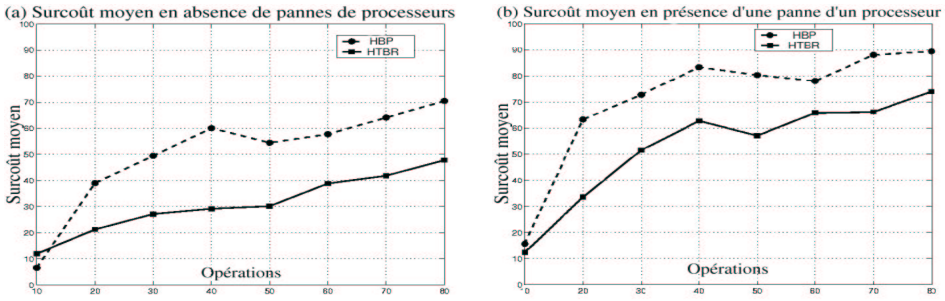

Figure 11. *Effet du nombre d'opérations pour* $\mathcal{N}pf = 1$, $P = 4$ *et* $CCR = 5$.

La figure 12 montre que les performances de HTBR et HBP croissent lorsque CCR croît. Ceci s'explique par la réplication de chaque opération en plus de deux exemplaires afin de réduire les coûts de communications. Pour $CCR >= 1$, les résultats montrent que les performances de HTBR sont meilleurs relativement à HBP.

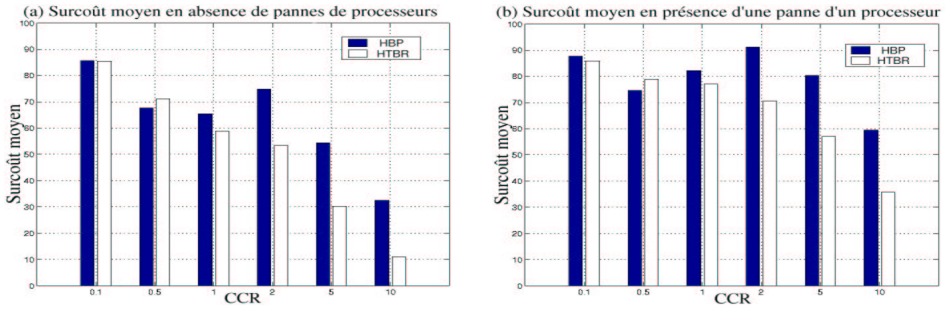

Figure 12. *Effet de CCR pour* $\mathcal{N}pf = 1$, $P = 4$ *et* $N = 50$.

Enfin, le calcul de la complexité en temps des deux algorithmes montre que la complexité en temps de HTBR ($O(N^4)$) est moindre que la complexité en temps de HBP ($O(PN^4)$). Ceci est dû à la stratégie de réplication des tâches utilisée dans HBP, où quelques prédécesseurs de chaque opération candidate o_{cand} sont répliqués sur *chaque processeur* p du graphe architecture afin de miminiser la date de début de o_{cand} sur p. Au contraire, dans HTBR, on ne réplique certains prédécesseurs de chaque opération o_{best} que sur les $\mathcal{N}pf$+1 processeurs $\mathcal{P}_{opts}^{\mathcal{N}pf+1}$ selectionnés à l'étape mSn.1 de l'heuristique de la figure 5.

6. Conclusion

Dans cet article, nous avons présenté une solution pour la génération automatique de code distribué tolérant aux pannes des processeurs pour les systèmes embarqués. Spécifiquement, nous avons présenté une nouvelle heuristique, appelée HTBR (pour « Heuristique Tolérante aux pannes Basée sur la Réplication active »), qui produit automatiquement un ordonnancement statique distribué tolérant aux pannes. Notre solution est basée sur la redondance logicielle des opérations de calculs et de communications.

Nous avons implanté l'heuristique HTBR dans l'outil SYNDEX, un environnement graphique interactif dédié au développement d'applications temps-réel embarquées. Afin d'étudier les performances de notre heuristique, nous avons comparé l'heuristique HTBR avec l'heuristique HBP (pour « Height-Based Partitioning ») proposée par Hashimoto et al. dans [HAS 02]. Les résultats montrent que HTBR se comporte bien mieux que HBP.

Enfin, notre solution ne tolère que les pannes des processeurs. Actuellement, nous travaillons sur une nouvelle solution pour prendre en compte également les pannes des liens de communications, et pour intégrer la notion de fiabilité [SHA 92] dans notre solution.

7. Bibliographie

[AHM 95] AHMAD I., KWOK Y., WU M., « Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors », *Proceedings of the 2nd Australian Conference on Parallel and Real-Time Systems*, Sep 1995, p. 185-192.

[AHM 98] AHMAD I., KWOK Y., « On exploiting task duplication in parallel program scheduling », *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, September 1998, p. 872–892.

[AHN 97] AHN K., KIM J., HONG S., « Fault-Tolerant Real-Time Scheduling using Passive Replicas », *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS)*, December 1997.

[ALO 01] AL-OMARI R., SOMANI A. K., MANIMARAN G., « A New Fault-Tolerant Technique for Improving the Schedulability in Multiprocessor Real-time Systems », *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, San Francisco, April 2001, IEEE Computer Society.

[CHE 99] CHEVOCHOT P., PUAUT I., « Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategie », *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, HongKong, China, December 1999, p. 356–363.

[DIM 01] DIMA C., GIRAULT A., LAVARENNE C., SOREL Y., « Off-Line Real-Time Fault-Tolerant Scheduling », *Euromicro Workshop on Parallel and Distributed Processing*, Mantova, Italy, February 2001, p. 410–417.

[GAR 79] GAREY M., JOHNSON D., *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[GIR 00] GIRAULT A., LAVARARENNE C., SIGHIREAU M., SOREL Y., « Fault-Tolerant Static Scheduling for Real-Time Distibuted Embedded Systems », rapport n° 4006, September 2000, INRIA.

[GRA 99] GRANDPIERRE T., LAVARENNE C., SOREL Y., « Optimized Rapid Prototyping for Real-Time Embedded Heterogeneous Multiprocessors », *7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, may 1999.

[GUE 96] GUERRAOUI R., SCHIPER A., « Fault-Tolerance by Replication in Distributed Systems », *Reliable Software Technologies — Ada-Europe'96*, Springer-Verlag, 1996, p. 38–57.

[HAS 02] HASHIMOTO K., TSUCHIYA T., KIKUNO T., « Effective Scheduling of Duplicated Tasks for Fault-Tolerance in Multiprocessor Systems », *IEICE Transactions on Information and Systems*, vol. E85-D, n° 3, 2002, p. 525–534.

[HIL 98] HILLER M., « Software Fault-Tolerance Techniques from a Real-Time Systems Point of View - an overview », rapport, november 1998, Department of Computer Engineering Chalmers University of Technology, SE-412 96 Göteborg Sweden.

[JAL 94] JALOTE P., *Fault-Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[OH 97] OH Y., SON S. H., « Scheduling Real-Time Tasks for Dependability », *Journal of Operational Research Society*, vol. 48, n° 6, 1997, p. 629-639.

[QIN 00] QIN X., HAN Z., JIN H., PANG L. P., LI S. L., « Real-time Fault-tolerant Scheduling in Heterogeneous Distributed Systems », *Proceeding of the 2000 International Workshop on Cluster Computing-Technologies, Environments, and Applications (CC-TEA'2000)*, Las Vegas, Nevada, USA, June, 2000, june 2000.

[RUS 94] RUSHBY J., « Critical System Properties : Survey and Taxonomy », *Reliability Engineering and Systems Safety*, vol. 43, n° 2, 1994, p. 189–219.

[SHA 92] SHATZ S., WANG J., GOTO M., « Task Allocation for Maximizing Reliability of Distributed Computer Systems », *IEEE Trans. Computers*, vol. 41, September 1992, p. 156-168.

[TOR 00] TORRES-POMALES W., « Software Fault Tolerance : A Tutorial », 2000.

[YAN 93] YANG T., GERASOULIS A., « List Scheduling With and Without Communication Delays », *Parallel Computing*, vol. 19, n° 12, 1993, p. 1321–1344.