Robustesse des applications temps-réel multicoeurs : techniques de construction d'un ordonnacement équitable tolérant aux pannes matérielles

Yves Mouafo Tchinda

THESE

Pour l'obtention du grade de
DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE
(Diplôme National – Arrêté du 25 mai 2016)

Ecole Doctorale : Sciences et Ingénierie pour l'Information, Mathématiques
Secteur de recherche : Informatique et Application

Présentée par :

Yves Marcellin MOUAFO TCHINDA

****

Robustesse des Applications Temps-Réel Multicœurs : Techniques de Construction d'un Ordonnancement Equitable Tolérant aux Pannes Processeurs

****

Directrice de Thèse : **Annie CHOQUET-GENIET**

Co-encadrante : **Gaëlle LARGETEAU-SKAPIN**

Soutenue le 12 octobre 2017

Devant la Commission d'Examen

JURY

Rapporteurs

Maryline CHETTO	Professeure, Université de Nantes
Claire PAGETTI	Ingénieure de recherche, ONERA/DTIM Toulouse

Membres

Emmanuel GROLLEAU	Professeur, LIAS/ISAE-ENSMA
Annie CHOQUET-GENIET	Professeure, LIAS/Université Poitiers
Julien FORGET	Maître de Conférences, IRCICA/Polytech'Lille
Gaëlle LARGETEAU-SKAPIN	Maître de Conférences, XLIM/Université Poitiers

« Car celui qui cherche trouve »

Matthieu 7 :8

A mon père Sa Majesté TCHINDA Jean dit FO'ONAPGOUONG ,

A ma défunte mère MAGNI NJUNE.

Vos conseils, vos encouragements et votre amour m'ont accompagné durant tout mon cursus académique et particulièrement durant ces études doctorales.

Ce travail est le fruit de votre semence, je vous le dédié.

A toi mon fils Emmanuel David MOUAFO TCHINDA.

Tu es venu en fin de cette thèse tel un rayon de lumière pour me redonner espoir et ouvrir de nouvelles perspectives après de nombreuses épreuves surmontées.
Je te dédie cette thèse, en ce moment particulier où Emmanuel MACRON est Président de la France et Emmanuel GROLLEAU mon directeur de Laboratoire. Puisses-tu trouver dans leurs travaux une source d'inspiration pour réaliser de grandes œuvres.

Mon vœu le plus cher est que devenu adulte, tu fasses la connaissance de l'EMMANUEL et qu'ayant compris le sens de son sacrifice pour l'humanité et pour ta vie, tu t'abandonnes complètement à lui afin qu'il te guide dans toutes tes entreprises.

« Je puis tout par celui qui me fortifie »

Philippiens 4 :13

Je souhaiterais tout d'abord remercier mes encadrantes de thèse, Annie CHOQUET-GENIET et Gaëlle LARGETEAU-SKAPIN sans qui ce travail n'aurait abouti. Vos conseils avisés et votre encadrement ont permis l'atteinte des objectifs fixés.

A vous Mesdames Maryline CHETTO et Claire PAGETTI, les remarques pertinentes que vous avez portées sur mon manuscrit ont contribuée significativement à son amélioration. Je vous en remercie sincèrement.

Merci également à Julien FORGET et Emmanuel GROLLEAU qui m'ont fait l'honneur d'accepter de participer à mon Jury.

Je remercie particulièrement tout le personnel de l'ENSMA et tous les membres du LIAS pour leur accueil, leur disponibilité et tous les moments partagés.

Merci à La Région La Nouvelle Aquitaine pour le financement de ma thèse.

Mes remerciements s'adressent également :

- *A Michaël BARON qui m'a accompagné depuis mon stage Ingénieur et m'a encouragé à postuler pour cette thèse ;*
- *A Maman Sylvie AGNOROGOULET dont le soutien financier, moral et spirituel a été d'un grand apport dans mon cheminement ;*
- *A mes frères Hippolyte, Théodore, Remy, Serge, Patrice, Aristide, Williams et à autres frères et sœurs qui m'ont témoigné toute leur affection ;*
- *A Mmes Evelyne SAMBISSA, Nathalie HERY et M. Ismaël LIBIZANGOMO ;*
- *A Valéry TEGUIAK, Géraud FOKOU, Georges KEMAYO et Raoul TIAM pour leur amitié et leur solidarité au cœur des épreuves ;*
- *A mes frères et sœurs de l'Eglise Protestante Evangélique de Poitiers, pour leur soutien multiforme ;*
- *A l'Association des Stagiaires et Etudiants Camerounais de Poitiers ;*
- *A Laurentine et Gilbert OGOULA, ainsi qu'à Mme Cyrille MBIA à Libreville ;*
- *A l'Honorable Sénateur SONKIN Etienne dit FO'OKWALA ;*
- *A Mlle Audrey POINT pour son précieux accompagnement et à toute la famille POINT pour son accueil ;*
- *A Dagobert et Christine TANKOUA pour leur soutien indéfectible ;*
- *A ma tante Lucienne WOUMBE et mes mamans Rachel et Nathalie TCHINDA.*

A tous ceux qui, par leurs gestes, leurs actions, leurs prières ou leurs pensées, ont été de près ou de loin d'une contribution, aussi infime soit-elle, à la réussite de cette thèse, je vous prie de trouver ici, l'expression de ma profonde gratitude.

Table des matières

Chapitre

1 Introduction Générale

Sommaire

Résumé

Ce chapitre présente le contexte général de notre étude, à savoir les systèmes temps-réel et les architectures multicoeurs. Cette présentation débouche sur la problématique que nous traitons et les résultats obtenus sont annoncés. Le chapitre se termine par la présentation de l'organisation globale du document.

L'évolution technologique des semi-conducteurs réalisée ces dernières décennies a rendu possible la mise sur une seule puce de millions de transistors [1]. Grâce à cette évolution, une nouvelle génération de processeurs a vu le jour : les processeurs multicœurs. Cette génération a l'avantage d'augmenter les capacités de traitement tout en réduisant le poids, la consommation d'énergie et le coût financier des processeurs [2]. Depuis lors, on note une intégration sans cesse croissante des processeurs multicœurs au sein des systèmes embarqués temps-réel. C'est par exemple le cas des systèmes avioniques et automobiles dont la migration vers le multicœur est due à l'intégration de nouvelles fonctionnalités critiques [3]. En effet, dans l'automobile, on voit l'apparition des technologies X-by-wire qui ont pour rôle de remplacer les systèmes de contrôle traditionnellement construits mécaniquement (frein, conduite, etc.). Dans l'aéronautique, c'est l'avionique modulaire intégrée (IMA) qui est la tendance de conception actuelle afin de remplacer des calculateurs dédiés par des calculateurs généralistes qui factorisent des fonctionnalités différentes, apportant une modularité de conception. Cela engendre un besoin croissant en capacité de calcul et maîtrise des coûts de production, d'où le recours aux composants sur étagère (COTS :Commercial Off-The-Shell) qui reposent essentiellement sur une architecture multicoeur [4].

Bien que le multicœur ait de nombreux avantages par rapport aux architectures monocœur, il est susceptible de subir des pannes qui peuvent affecter temporairement ou durablement un ou plusieurs cœurs du processeur [5]. Cela peut conduire le système en cours d'exécution à des fautes temporelles. C'est pourquoi les concepteurs de systèmes devant s'exécuter sur une telle architecture accordent une place primordiale aux mécanismes de tolérance aux pannes. La présente thèse s'inscrit dans ce sillage et propose des techniques de tolérance pour les systèmes temps-réel ordonnancés par un algorithme équitable.

Dans cette introduction générale, nous commençons par présenter le contexte de l'étude suivant deux principales articulations : les systèmes temps réel et les architectures multicœurs. Ensuite, la problématique de panne est soulevée et nos contributions pour la résoudre sont présentées. Nous terminons par l'organisation globale du document.

I. Les systèmes temps-réel

I.1. Définition et caractéristiques

On qualifie de *temps réel* tout système informatique dont le fonctionnement est conditionné par des contraintes temporelles [6] [7]. La validité d'un tel système ne dépend pas seulement de la justesse des calculs, mais aussi de l'instant de production des résultats. De ce fait, un résultat juste, mais ne respectant pas ses contraintes temporelles, est un résultat inutilisable assimilable à un résultat erroné. La présence de contraintes temporelles est l'aspect fondamental qui distingue le système temps-réel des systèmes informatiques classiques. La validation d'un

tel système doit donc s'opérer aussi bien sur le plan fonctionnel (exactitude des résultats) que sur le plan temporel (respect des contraintes temporelles). Dans la plupart des cas, il s'agit de systèmes de contrôle-commande chargés de suivre le déroulement d'un procédé physique. A cet effet, il est en relation avec l'environnement physique externe par l'intermédiaire de capteurs et/ou d'actionneurs. C'est ainsi qu'il reçoit des informations sur l'état du procédé externe, traite ces données et, en fonction du résultat, prend une décision qui agit sur cet environnement extérieur afin d'assurer un état acceptable. Souvent, de tels systèmes sont embarqués sur le procédé contrôlé, ce qui implique davantage de contraintes d'encombrement (taille, poids, forme), de ressources (puissance de calcul, mémoire) et d'énergie. Leurs principales caractéristiques sont [8] : la diversité des dispositifs d'entrées/sorties, la prise en compte des comportements concurrents, le respect des contraintes temporelles et la sûreté de fonctionnement. Pour assurer la dernière caractéristique, plusieurs éléments sont à prendre en compte tels que la correction, la prévisibilité, la continuité de service et la tolérance aux pannes.
Dans le cadre de cette thèse, nous nous intéressons à la tolérance aux pannes.

En fonction du niveau d'exigence attendu quant au respect des contraintes temporelles, on peut classer les systèmes temps-réel en deux grandes catégories :

- Les systèmes temps-réel *mou* (*soft real-time system*), où les fautes temporelles sont tolérables, mais leur multiplication et/ou des retards trop importants dégradent les performances de l'application. C'est le cas des jeux vidéos, des applications multimédias, de la téléphonie mobile, etc.
- Les systèmes temps-réel *durs* (*hard real-time system*), où les fautes temporelles ne sont pas tolérables. C'est le cas des applications que l'on trouve dans les secteurs aéronautique, automobile et nucléaire.

Il existe deux variantes : les systèmes temps-réel *fermes* où un nombre limité de fautes temporelles est accepté et les systèmes *incrémentaux* où la qualité du résultat produit s'améliore avec le temps.
L'exécution du système peut se faire sur une plateforme matérielle composée d'un processeur à une unité de traitement ou un coeur (on parle de système mono-processeur) ou de plusieurs processeurs ayant chacun une unité de traitement (on parle de système multi-processeur). Aujourd'hui, la tendance est aux processeurs à plusieurs coeurs (on parle de système multicoeurs). Nous présentons cette dernière plateforme après la description de l'architecture logicielle du système.

I.2. Modèle temporel

Un système temps réel est constitué d'un ensemble de *tâches*. Une tâche contrôle le flot d'exécution d'un programme pour différentes données. Les instructions exécutées forment ce que l'on appelle un *travail* (*job*). Ainsi, une tâche est constituée d'un ensemble infini de travaux. On appelle aussi ces travaux, les *instances* de la tâche. Nous nous intéressons ici au modèle

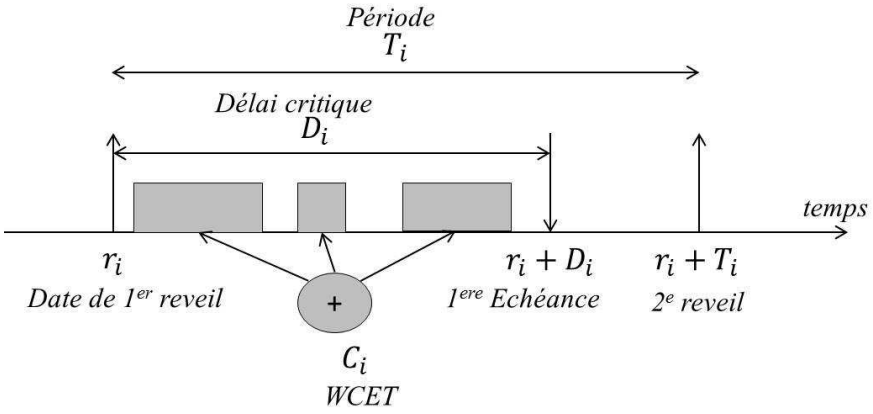

FIGURE 1.1 – Modélisation d'une tâche périodique $\tau_i < r_i, C_i, D_i, T_i >$.

temporel des tâches. A cet effet, nous adoptons le modèle le plus fréquemment employé [9], à savoir le modèle de Liu et Layland [10]. Ce modèle a ensuite été complété par [11] et [12]. Les notations varient d'un auteur à l'autre mais le choix a été fait ici de reprendre les notations de Goossens [13]. Les définitions proviennent principalement des articles [14] et sont complétées par [15] et [16].
Selon la manière dont les travaux sont activés, on distingue principalement trois natures de tâches :

- Les tâches *périodiques* sont activées régulièrement à période fixe ;
- Les tâches *sporadiques* sont activées de manière irrégulière mais avec toutefois au moins une propriété sur la durée entre l'arrivée de deux travaux consécutifs ;
- Les tâches *apériodiques* sont activées de manière irrégulière.

Une tâche τ_i, périodique ou sporadique, est caractérisée par un triplet (C_i, D_i, T_i) qui représente :

- la *durée d'exécution* C_i dans le pire cas (WCET) de chaque travail de la tâche ;
- sa *période d'activation* T_i. Dans le cas de tâches sporadiques, c'est la durée minimale entre ses activations successives ;
- son *échéance relative* ou *délai critique* D_i. C'est la durée entre l'arrivée d'un travail et son échéance (un travail qui arrive à l'instant t doit se terminer avant l'instant $t + D_i$). Une tâche est dite *critique* si le non respect de son échéance peut entraîner des conséquences catastrophiques sur le système.

A ces trois valeurs peuvent se rajouter, pour les tâches périodiques, la date de première activation (*first release date*) r_i. Nous noterons une telle tâche $\tau_i < r_i, C_i, D_i, T_i >$.
La Figure 1.1 illustre la modélisation d'une tâche périodique.
Dans cette étude, nous nous intéressons aux tâches périodiques.

Les tâches et le système se caractérisent par les paramètres temporels. C'est ainsi qu'on parlera de :

- tâche *concrète* lorsqu'on connaît les dates d'activation de ses travaux. C'est le cas par exemple des tâches périodiques lorsqu'on connaît la date de la première activation ;
- système de tâches à *départs simultanés* si les tâches démarrent au même instant (i.e $\forall i$, $r_i = 0$). Sinon, on parle de système à *départs différés* ;
- système de tâches à *échéances implicites ou à échéances sur requêtes* si $\forall i$, $D_i = T_i$ et de tâches à *échéances contraintes* si $\forall i$, $D_i \leqslant T_i$. Dans le cas contraire, on parle de tâches à *échéances arbitraires*.

Enfin, deux tâches sont dites *dépendantes* si l'une a besoin du résultat d'exécution de l'autre pour pouvoir être exécutée à son tour (contrainte de précédence) ou si elles partagent d'autres ressources critiques que le processeur. Dans le cas contraire, s'il n'y a ni section critique, ni contrainte de précédence, elles sont dites *indépendantes*.

On associe également des grandeurs au système de tâches dans sa globalité :

- L'*hyper-période* d'un système de n tâches périodiques, noté H, est égale au plus petit multiple commun des périodes des tâches qui la composent ($H = ppcm(T_i), i = 1..n$). Elle représente la période du système global, c'est-à-dire, l'intervalle de temps après lequel l'exécution du système se reproduit à l'identique, si le système est à départs simultanés. Dans le cas de systèmes à départs différés, la cyclicité se met en place après une phase de montée en charge [17].

- Le *facteur d'utilisation* d'une tâche τ_i est égal au quotient de son WCET par sa période ($U_i = \frac{C_i}{T_i}$). Il représente le taux d'activité du processeur pour l'exécution des instances successives de la tâche. La *charge* du système est égale à la somme des facteurs d'utilisation des tâches qui le constituent ($U = \sum_{i=1}^{n} U_i$). C'est le taux d'activité du processeur pour l'exécution du flot d'instances des tâches de l'application. L'utilisation réelle du processeur par une tâche à échéance contrainte se situe entre son activation et son échéance. On l'estime par le *facteur de charge* qui est quotient de son WCET par son délai critique ($CH_i = \frac{C_i}{D_i}$). Le facteur de charge du système est la somme des facteurs de charge de ses tâches ($CH = \sum_{i=1}^{n} CH_i$).

Dans cette thèse nous considérons des systèmes composés de tâches critiques indépendantes et périodiques, à départs simultanés et à échéances sur requête.

I.3. Modèle d'exécution

I.3.a. L'ordonnancement des tâches

L'exécution d'un système temps-réel sur sa plateforme matérielle est réalisée par un dispositif logiciel appelé *exécutif temps réel*. Parmi les exécutifs temps réel utilisés dans les systèmes

embarqués on peut citer : VxWorks [18], RT Linux [19], Ada Temps Réel [20], Osek/VDK [21] et POSIX [22]. Les trois derniers sont en réalité des normes définissant les spécifications qu'une implémentation se doit de suivre. Un exécutif est composé d'un noyau et de modules ou bibliothèques complémentaires. Le noyau fournit des services de base [6] dont l'ordonnanceur qui est au cœur de notre problématique.

L'*Ordonnancement* est donc une fonctionnalité de l'exécutif qui organise l'exécution des instances des tâches suivant une politique préétablie. Un ordonnancement *temps-réel* devra garantir le respect des contraintes temporelles. Dans ce cas, on dira que l'ordonnancement est *valide*. Le résultat produit par l'ordonnanceur est une séquence qui définit l'ordre dans lequel les tâches ont été affectées aux différentes unités de traitement. Plusieurs qualificatifs peuvent caractériser un ordonnancement :

- *conservatif/non conservatif* : l'ordonnancement est conservatif lorsqu'il fonctionne sans insertion de temps creux (temps durant lequel une unité de traitement est libre, alors qu'il y a des tâches en attente). Dans le cas contraire il est non conservatif.
- *En-ligne/hors-ligne* : l'ordonnancement est hors-ligne si la séquence d'ordonnancement est établie avant le lancement de l'application pour être répétée à l'infini ; l'ordonnanceur dispose alors de la séquence. Il est en-ligne si la séquence est construite en cours d'exécution du système ; l'ordonnanceur dispose alors d'un algorithme.
- *Préemptif/non-préemptif* : l'ordonnancement est préemptif s'il autorise la suspension temporaire de l'exécution d'une tâche au profit d'une autre tâche plus prioritaire. Dans le cas contraire, il est non préemptif.
- *Statique/dynamique :* l'ordonnancement est statique (prédictif) si la priorité accordée aux tâches ne change pas au cours de l'exécution. À l'inverse, un ordonnancement est dynamique (réactif) si la priorité entre les tâches varie au cours de l'exécution ;
- *Monoprocesseur/multiprocesseur* : un ordonnancement est monoprocesseur s'il permet d'ordonnancer un système sur une architecture monoprocesseur. Un ordonnancement multiprocesseur permet d'ordonnancer un système sur une architecture multiprocesseur ou multicoeur. Dans ce cas, on distingue la *politique globale* dans laquelle il y a une seule file d'attente de tâches et un seul ordonnanceur pour tous les coeurs et *la politique partitionnée* où chaque cœur dispose de sa file de tâches et de son ordonnanceur. Ces politiques sont présentées dans le chapitre suivant.

Pour assurer le respect des contraintes temporelles, une analyse d'ordonnancement doit être effectuée. Elle se focalise principalement sur deux problèmes :

- **Le problème de faisabilité** : en considérant les tâches, leurs contraintes ainsi que les unités de traitement dont dispose l'architecture, le but est de dire s'il existe un ordonnancement qui satisfait toutes les échéances, c'est-à-dire un ordonnancement valide ;
- **Le problème d'ordonnançabilité** : en considérant les tâches, leurs contraintes, les unités de traitement dont dispose l'architecture ainsi qu'un algorithme d'ordonnance-

ment, le but est de déterminer si l'ordonnancement obtenu avec cet algorithme respecte toutes les contraintes.

Il en ressort qu'un système ordonnançable par un algorithme donné est toujours faisable alors qu'un système faisable n'est pas toujours ordonnançable par un algorithme donné. L'analyse d'ordonnancement peut se faire suivant trois approches :

1. **L'approche analytique** qui consiste à identifier le ou les pires cas d'exécution pour une tâche dans un système donné et à en déduire une expression analytique permettant d'évaluer les caractéristiques du système. Cette expression est appelée condition d'ordonnançabilité ou/et de faisabilité. Selon les paramètres des tâches, il peut s'agir simplement d'une condition suffisante ou bien d'une condition nécessaire et suffisante d'ordonnançabilité [23].

2. **La simulation** qui consiste à modéliser le comportement du système à l'aide d'un formalisme adéquat puis à l'exécuter avec un outil de simulation sur des scénarios définis par l'utilisateur. Les informations résultantes peuvent être de simples traces de l'exécution du système ou des évaluations sur certains comportements du système. L'un des principaux inconvénients de la simulation est qu'elle peut être optimiste [24]. En effet, comme il est difficile de réaliser une étude exhaustive de tous les scénarios possibles, elle peut montrer qu'un système n'est pas ordonnançable (en montrant un scénario non ordonnançable), mais elle ne peut prouver qu'un système est ordonnançable (même si tous les scénarios testés le sont). Des outils de simulations existent et sont utilisés dans l'industrie, parmi lesquels on peut citer Cheddar [25].

3. **Le Model-checking** qui est une méthode permettant l'exploration exhaustive de l'espace d'états du système. Ces méthodes permettent de plus de vérifier de nombreuses propriétés sur les systèmes autres que l'ordonnancement [26]. Le formalisme utilisé est souvent celui des systèmes de transitions étendus temporellement : automates, réseaux de Petri. Cependant, il existe des extensions spécifiques aux systèmes temps-réel comme les réseaux de Petri étendus à l'ordonnancement [27], les modèles géométriques [17] et les chaines de Markov [28]. L'inconvénient majeur est que l'espace d'états que les méthodes de model-cheking doivent prendre en compte a une taille exponentielle ; par conséquent, cela limite la taille des modèles analysables.

Dans cette thèse, l'analyse d'ordonnancement nous permet de vérifier si les méthodes de tolérance proposées permettent d'assurer le respect des contraintes temporelles en dépit de la défaillance d'un ou de plusieurs cœurs du processeur. Pour cela, nous utilisons la simulation pour une validation expérimentale des méthodes proposées et une approche analytique pour les prouver.

I.3.b. Les algorithmes d'ordonnancement

Globalement, les algorithmes d'ordonnancement s'appuient sur l'affectation de priorités aux tâches : on parle d'ordonnancement dirigé par les priorités. En fonction de la manière dont ils attribuent les priorités, on peut classer les algorithmes en trois catégories :

- Les *algorithmes à priorités fixes au niveau des tâches*, assignent des priorités fixes à chaque tâche lors de la conception du système ; pendant l'exécution du système, chaque travail hérite de la priorité de sa tâche. Dès lors, tous les travaux issus d'une même tâche ont la même priorité. Un exemple d'algorithme à priorité fixe au niveau des tâches, pour des tâches périodiques, est RM (Rate Monotonic) [10].
- Les *algorithmes à priorités fixes au niveau des travaux*, assignent pendant l'exécution du système, des priorités fixes aux travaux. C'est-à-dire que les travaux d'une même tâche peuvent avoir des priorités différentes mais la priorité d'un travail ne varie pas au cours du temps. EDF (Earliest Deadline First) [10] est un tel algorithme.
- Les *algorithmes à priorités dynamiques au niveau des travaux*, définissent à chaque instant une priorité pour chaque travail (actif). C'est le cas le plus général, il n'y a aucune restriction et la priorité d'un travail peut, par exemple, varier au cours du temps. LLF (Least Laxity First) [11] et PD2 [29] sont des exemples.

Les critères de comparaison des algorithmes sont les suivants :

- **Optimalité** : un algorithme a est optimal s'il produit un ordonnancement valide pour tout ensemble faisable de tâches (i.e pour lequel il existe au moins un ordonnancement valide).
- **Dominance** : a domine b si tous les jeux de tâche ordonnançables par b le sont aussi par a et s'il existe des jeux de tâches ordonnançables par a qui ne le sont pas par b.
- **Equivalence** : a et b sont équivalents si tous les jeux de tâches ordonnançables par a le sont aussi par b, et inversement.
- **Incomparable** : a et b sont incomparables s'il existe des jeux de tâches ordonnançables par a qui ne sont pas ordonnançables par b et inversement.

II. L'architecture multicoeur

II.1. Une architecture en plein essor

Depuis les débuts de l'ère informatique, l'accroissement de la puissance de calcul des processeurs est la préoccupation majeure des constructeurs. En effet, jadis on disposait des processeurs à une unité de traitement appelés processeurs monocoeurs. Très vite, leurs capacités de traitement sont devenues insuffisantes à mesure que les besoins des utilisateurs croissaient et que les fonctionnalités des ordinateurs augmentaient. C'est ainsi que l'augmentation de la fréquence, couplée à des changements architecturaux au sein de ces processeurs a permis d'obtenir un gain de puissance. Dans un premier temps, le gain de place a permis aux constructeurs d'ajouter des

composants à leurs processeurs (registres, pipelines, prédiction de branchement, réordonnancement d'instructions, etc) produisant des processeurs de plus en plus riches et sophistiqués. Parmi les évolutions marquantes, on pourra noter la naissance des processeurs superscalaires qui permettent l'exécution simultanée de plusieurs instructions d'un programme séquentiel, lorsque la dépendance des données le permet, chacune dans un pipeline différent. Ensuite, l'amélioration de la finesse de gravure des transistors sur une puce a apporté un progrès remarquable. En une trentaine d'années, nous sommes passés de microprocesseurs de quelques milliers de transistors, 29000 par exemple pour le 8086 d'INTEL (1979), à plusieurs milliards de transistors avec 2,3 milliards pour le Nehalem-EX Xeon octo-cœur sorti en mars 2010.

Cependant, ces changements architecturaux compliquent grandement les processeurs. De plus l'augmentation de la fréquence implique une augmentation de la consommation d'énergie et du dégagement de chaleur, rapprochant dangereusement les processeurs de la limite de chaleur que supportent leurs composants [30]. Partant de ce constat, les concepteurs de processeurs ont adopté une autre approche. Ainsi, au lieu de complexifier le cœur d'un processeur, leur idée est de multiplier le nombre de cœurs au sein d'un processeur. Cela permet de continuer à augmenter la puissance de calcul globale d'un processeur, tout en contournant les difficultés liées à l'augmentation de fréquence et à la dissipation thermique. La miniaturisation permet de graver directement plusieurs cœurs sur une même puce : on parle alors de processeurs multicœurs. Au lieu d'accélérer le flot d'exécution du processeur, les multicœur créent d'autres flots d'exécution parallèles [31]. Les puces multicœurs constituent le noyau des architectures actuelles. Aujourd'hui, on trouve des processeurs multicoeurs dans divers équipements tels que [32] : les smartphones, les plateformes embarquées telles que Rasperry Pi 2 et même dans les plateformes IoE telles que Google Glass.

II.2. Différences avec l'architecture multiprocesseur

Les multicœurs ont la particularité de regrouper plusieurs cœurs sur une même puce, produisant ainsi des machines multiprocesseurs à moindre coût. Si beaucoup peuvent considérer qu'un cœur est la même entité qu'un processeur mono-cœur, cela est loin d'être exact. En effet, les systèmes multiprocesseurs (SMP) sont connus et utilisés depuis plusieurs décennies par la communauté du calcul scientifique. Ils n'ont cependant pas les mêmes spécificités que les multi-cœur (CMP), notamment au niveau des accès mémoire. Un processeur multicoeur est un processeur comportant plusieurs cœurs reliés ensembles par un tronc commun tandis qu'un ordinateur multi-processeurs comporte vraiment plusieurs processeurs distincts et séparés. La principale différence entre les deux types d'architecture est que pour un processeur comportant plusieurs cœurs la séparation de l'information se fait au niveau du Bus processeur (CMP : Chip Level Multiprocessing) tandis que pour le système comportant plusieurs processeurs, elle se fait au niveau du Bus système (Front Side Bus). Un processeur multicœur partage donc souvent la même mémoire cache L2 entre ses coeurs.
L'usage d'un système équipé d'un processeur multicoeur bénéficie de quelques avantages par

rapport à un système multiprocesseur. La distance entre deux unités de traitement est grandement réduite ce qui augmente la rapidité d'échange d'informations (les cœurs sont directement liés ensemble par le Bus processeur). De plus, un des problèmes majeurs d'un système multiprocesseur est dû au fait qu'il comporte plusieurs composants isolés qui travaillent séparément, parfois sur les mêmes tâches. Ces composants distincts ne doivent pas accéder au même espace mémoire ou périphérique d'entrées/sorties simultanément. C'est pourquoi il doit y avoir communication et synchronisation entre les processeurs. C'est donc un gros avantage pour un processeur multicoeur puisque l'échange d'informations se faisant à l'intérieur du processeur est plus rapide.

II.3. Structure physique

Les plateformes multicoeur peuvent être regroupées selon le mode d'accès à la mémoire ou selon les caractéristiques des coeurs. En fonction du mode d'accès à la mémoire, on distingue les processeurs multicoeur à accès uniforme à la mémoire (UMA : Uniform Memory Access) et les processeurs multicoeur à accès non uniforme à la mémoire (NUMA : Non-Uniform Memory Access) [33]. La différence entre les deux se situe au niveau des communications avec la mémoire centrale. Dans les processeurs UMA, les communications se font via le bus mémoire. Ceci constitue un goulot d'étranglement en cas de défaillance de ce bus. C'est pour y remédier que les fabricants ont mis au point les processeurs NUMA. L'idée est de distribuer la mémoire en plusieurs bancs, pour ne plus avoir un seul bus mémoire, et ainsi se débarrasser de ce goulot d'étranglement. Chaque banc mémoire est associé à un nœud NUMA. Chaque nœud NUMA peut regrouper plusieurs cœurs. Les nœuds NUMA communiquent via une interconnexion rapide. Malgré la distribution physique de la mémoire, celle-ci apparaît tout de même de façon centralisée et cohérente au programmeur.
La Figure 1.2 (*source [33]*) présente une vue d'ensemble l'architecture de l'Intel Core 2 comprenant deux nœuds NUMA. Les cœurs d'un même nœud partagent le même cache L2.

En fonction des caractéristiques des coeurs, on distingue les plateformes *homogènes* des plateformes *hérétogènes* [34]. Les plateformes homogènes sont constituées de coeurs indentiques c'est-à-dire de même caractéristiques (interchangeables). Les plateformes hétérogènes peuvent être constituées de deux types de coeurs : les coeurs uniformes et les coeurs indépendants. Les cœurs uniformes ont des capacités de calcul proportionnelles entre elles : si une tache s'exécute en t unités de temps sur un processeur P_i, elle s'exécutera en une durée $S_{i,j} \times t$ sur un processeur P_j. $S_{i,j}$ est le facteur d'accélération qui ne dépend que de P_i et de P_j. Les coeurs indépendants ont des vitesses non corrélées différentes et la durée d'exécution d'un travail dépend du coeur qui l'exécute et donc varie d'un coeur à l'autre. De plus, le facteur d'accélération dépend ici des processeurs et de la tâche à exécuter.
Selon [35] les plateformes les plus utilisés sont les plateformes homogènes. C'est pour cette raison que **l'architecture que nous retenons dans cette thèse est composée de coeurs homogènes.**

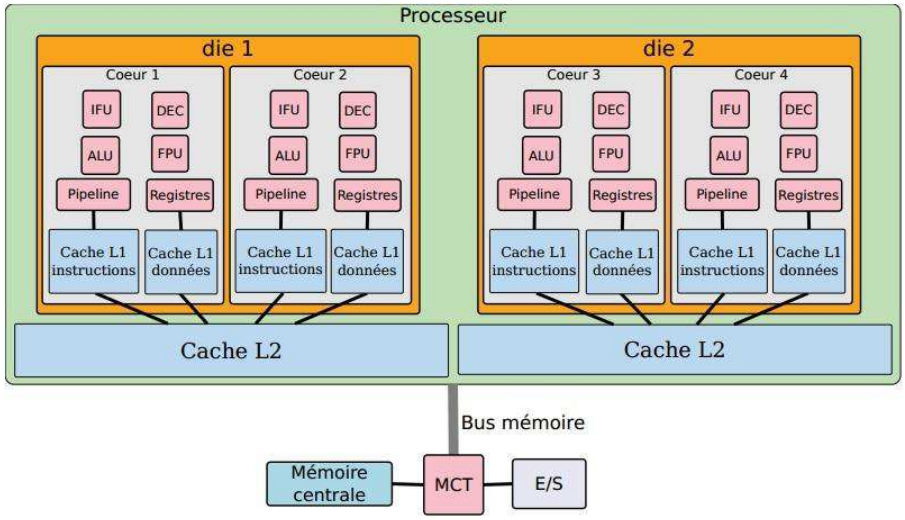

FIGURE 1.2 – Représentation d'une architecture multicoeur, exemple de l'Intel Core 2.

II.4. Architecture puissante, mais faillible !

S'il est indéniable que les processeurs multicoeurs apportent un gain en terme de capacité de calcul, de coût, de poids et d'espace, leur faiblesse réside dans leur structure physique, constituée essentiellement de transistors. En effet, la fiabilité des dispositifs à semi-conducteurs tels que les transistors dépend de leur résistance aux contraintes qui peuvent s'exercer sur les éléments qui les constituent. A cet effet, on peut citer [36] les contraintes internes (électriques, thermiques, mécaniques, etc) et les contraintes externes (température ambiante, interaction électromagnétique..). Si une partie d'un tel dispositif a une structure particulièrement faible, cette partie faible peut très mal réagir à la contrainte appliquée, ce qui peut provoquer des défaillances graves.

Il est à noter que l'augmentation sans cesse croissante du nombre de transistors et la réduction de leurs tailles sur les puces multicoeurs a pour conséquence l'élévation des températures, l'apparition d'effets électromagnétiques et des variations de la tension électrique [37]. Ces facteurs aggravent les risques de défaillance des transistors et augmentent la probabilité de défaillance d'un ou plusieurs coeurs du processeur. **C'est ce problème de défaillance de cœur qui nous intéresse ici.**

III. Problématique

L'architecture multicoeur est désormais la plateforme matérielle de base choisie par plusieurs constructeurs de systèmes embarqués temps-réel critiques [38]. ARM MPCore et IBM Cell sont des exemples de processeurs multicoeurs embarqués dans des systèmes temps-réel [39]. Si du

point de vue de la structure physique une architecture multicoeur est différente d'une architecture multiprocesseur, du point de vue de l'ordonnancement temps-réel, elles sont assimilables. C'est pourquoi les politiques d'ordonnancement multiprocesseurs s'appliquent au multicoeur.
Comme nous l'avons vu plus haut (§$II.4$), à tout instant durant l'exécution du système, un ou plusieurs coeurs du processeur peuvent tomber en panne. Cette panne peut durer quelques instants (panne temporaire) ou être définitive (panne permanente). La première conséquence de la panne est que si elle est détectée tardivement, la portion d'exécution prévue sur le coeur défaillant de l'instant de survenue de la panne jusqu'à l'instant de sa détection est perdue. La deuxième conséquence est que le nombre de coeurs restants peut ne plus garantir l'ordonnançabilité du système.
Ce sont ces problèmes (exécution perdue et étude du nombre de coeurs à prévoir) que nous considérons.

Dans les systèmes temps-réel, l'exigence de satisfaire les contraintes temporelles oblige à prendre en compte l'éventualité des pannes dès la conception du système. En effet, le système doit respecter ses contraintes même en cas de panne. Pour ce qui est des pannes temporaires, l'ordonnançabilité du système est compromise juste durant un laps de temps. A ce sujet plusieurs travaux ont été menés et ont conduit à réduire le temps de récupération du coeur défaillant [40] et même à concevoir des processeurs multicoeurs capables de s'autoréparer [41]. Par contre dans le cas d'une panne permanente, l'ordonnançabilité du système peut être définitivement compromise. Dans certains cas, la panne peut nécessiter le remplacement des unités défaillantes durant l'exécution du système. Ce type de panne, sujet de préoccupations majeures pour la communauté scientifique, retient notre attention. La majorité des résultats publiés (voir chapitre 3) portent essentiellement sur la politique d'ordonnancement partitionnée. Ayant constaté que peu de travaux portent sur la politique globale, nous avons choisi d'y orienter nos travaux. De plus, il existe pour cette politique des algorithmes optimaux, notamment les algorithmes équitables, ce qui justifie notre choix. Enfin, ces algorithmes ont l'avantage de disposer d'une condition nécessaire et suffisante d'ordonnançabilité facile à vérifier. L'algorithme équitable PD2 nous servira de support d'analyse.
La problématique centrale de cette thèse est donc la suivante :

Etant donné un système temps-réel multicoeur homogène constitué de tâches indépendantes et périodiques, à départs simultanés et à échéances sur requêtes ordonnancé par un algorithme équitable, comment garantir la validité de la séquence d'ordonnancement en dépit de la défaillance d'un ou de plusieurs cœurs du processeur ?

Notre objectif est donc de proposer des techniques permettant à un tel système de s'exécuter dans une architecture multicoeur tout en respectant les échéances des tâches, même dans le cas où un ou plusieurs coeurs du processeur tombent en panne. Les résultats obtenus sont

sommairement présentés dans la section suivante.

IV. Contributions et Publications

Afin de mieux aborder la problématique que nous venons de soulever, nous distinguons deux cas de figure : le cas où les travaux manqués à cause de la défaillance des cœurs peuvent être abandonnés (appelé ordonnancement sans reprise) et le cas où ils doivent être repris (ordonnancement avec reprise). La méthode classique de tolérance aux pannes est le recours à la redondance. Pour ce qui nous concerne, dans un premier temps, nous considérons que la panne affecte un seul cœur du processeur. Dans ce cas, lorsque l'ordonnancement est sans reprise, nous montrons que la redondance matérielle limitée, qui consiste à exécuter le système sur une architecture disposant d'un cœur de plus que nécessaire, garantit la tolérance du système. Dans le cas d'un ordonnancement avec reprise, nous proposons trois techniques de tolérance. La première consiste à augmenter le WCET des tâches pour prévoir du temps additionnel afin de compenser la perte d'exécution. C'est une technique sûre et facile à mettre à œuvre. Cependant, elle nécessite plus de redondance matérielle et augmente la charge du système. La deuxième technique crée au sein de chaque tâche une marge de temps entre le délai critique et la période qui sera utilisée pour rattraper l'exécution perdue. Lorsque la panne est détectée, des changements de mode permettent au système de s'adapter à la nouvelle configuration. Cette technique est plus flexible que la première et la validité de l'ordonnancement est garantie. Toutefois, elle s'applique à une plage plus réduite de systèmes ordonnançables. La dernière technique, également flexible, s'applique à tout système ordonnançable. Elle utilise un flux apériodique pour rattraper l'exécution perdue dans les unités de temps creux identifiées dans l'ordonnancement. C'est une technique appropriée pour les systèmes où un nombre limité de dépassements d'échéances est toléré. Enfin, ces techniques peuvent être mixées lorsque les tâches sont de natures distinctes.

Dans un second temps, nous considérons qu'à un instant donné, il y a plusieurs cœurs défaillants. Nous proposons alors deux approches de tolérance : la première est basée sur les changements de mode de criticité et la deuxième sur la suppression des tâches ou des travaux en selon leur importance dans le système.

Les travaux de cette thèse ont donné lieu à des publications dans les conférences et revues suivantes :

- Yves MOUAFO, Annie CHOQUET-GENIET, Gaëlle LARGETEAU. Robustesse des applications temps-réels multicoeurs, Actes de l'Ecole d'Eté Temps-Réel *ETR'2015*, pages 143-146, 2015 ;
- Yves MOUAFO, Annie CHOQUET-GENIET, Gaëlle LARGETEAU. Failure Tolerance for a Multicore Real-Time System Scheduled by PD2. *Proceedings of the 9th Junior Researcher Workshop on Real-Time Computing, JRWRTC'2015*, pages 1-4, 2015 ;

— Yves MOUAFO, Annie CHOQUET-GENIET, Gaëlle LARGETEAU. A Dynamic Feasibility Windows Reconfiguration for a Fault-Tolerant PFair Scheduling. *Proceedings of the 10th International Conference on Verification and Evaluation of Computer and Communications Systems VECOS'2016*, pages , 2016.
— Yves MOUAFO, Annie CHOQUET-GENIET, Gaëlle LARGETEAU. Multicore Scheduling of Real-time Systems Subject to Permanent Failure of One Core with Detection Delay. *International Journal of Critical Computer-Based Systems* (en cours de publication).

V. Organisation du document

La suite de ce document s'organise en deux parties. La première qui présente l'état de l'art comporte trois chapitres. Dans le Chapitre 2, une présentation générale des politiques d'ordonnancement multiprocesseur est abordée. Elle débouche sur la présentation de l'algorithme PD2 retenu pour notre étude. Le Chapitre 3 quant à lui présente un état de l'art des techniques de tolérance aux pannes et ressort les particularités des techniques proposées dans cette thèse. Le Chapitre 4 introduit les notions de changement de mode et de systèmes à criticités multiples qui sont utilisées dans nos techniques de tolérance.

La deuxième partie de ce document est consacrée à nos contributions. Elle comprend trois chapitres. Le Chapitre 5 est consacré à la tolérance dans le cas d'un ordonnancement sans reprise. Dans le Chapitre 6, nous présentons les trois techniques de tolérance à la défaillance d'un cœur dans le cas d'un ordonnancement avec reprise. Dans le dernier chapitre de ce document (Chapitre 7), nous nous intéressons à la défaillance de plusieurs cœurs.

Le manuscrit se termine par une conclusion mettant particulièrement en avant les perspectives de travaux futurs faisant suite à ceux menés dans le cadre de cette thèse.

Première partie

Etat de l'art

Chapitre

2 L'ordonnancement équitable

Sommaire

Résumé

Ce chapitre présente l'algorithme d'ordonnancement qui sert de support à l'étude de la tolérance aux pannes des systèmes temps-réel. Il s'agit de l'algorithme PD2 qui appartient à la famille des algorithmes équitables. Les algorithmes de cette famille sont optimaux pour l'ordonnancement des systèmes multicoeurs composés de tâches indépendantes et périodiques, à départs simultanés et à échéances sur requête.

Introduction

En architecture multiprocesseur, trois principales politiques permettent d'organiser l'attribution des tâches aux processeurs : la politique partitionnée, la politique globale et la politique semi-partitionnée. Ces politiques s'appliquent également en architecture multicoeur, car le problème d'affectation des tâches aux processeurs (en architecture multiprocesseur) est assimilable au problème d'affectation des tâches aux coeurs (en architecture multicoeur). C'est pourquoi, tout au long de ce chapitre, nous parlerons indifféremment d'ordonnancement multiprocesseur ou d'ordonnancement multicoeur.

Dans la première section de ce chapitre, nous présentons les particularités de chaque politique. Peu de travaux sur la tolérance aux pannes se basent sur l'ordonnancement global, or il existe des algorithmes optimaux dans ce contexte. C'est pourquoi nous avons choisi d'axer nos travaux sur cette politique. C'est ainsi que nous choisissons l'algorithme équitable PD2 qui est optimal et dispose d'une condition nécessaire et suffisante d'ordonnançabilité facile à évaluer. La deuxième section de ce chapitre présente l'ordonnancement équitable et la troisième section l'algorithme PD2.

I. Les politiques d'ordonnancement multicoeur

La particularité d'un système temps réel dur est d'imposer que les tâches s'exécutent dans des limites temporelles précises. En architecture monoprocesseur, le problème d'ordonnancement consiste simplement à déterminer à chaque instant la tâche qui doit s'exécuter. Le cas multiprocesseur est plus compliqué du fait de la non parallélisation des tâches lors de l'exécution (une même tâche ne peut pas être exécutée à un instant donné par plusieurs processeurs à la fois). Ceci a pour conséquence le fait que les algorithmes optimaux dans le cas monoprocesseur peuvent perdre cette propriété dans le cas multiprocesseur, d'où le développement de nouvelles politiques. A cet effet, plusieurs approches ont été étudiées. La plus simple consiste à transformer le problème d'ordonnancement multiprocesseur sur m processeurs en m problèmes d'ordonnancement mono-processeur (partitionnement) couplés à un problème de répartition des tâches. Une seconde approche, dite globale, consiste à n'avoir qu'un seul ordonnanceur pour l'ensemble des processeurs. Les premières politiques globales proposées étaient des généralisations d'algorithmes monoprocesseurs, puis en 1996, *Baruah et al.* [42] ont introduit la notion d'équité (*fairness*) qui permet d'atteindre l'optimalité. Cependant, cette optimalité se fait à la faveur d'un grand nombre de préemptions et de migrations. Ce constat est à la base des algorithmes semi-partitionnés. Afin de limiter le nombre de migrations, l'idée est alors de partir d'un algorithme partitionné et de permettre à un nombre limité de tâches de migrer.

Toutes ces politiques d'ordonnancement multiprocesseur sont présentées en détail dans [9]. Par ailleurs, ces politiques sont incomparables et donc aucune n'est meilleure que les autres. Il est à noter que ces politiques ne gèrent pas le problème d'allocation des tâches élues sur les différents processeurs. Dans cette section, nous donnons une présentation générale en conservant l'hypothèse que l'architecture matérielle est composée de processeurs (ou cœurs) identiques. Les Figures utilisées sont tirées de [43].

I.1. La politique partitionnée

L'ordonnancement par partitionnement consiste à subdiviser l'ensemble des tâches du système à ordonnancer en créant autant de sous-ensembles qu'il y a de processeurs, chaque tâche étant affectée à un unique processeur. Ensuite, les tâches associées à un processeur sont ordonnancées selon un ordonnanceur monoprocesseur (voir Figure 2.1).

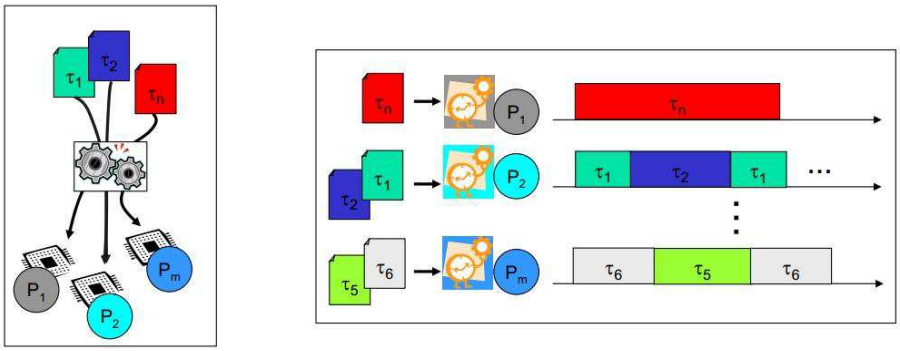

FIGURE 2.1 – Schématisation d'un ordonnancement partitionné

Généralement, le partitionnement est effectué hors-ligne, en se basant sur les conditions d'ordonnançabilité et les caractéristiques des tâches, en particulier leur facteur d'utilisation. Il existe deux façons d'aborder cette politique. Dans la première, l'ensemble des tâches du système est connu et on souhaite déterminer le nombre minimum de processeurs nécessaires pour pouvoir ordonnancer correctement le système. Dans ce cas, il s'agit d'un problème de minimisation. Dans le deuxième cas, le nombre de processeurs est fixé et l'on souhaite juste trouver un partitionnement tel que le système soit ordonnançable : c'est un problème de satisfaction de contraintes. Le partitionnement peut se faire suivant des méthodes exactes ou des heuristiques.

Les méthodes exactes assimilent le problème du partitionnement au problème de bin-packing (mise en sac) [44], très connu en recherche opérationnelle et optimisation combinatoire, qui consiste à ranger des paquets dans un nombre minimum de sacs. Pour l'ordonnancement partitionné, les tâches, caractérisées par leurs facteurs d'utilisation, représentent les paquets et les processeurs, caractérisés par leurs capacités, représentent les sacs. Quelques travaux [45] [46]

[47] se sont inspirés des résultats existants au problème de bin-packing pour proposer des algorithmes de partitionnement. Ces algorithmes permettent de réaliser un partitionnement exact lorsque le nombre de tâches du système est faible. Dès que le nombre de tâches dans le système devient trop grand, il n'est plus possible d'utiliser une méthode exacte. En effet, le nombre de partitions de n éléments en m sous-ensembles est tel qu'énumérer toutes les possibilités devient rapidement impossible, le problème de bin-packing étant reconnu NP-difficile. D'où le recours à des heuristiques.

Une heuristique est une méthode de calcul qui fournit rapidement (en temps polynomial) une solution réalisable, pas nécessairement optimale, pour un problème d'optimisation NP-difficile. Les heuristiques classiques applicables au partitionnement sont : First-Fit, Best-Fit, Next-Fit et Worst-Fit. Elles se décrivent comme suit :

- **First-Fit** : L'idée principale est, comme son nom l'indique, d'affecter chaque tâche au premier processeur trouvé tel que l'ordonnanceur local peut l'ordonnancer avec les tâches déjà affectées. Pour chaque tâche, on reprend la recherche depuis le début de la liste des processeurs.
- **Next-Fit** : L'algorithme est similaire au First-Fit à la différence près que l'on ne reprend pas la recherche depuis le début à chaque tâche. Lorsqu'une tâche est affectée à un processeur, la recherche de processeur pour la tâche suivante débute au processeur qui suit celui auquel on vient d'affecter une tâche.
- **Best-Fit** : L'algorithme choisit le processeur disposant de la plus petite capacité disponible et capable d'accepter la tâche tout en restant ordonnançable. En pratique, c'est équivalent au First-Fit mais en triant les processeurs par capacité.
- **Worst-Fit** : Identique au Best-Fit mais en triant les processeurs par ordre décroissant des capacités disponibles.

  Notons qu'il existe des variantes de ces heuristiques où les tâches sont d'abord triées par ordre décroissant selon leur facteur d'utilisation, on les nomme Decreasing First-fit, Decreasing Next-fit et Decreasing Best-fit. Selon [48] et [49], toutes ces heuristiques produisent en général des résultats plutôt corrects.

L'ordonnancement partitionné a pour avantage d'être facile à mettre en œuvre (plusieurs ordonnançeurs monoprocesseurs indépendants) et de pouvoir réutiliser les méthodes d'analyse d'ordonnançabilité monoprocesseur. De plus, il n'y a pas de coût système lié à la migration des tâches et les effets de surcharge sont limités au processeur concerné. En revanche, cette politique est moins souple, car adaptée uniquement aux configurations statiques, dans lesquelles les tâches sont dédiées à des processeurs. De plus, le partitionnement est reconnu NP-difficile.

I.2. La politique globale

L'ordonnancement global n'emploie qu'un seul ordonnanceur et une seule file de tâches (voir Figure 2.2). A chacune de ses invocations, l'ordonnanceur décide de l'affectation et de l'exécution d'au plus m tâches (où m désigne le nombre de processeurs).

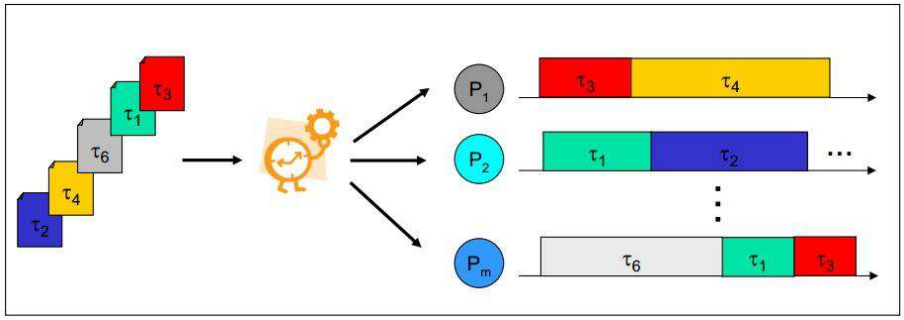

FIGURE 2.2 – Schématisation d'un ordonnancement global

Contrairement à la stratégie par partitionnement, les tâches ne sont pas destinées à un processeur en particulier. C'est cette différence majeure qui permet d'autoriser les migrations. Puisque l'ordonnancement global permet de supprimer la contrainte de non-migration, il semble logique que la classe de problèmes ordonnançables soit plus grande. Cependant, il a été montré [50] qu'appliquer RM ou EDF de manière globale, ne permet pas d'obtenir des conditions suffisantes plus avantageuses qu'un ordonnancement par partitionnement. C'est pour cette raison que de nouveaux algorithmes dédiés à l'ordonnancement multiprocesseur ont vu le jour, à savoir les algorithmes équitables qui sont optimaux pour les systèmes de tâches indépendantes, périodiques, à échéances sur requête. Cette famille d'algorithmes est présentée dans la section suivante.

La politique globale a l'avantage d'être adaptée aux configurations dynamiques et de disposer d'algorithmes optimaux. Par contre, les effets de surcharge ne concernent pas un seul processeur. De plus, elle engendre des coûts systèmes liés aux migrations des tâches et aux manipulations de la file unique de tâches prêtes.

I.3. La politique semi-partitionnée

Nous avons vu que la faiblesse de l'ordonnancement partitionné est l'absence de migration des tâches d'un processeur à un autre. En effet, certains systèmes non-ordonnançables le deviendraient si les migrations étaient autorisées. A l'opposé, l'ordonnancement global offre une totale liberté dans la migration des tâches, et c'est cela qui lui permet d'atteindre l'optimalité. Cependant, cette flexibilité peut avoir un coût à l'exécution dû à de nombreuses préemptions. Il est donc souhaitable de limiter le nombre de migrations et donc de préemptions [51].
Les algorithmes semi-partitionnés se situent entre ces deux extrêmes et proposent d'autoriser

une migration contrôlée de certaines tâches. Ils peuvent donc être vus comme une amélioration des algorithmes partitionnés en ajoutant plus de flexibilité mais ils s'inspirent aussi des politiques globales (voir figure 2.3).

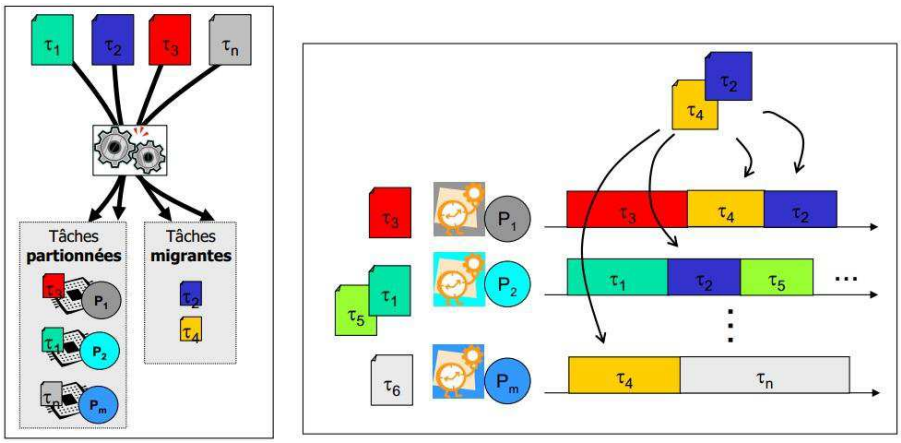

FIGURE 2.3 – Schématisation d'un ordonnancement semi-partitionné

Les principaux algorithmes semi-partitionnés ont été mis en œuvre dans LITMUS RT [52] et les résultats indiquent que les politiques semi-partitionnées offrent souvent de meilleurs choix [53] (par exemple, le compromis entre le nombre de préemptions et l'ordonnançabilité). Nous ne détaillons pas ces politiques ici, car elles ne rentrent pas dans le périmètre de notre étude.

II. Les algorithmes équitables

Les algorithmes de la famille PFair (Proportionate Fair) [42] sont des algorithmes globaux. Ces algorithmes, encore appelés algorithmes équitables, se différencient des algorithmes classiques d'ordonnancement en ce qu'ils imposent explicitement l'exécution des tâches à un taux régulier. L'objectif est de se rapprocher d'un ordonnancement idéal. Un ordonnancement est dit idéal, ou équitable, lorsque chaque tâche τ_i reçoit exactement $U_i \times t$ unités de temps processeur dans l'intervalle de temps $[0, t)$. Bien qu'un tel ordonnancement soit impossible dans le cas discret, un algorithme PFair va simplement s'en rapprocher. Pour ce faire, il est procédé à une discrétisation du temps et à une subdivision des tâches.

II.1. Discrétisation du temps

L'objectif d'un ordonnancement PFair est d'allouer les processeurs aux tâches de manière "idéale ou fluide" soit, proportionnellement dans le temps à leur facteur d'utilisation. Le temps est cependant discrétisé en intervalles uniformes appelés *slots* de temps. L'intervalle $[t, t+1)$

avec $t \in \mathbb{N}$ correspond au slot t. Pour modéliser une séquence d'ordonnancement, une fonction binaire $S : \tau_i \times \mathbb{N} \longmapsto \{0, 1\}$ est définie. Elle renvoie 1 lorsqu'une tâche τ_i est ordonnancée dans le slot t et 0 sinon. Grâce à cette fonction, on introduit la notion de décalage en temps qui mesure l'écart entre l'exécution idéale et la séquence construite. Le décalage d'une tâche τ_i à l'instant t se note :

$$lag(\tau_i, t) = U_i \times t - \sum_{l=0}^{t-1} S(\tau_i, l) \tag{2.1}$$

Un algorithme est dit PFair lorsqu'à tout instant le décalage entre l'exécution idéale ($U_i \times t$) et l'exécution réelle ($\sum_{l=0}^{t-1} S(\tau_i, l)$) est inférieur à 1 en valeur absolue. Ce qui se traduit de manière formelle par :

$$\forall t \in \mathbb{N}, \forall \tau_i \in \tau, |lag(\tau_i, t)| < 1.$$

Autrement dit, comme le montre la Figure 2.4 (*source [43]*), à chaque instant, l'erreur d'exécution en valeur absolue est strictement inférieure à un quantum de temps. Donc, l'exécution réelle approxime l'exécution idéale soit par sa partie entière inférieure, soit par sa partie entière supérieure.

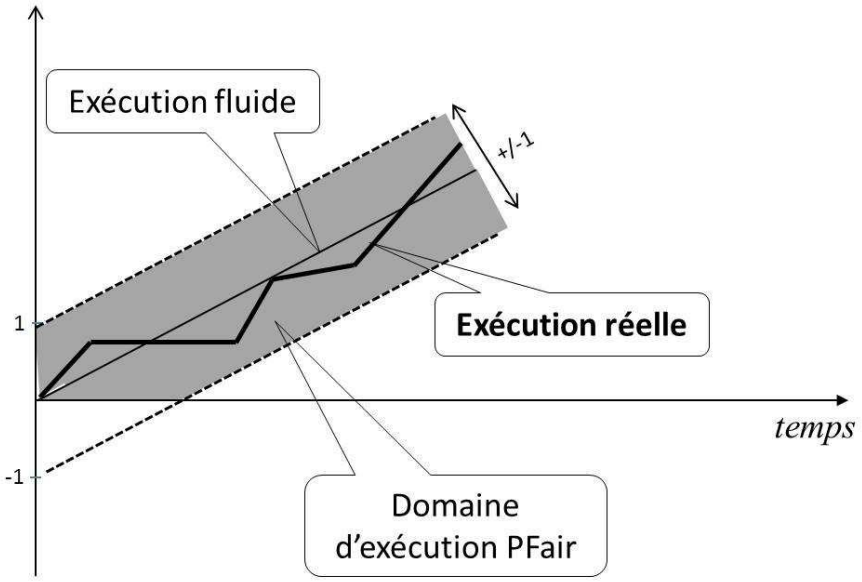

FIGURE 2.4 – Exemple d'exécution Pfair

II.2. Condition Optimalité

L'optimalité d'un ordonnancement PFair peut être obtenue pour un système de n tâches indépendantes et périodiques, à départs simultanés et à échéances sur requête sur m processeurs

identiques. La condition nécessaire et suffisante d'ordonnançabilité est définie par [42]. Un tel système est ordonnançable si et seulement si sa charge est inférieure au nombre de processeurs disponibles.

$$\sum_{i=1}^{n} U_i \leq m \tag{2.2}$$

Bien qu'en théorie un tel ordonnancement soit optimal, il convient de nuancer ce résultat en pratique. En effet, les surcoûts d'exécution tels que les prises de décision à chaque quantum, ou encore les nombreuses préemptions et possibles migrations, ne sont explicitement pas pris en compte. Pour en tenir compte, il faut augmenter le WCET. Et enfin, les processeurs doivent être parfaitement synchronisés entre eux pour respecter les précédences des morceaux de tâches (des travaux visent à adapter PFair pour tenir compte de ce problème de synchronisation [54]). Toutefois, cette condition servira de base à l'analyse d'ordonnançabilité tout le long de nos travaux.

II.3. Découpage de tâches en sous-tâches

La construction d'un ordonnancement PFair passe par la division de chaque tâche en sous-tâches de durée d'exécution unitaire. Une tâche τ_i est ainsi divisée en une séquence infinie de sous-tâches $\tau_i^j, j \geq 0$. Chaque sous-tâche possède une *pseudo-date d'activation* ou de réveil notée r_i^j et une *pseudo-échéance* notée d_i^j. L'intervalle $w(\tau_i^j) = [r_i^j, d_i^j)$ représente la *fenêtre d'exécution* de la sous-tâche τ_i^j. Sa longueur est définie par : $|w(\tau_i^j)| = d_i^j - r_i^j$. La pseudo-date de réveil et la pseudo-échéance peuvent être calculées par les formules suivantes :

$$r_i^j = \left\lfloor \frac{j}{U_i} \right\rfloor \text{ et } d_i^j = \left\lceil \frac{j+1}{U_i} \right\rceil \tag{2.3}$$

La figure 2.5 (*source [9]*) montre les fenêtres d'exécution des sous-tâches d'une tâche τ_i de paramètres $C_i = 7$ et de période $D_i = T_i = 13$.

Il est important de souligner que cette subdivision de la tâche en sous-tâches n'obéit pas à une logique fonctionnelle, mais temporelle. En d'autre terme, une sous-tâche ne correspond pas à une partie du code exécutable de la tâche, mais à une exécution durant une unité de temps de celle-ci. Ainsi, dire que la sous-tâche τ_i^j s'exécute dans la fenêtre $[r_i^j, d_i^j)$ revient à dire que la tâche τ_i passe de nouveau à l'état prêt à l'instant r_i^j et doit s'exécuter pendant une unité de temps avant l'instant d_i^j.
D'un point de vue du découpage des tâches en sous-tâches la notion d'équité se traduit par le fait que chaque sous-tâche du système s'exécute dans sa fenêtre. Il est dit valide si la dernière sous-tâche de chaque tâche s'exécute avant l'échéance de la tâche. Il en découle que l'équité entraîne la validité.

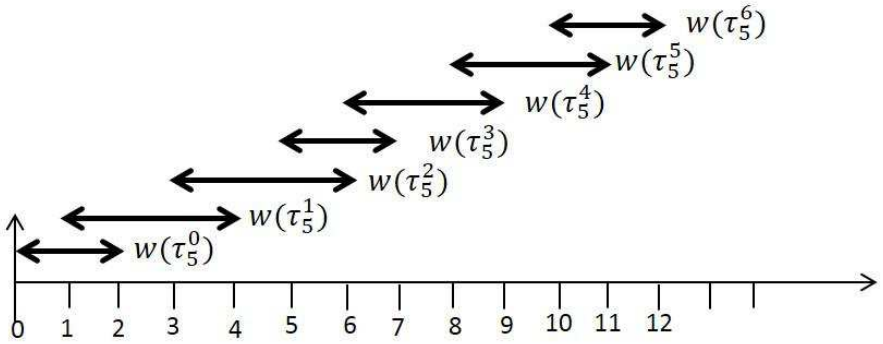

FIGURE 2.5 – Exemple de Fenêtres d'exécution

Suivant l'approche PFair, une sous-tâche τ_i^j est prioritaire par rapport à une sous-tâche τ_k^q si elle a une pseudo-échéance strictement inférieure à celle de cette dernière (c'est-à-dire si $d_i^j < d_k^q$).

II.4. Cas des tâches à échéances contraintes

Le problème de l'ordonnancement des tâches à échéances contraintes par un algorithme équitable a été abordé par Ramamurthy dans [55] pour des départs simultanés. Ces travaux ont été étendus par Malo [56] pour des tâches à départs différés. Il ressort de ces travaux que les algorithmes équitables ne sont pas optimaux pour ce type de tâche. Cependant, les travaux de Malo donnent une condition suffisante d'ordonnançabilité. Ainsi, l'auteur a montré que tout système de tâches périodiques indépendantes à échéances contraintes vérifiant la condition $\sum_{i=1}^{n} CH_i \leq m$ admet un ordonnancement équitable sur une architecture composée de m cœurs. Rappelons que $CH_i = \frac{C_i}{D_i}$ est le facteur de charge de la tâche τ_i. $k = \left\lfloor \frac{j}{C_i} \right\rfloor$ est le numéro d'instance de la sous-tâche τ_i^j. La fenêtre d'exécution d'une sous-tâche τ_i^j a pour paramètres :

$$r_i^{'j} = k \times T_i + \left\lfloor \frac{j - k \times C_i}{CH_i} \right\rfloor \text{ et } d_i^{'j} = k \times T_i + \left\lceil \frac{j + 1 - k \times C_i}{CH_i} \right\rceil \tag{2.4}$$

III. L'algorithme PD2

Il existe plusieurs algorithmes dans la famille PFair. On peut citer PF, PD [42] et PD2 [29]. Ils diffèrent l'un de l'autre par la manière dont ils déterminent la priorité entre les sous-tâches lorsque leurs pseudo-échéances sont égales. Dans le cadre de nos travaux nous avons choisi l'algorithme PD2 qui est reconnu comme étant le plus efficace en raison de sa faible complexité pour les prises de décision.

III.1. Principe

Lorsque les pseudo-échéances sont égales, l'algorithme PD2 utilise deux paramètres supplémentaires pour déterminer la priorité entre sous-tâches : le *bit successeur* et *l'échéance de groupe*. Le deuxième paramètre concerne les tâches lourdes. Une tâche est qualifiée de *lourde* si et seulement si son facteur d'utilisation est supérieur ou égal à 1/2, sinon il s'agit d'une tâche *légère*. Il est intéressant de remarquer alors que seule une tâche lourde possède des fenêtres d'exécution de taille 2 [29].

Définition 1. *Le bit Successeur*
Si τ_i^j et τ_i^{j+1} sont deux sous-tâches consécutives d'une tâche τ_i, la fonction bit successeur b_i^j : $\tau_i^j \longmapsto 0,1$ se définit par

$$b_i^j = \begin{cases} 1 & si\ r_i^{j+1} = d_i^j - 1 \\ 0 & sinon \end{cases}$$

Autrement dit, $b_i^j = 1$ si les fenêtres d'exécution successives des sous-tâches τ_i^j et τ_i^{j+1} se chevauchent précisément d'un slot. S'il n'y a pas de chevauchement, $b_i^j = 0$.

Définition 2. *L'échéance de groupe*
Soit $\tau_i^j, \tau_i^{j+1}, ..., \tau_i^l$ une séquence des sous-tâches consécutives d'une tâche lourde τ_i telle que les deux conditions suivantes sont réunies :
(1) $\forall k \in]j, l], |w(\tau_i^k)| = 2$ et $b_i^{k-1} = 1$, c'est-à-dire que les fenêtres d'exécution sont de taille 2 et que la fenêtre de chaque sous-tâche de la séquence chevauche celle de la sous-tâche suivante ;
(2) soit $|w(\tau_i^{l+1})| = 3$, soit $b_i^l = 0$, c'est-à-dire que la sous-tâche qui suit la dernière sous-tâche de la séquence a une fenêtre de taille 3 ou il n' y a pas chevauchement entre la fenêtre de cette dernière sous-tâche et celle de la suivante. Les sous-tâches d'une telle liste ont cette particularité que si l'une d'elle s'exécute dans le dernier slot de sa fenêtre, alors toutes les sous-tâches qui la suivent n'auront pas d'autre choix que de s'exécuter dans le dernier slot de leurs fenêtres respectives. Ces sous-tâches partagent donc une sorte d'échéance de groupe. Formellement, l'échéance de groupe d'une suite de sous-tâches $\tau_i^j, ..., \tau_i^l$ peut être calculée pour chaque sous-tâche par la formule suivante :

$$D_i^j = \begin{cases} d_i^l + 1 & si\ |w_i^{l+1}| = 3 \\ d_i^l & si\ b_i^l = 0 \end{cases}$$

Les sous-tâches d'une tâche légère ont une échéance de groupe égale à 0.

A l'aide de ces deux paramètres supplémentaires, l'algorithme PD2 détermine la priorité entre sous-tâche de la manière suivante : une sous-tâche τ_i^j est prioritaire par rapport à une tâche τ_k^q si l'une des conditions suivantes est vérifiée :

1. $d_i^j < d_k^q$
2. $d_i^j = d_k^q \wedge b_i^j > b_k^q$
3. $d_i^j = d_k^q \wedge b_i^j = b_k^q = 1 \wedge D_i^j > D_k^q$
4. $d_i^j = d_k^q \wedge b_i^j = b_k^q = 1 \wedge D_i^j = D_k^q \wedge i < k$

La dernière condition assure le déterminisme de l'algorithme.

III.2. Description formelle

Nous n'avons rencontré aucune description formelle de l'algorithme PD2 dans la littérature. Cependant, en nous basant sur la présentation ci-dessus, nous proposons la description suivante qui est implémentée dans notre prototype d'expérimentation. Eu égard à la cyclicité de l'ordonnancement du système, l'algorithme suivant qui est proposé sur la première hyperpériode est itéré sur les hyperpériodes suivantes. Nous commençons par présenter les notations et primitives utilisées. **NOTATIONS**

- $L(i)$: i^{eme} élément de la liste L ;
- $Seq_{[k]}^{(S)}$: séquence d'ordonnancement du système S sur k coeurs ;
- $Seq_{[k]}^{(S)}(C_k, t)$: sous-tâche ordonnancée sur le coeur C_k à l'instant t dans $Seq_{[k]}^{(S)}$.

PRIMITIVES

On suppose l'existence des primitives suivantes :

- $sousTaches(S)$: retourne la liste des sous-tâches du système S sur une hyperpériode ;
- $pretes(ST, t)$: retourne la liste des sous-tâches de ST prêtes à l'instant t ;
- $card(L)$: retourne le nombre d'éléments de la liste L ;
- $ranger(L)$: range les sous-tâches de la liste L par ordre de priorité PD2 décroissante ;
- $init(Seq)$: initialise la séquence Seq par des unités de temps creux.

III.3. Exemple

Soit à ordonnancer avec PD2 le système de tâches suivant :
$\Psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$.
La charge est $U = 2.16$ et donc le système est ordonnançable sur 3 cœurs. L'hyperpériode est $H = ppcm(3, 4, 6, 12) = 12$. τ_3 et τ_5 sont des tâches lourdes car leurs facteurs d'utilisation sont supérieurs ou égaux à 0.5.

Algorithm 1: Ordonnancement d'un système avec PD2.

Inputs : $S : \tau_i < C_i, T_i >$: système à ordonnancer ;
$C = (C_1, C_2, .., C_m)$: Liste des cœurs du processeur ;

Output: $Seq = Seq_{[m]}^{(S)}$: Séquence d'ordonnancement du système S sur m cœurs durant l'intervalle de temps $[0, H)$.

```
init(Seq);
t ← 0 ;
ST ← sousTaches(S);
while t < H do // i.e à tout instant t
    P ← pretes(ST, t) ;
    ranger(P);
    k ← 1 ;
    j ← 1 ;
    while (k ≤ m) ∧ (j ≤ card(P)) do
        Seq(C(k), t) ← P(j) ;
        j ← j+1 ;
        k ← k+1 ;
    t ← t+1 ;
```

Découpage des tâches en sous-tâches

Le Tableau 2.1 présente les paramètres des sous-tâches de Ψ calculés avec les formules précédentes. Les tâches étant périodiques, nous limitons l'étude à l'hyperpériode, date à laquelle l'ordonnancement se reproduit à l'identique.

Affectation des sous-tâches aux processeurs

Le listing du Tableau 2.2 montre le déroulement de l'algorithme PD2 pour l'ordonnancement du système S sur trois cœurs. La première colonne définit l'instant de l'ordonnancement et la deuxième, la liste des sous-tâches prêtes à cet instant, rangées par ordre de priorité décroissant. La troisième colonne contient les sous-tâches élues à chaque instant et la dernière colonne indique pour chaque sous-tâche élue, le cœur où elle s'exécute. L'heuristique d'affectation des sous-tâches est l'ordre croissant de numérotation des cœurs. Notons que d'autres heuristiques permettant de réduire le nombre de migrations peuvent être utilisées. Nous ne les discuterons pas ici.

Séquence générée

La Figure 2.6 présente l'ordonnancement du système S par l'algorithme PD2. Pour chaque cœur et à chaque instant on peut observer la sous-tâche en cours d'exécution.

TABLE 2.1 – Paramètres des sous-tâches des premières instances des tâches du système Ψ

τ_i	τ_i^j	r_i^j	d_i^j	b_i^j	D_i^j
τ_1	τ_1^0	0	3	0	0
	τ_1^1	3	6	0	0
	τ_1^2	6	9	0	0
	τ_1^3	9	12	0	0
τ_2	τ_2^0	0	3	0	0
	τ_2^1	3	6	0	0
	τ_2^2	6	9	0	0
	τ_2^3	9	12	0	0
τ_3	τ_3^0	0	2	0	2
	τ_3^1	2	4	0	4
	τ_3^2	4	6	0	6
	τ_3^3	6	8	0	8
	τ_3^4	8	10	0	10
	τ_3^5	10	12	0	12
τ_4	τ_4^0	0	3	1	0
	τ_4^1	2	5	1	0
	τ_4^2	4	8	1	0
	τ_4^3	7	10	1	0
	τ_4^4	9	12	0	0
τ_5	τ_5^0	0	2	1	3
	τ_5^1	1	4	1	5
	τ_5^2	3	6	1	8
	τ_5^3	5	7	1	8
	τ_5^4	6	9	1	10
	τ_5^5	8	11	1	12
	τ_5^6	10	12	0	12

TABLE 2.2 – Déroulement de PD2 sur le système Ψ durant la première hyper-période

Instant	**s/tâches prêtes**					**s/tâches élues**
	τ_i^j	r_i^j	d_i^j	b_i^j	D_i^j	
t=0	τ_5^0	0	2	1	3	$\tau_5^0 \longmapsto C_1$
	τ_3^0	0	2	0	2	$\tau_3^0 \longmapsto C_2$
	τ_4^0	0	3	1	0	$\tau_4^0 \longmapsto C_3$
	τ_1^0	0	3	0	0	
	τ_2^0	0	3	0	0	
t=1	τ_1^0	0	3	0	0	$\tau_1^0 \longmapsto C_1$
	τ_2^0	0	3	0	0	$\tau_2^0 \longmapsto C_2$
	τ_5^1	1	4	1	5	$\tau_5^1 \longmapsto C_3$
t=2	τ_3^1	2	4	0	4	$\tau_3^1 \longmapsto C_1$
	τ_4^1	2	5	1	0	$\tau_4^1 \longmapsto C_2$
t=3	τ_5^2	3	6	1	8	$\tau_5^2 \longmapsto C_1$
	τ_1^1	3	6	0	0	$\tau_1^1 \longmapsto C_2$
	τ_2^1	3	6	0	0	$\tau_2^1 \longmapsto C_3$
t=4	τ_3^2	4	6	0	6	$\tau_3^2 \longmapsto C_1$
	τ_4^2	4	8	1	0	$\tau_4^2 \longmapsto C_2$
t=5	τ_5^3	5	7	1	8	$\tau_5^3 \longmapsto C_1$
t=6	τ_3^3	6	8	0	8	$\tau_3^3 \longmapsto C_1$
	τ_5^4	6	9	1	10	$\tau_5^4 \longmapsto C_2$
	τ_1^2	6	9	0	0	$\tau_1^2 \longmapsto C_3$
	τ_2^2	6	9	0	0	
t=7	τ_2^2	6	9	0	0	$\tau_2^2 \longmapsto C_1$
	τ_4^3	7	10	1	0	$\tau_4^3 \longmapsto C_2$
t=8	τ_3^4	8	10	0	10	$\tau_3^4 \longmapsto C_1$
	τ_5^5	8	11	1	13	$\tau_5^5 \longmapsto C_2$
t=9	τ_1^3	9	12	0	0	$\tau_1^3 \longmapsto C_1$
	τ_2^3	9	12	0	0	$\tau_2^3 \longmapsto C_2$
	τ_4^4	9	12	0	0	$\tau_4^4 \longmapsto C_3$
t=10	τ_3^5	10	12	0	12	$\tau_3^5 \longmapsto C_1$
	τ_5^6	10	12	0	13	$\tau_5^6 \longmapsto C_2$
t=11	-	-	-	-	-	-

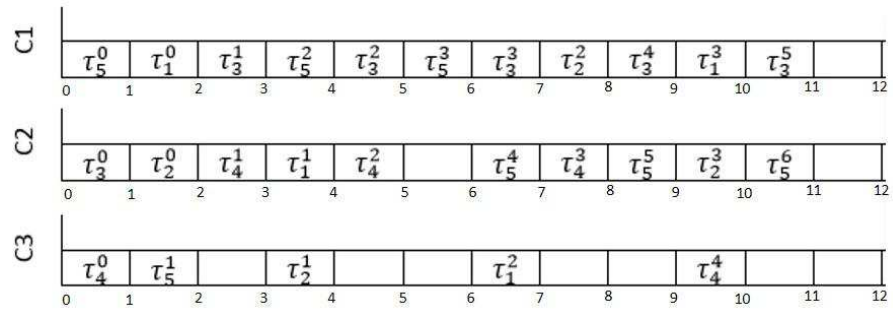

FIGURE 2.6 – Ordonnancement par PD2 du système S sur 3 cœurs

Conclusion

Dans ce chapitre, nous avons présenté les trois approches classiques d'ordonnancement d'un système temps-réel en architecture multicoeur : l'approche partitionnée, l'approche globale et l'approche semi-partitionnée. Aucune de ces approches n'est meilleure que l'autre, chacune ayant sa spécificité. Dans le cadre de cette thèse, nous avons choisi de nous intéresser à l'ordonnancement global par les algorithmes équitables, lesquels sont optimaux dans notre contexte. De ces algorithmes, nous retenons l'algorithme PD2 dans cette étude.

La validité d'un algorithme équitable peut être compromise dès lors qu'à un instant donné, à cause d'une panne, le nombre de cœurs restant ne garantit plus la satisfaction de la condition nécessaire et suffisante d'ordonnançabilité. C'est pourquoi des techniques de tolérance doivent être mises en œuvre pour répondre à une telle éventualité. Dans le chapitre suivant, nous faisons un état de l'art sur les techniques de tolérance existantes.

Chapitre

3 Tolérance aux pannes des systèmes temps-réel

Sommaire

Résumé

Ce chapitre présente la notion de panne et les mécanismes de tolérance. Il fait un état de l'art des techniques de construction d'un ordonnancement valide d'un système temps-réel en dépit de la survenue d'une panne. L'analyse de ces techniques permet d'en déceler les limites et de ressortir les particularités des techniques proposées dans cette thèse.

Introduction

Comme nous l'avons précisé dans l'introduction générale, l'exactitude des résultats d'un système temps réel ne dépend pas seulement de l'exactitude algorithmique des calculs mais aussi de la date à laquelle le résultat est produit. Cependant, au cours de son exécution, plusieurs changements peuvent se produire dans son environnement d'exécution. Ces changements peuvent porter aussi bien sur l'architecture matérielle (état de la batterie, défaillance d'un composant, etc.) que sur la structure logicielle (prise en compte de nouvelles exigences, mises à jour des fonctionnalités). Étant donné qu'ils ont une incidence sur le respect des contraintes temporelles et la qualité des résultats produits, l'adaptation du système temps-réel aux dits changements constitue un défi supplémentaire pour les concepteurs [57]. C'est particulièrement le cas pour les systèmes critiques (conduite automatique de trains, de centrales nucléaires, de barrages, automobile, avionique) dont le domaine d'utilisation peut être particulièrement dangereux, mettant en jeu des vies humaines avec des coûts liés aux pannes qui peuvent être immenses. C'est pourquoi, la sûreté de fonctionnement de tels systèmes (*Safety critical systems*) est une exigence de conception qui permet d'assurer leur adaptation aux modifications environnementales, notamment à la survenue de pannes.

La sûreté de fonctionnement (*dependability*) désigne l'aptitude d'une entité à satisfaire à une ou plusieurs fonctions requises dans des conditions données [58]. D'après [59], c'est la propriété qui permet de placer une confiance justifiée dans le service que délivre un système informatique. Comme le montre la Figure 3.1, cette propriété s'évalue en fonction de différents attributs qui sont : la disponibilité (être prêt à l'utilisation), la fiabilité (assurer la continuité du service), la sécurité (absence de conséquences catastrophiques pour l'environnement d'exécution), l'intégrité (non altération des données et des fonctionnalités), la survivabilité (capacité à résister à la défaillance d'une partie du système) et la maintenabilité (aptitude aux réparations et aux évolutions). Les entraves à la sûreté de fonctionnement sont les fautes, les erreurs et les défaillances. Pour y faire face, des moyens logiciels ou matériels sont mis en place pour prévenir, tolérer, éliminer ou même prévoir les défaillances.

Tel que stipulé dans [60], la prévention ou évitement consiste à empêcher (par construction) l'occurrence ou l'introduction de fautes. L'élimination consiste à réduire (par vérification) la présence de fautes. Quant à la prévision, il s'agit d'estimer (par évaluation), la présence et les conséquences des fautes. **La présente étude porte particulièrement sur la tolérance, c'est-à-dire, comment délivrer un service conforme à la spécification du système.**

La suite de ce chapitre s'organise de la façon suivante : dans un premier temps, une présentation générale des notions de panne et de tolérance est proposée (section 1) ; elle débouche sur la circonscription de notre périmètre d'étude. Ensuite, un état de l'art sur les techniques de tolérance aux pannes en architecture multiprocesseur ou multicoeurs est proposé (section 2).

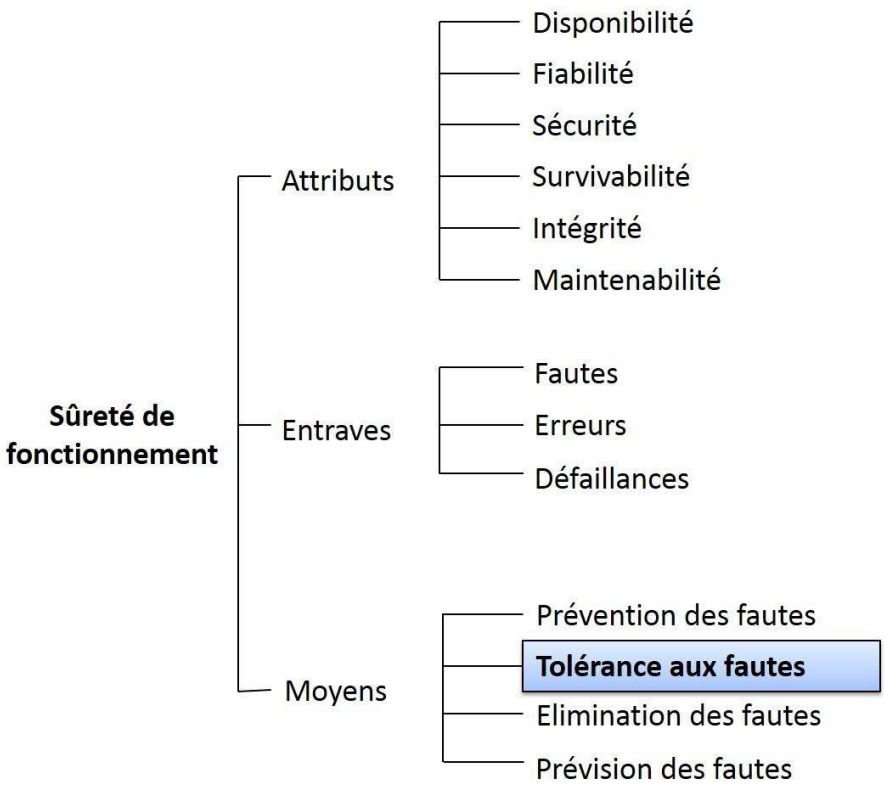

FIGURE 3.1 – Sûreté de fonctionnement des systèmes informatiques

Enfin, l'analyse des travaux présentés (section 3) permet de montrer leurs limites d'application à notre périmètre d'étude et de ressortir la particularité de nos travaux.

I. Généralités sur les pannes

I.1. Définition et typologie des pannes

I.1.a. La notion de panne

Plusieurs termes sont communément employés (par abus de langage) pour désigner une panne. Ce sont les termes *faute*, *erreur* et *défaillance*. Ces mots qui semblent présenter des similitudes sémantiques désignent des éléments bien distincts mais corrélés. Dans ce paragraphe, nous précisons la particularité de chaque terme et montrons la corrélation qui existe entre eux. Cela nous conduit à proposer une définition de la notion de panne. Les définitions ci-dessous sont tirées des travaux de [61].

— *Erreur* : une erreur est un état du système à partir duquel la poursuite de l'exécution

est susceptible de conduire à une défaillance.

— *Faute* : une faute est un évènement ayant entrainé une erreur. Il peut s'agir d'une faute de programmation (pour les erreurs logicielles) ou d'évènements physiques (usure, malveillance, catastrophe, ...) dans le cas d'erreurs matérielles. Dans le cadre de cette thèse, nous ne considérerons que des fautes matérielles (coupure d'un lien, perturbation électromagnétique, décharge électrostatique. . .).

— *Défaillance* : on parle de défaillance d'un composant (ou d'un système) lorsque son comportement n'est plus conforme à sa spécification.

La corrélation qui existe entre ces termes est qu'une faute peut ne pas gêner le fonctionnement du système (on parle de faute passive) ou le gêner en conduisant à un état erroné (faute active). Cet état peut rester indétecté longtemps (on parle de latence de la faute) mais conduit à moyen ou à long terme à une défaillance ou panne. Des trois termes ci-dessus définis, c'est le terme défaillance qui traduit la notion de panne. On dira donc qu'un système est en *panne* si suite à une faute, il se trouve dans un état erroné qui le rend inapte à respecter l'une de ses spécifications. Une panne est donc la manifestation au niveau du service rendu d'une faute.

I.1.b. Typologie des pannes

Plusieurs critères peuvent être utilisés pour classifier les pannes. Dans son ouvrage [62] utilise comme critères la durée et le comportement résultant. A ces deux critères, [63] ajoute la cause. Dans ce paragraphe, nous proposons une classification (voir Figure 3.2) qui complète les critères précédents en y ajoutant la nature et le niveau de gravité de la panne.

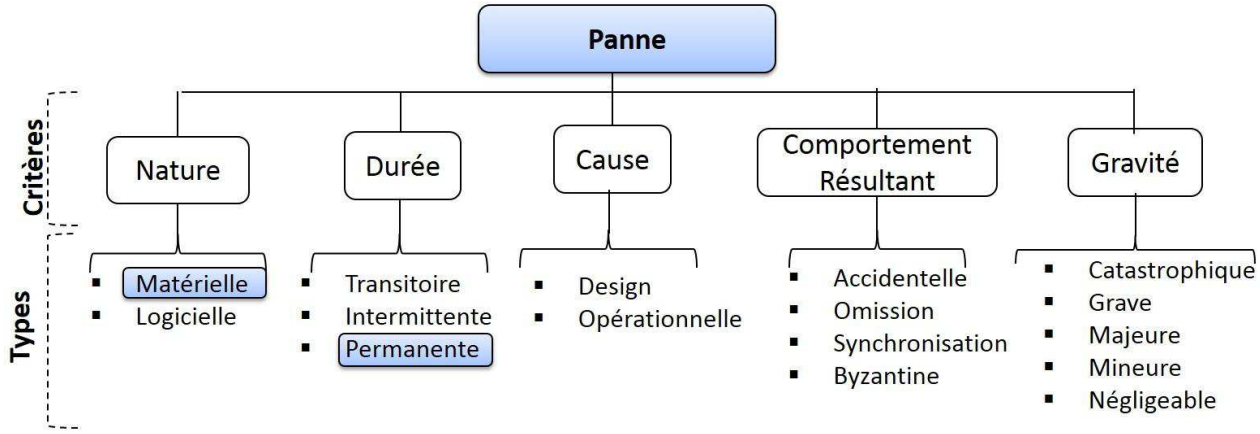

FIGURE 3.2 – Typologie de pannes

— **Selon la nature**

⊙ *Panne matérielle* : résulte de la défaillance d'une ressource matérielle ;

⊙ *Panne logicielle* : résulte d'un état erroné du système.

— **Selon la durée**

⊙ *Panne transitoire* : conséquence d'un impact environnemental temporaire, elle disparaît sans aucune intervention ;

⊙ *Panne intermittente* : variante de la panne transitoire, elle se produit occasionnellement et de façon imprévisible. Elle est généralement due à l'instabilité de certaines caractéristiques matérielles ou à l'exécution du programme dans un espace particulier de l'environnement ;

⊙ *Panne Permanente* : continue et stable dans le temps, la panne persiste tant qu'il n'y a pas d'intervention externe pour l'éliminer. Un changement physique dans un composant provoque une panne matérielle permanente.

— **Selon la cause**

⊙ *Panne de design* : due à une mauvaise spécification ou à une mauvaise conception du système. En pratique, ce genre de panne ne devrait pas exister grâce aux dispositifs de validation, de tests et de simulations ;

⊙ *Panne opérationnelle* : qui se produit durant le fonctionnement du système. Elle est généralement due à des causes physiques.

— **Selon le comportement résultant**

Après l'occurrence d'une panne, on distingue quatre différents comportements possibles du composant concerné :

⊙ *Panne accidentelle (Crash)* : le composant soit s'arrête complètement de fonctionner soit continue mais sans retourner à un état stable (valide) ;

⊙ *Panne d'omission* : le composant n'est plus capable d'améliorer son service (échec total) ;

⊙ *Panne de synchronisation (Timing)* : le composant effectue son traitement mais fournit le résultat en retard ;

⊙ *Panne Byzantine* : cette panne est de nature arbitraire ; le comportement du composant est donc imprévisible. Souvent causé par des attaques malicieuses, ce type de panne est considéré comme le plus difficile à gérer.

— **Selon le niveau de gravité**

⊙ *Panne Catastrophique* : risque de pertes de vies humaines ou d'effets sur l'environnement à long terme ;

⊙ *Panne Grave* : destruction importante de biens qui interrompent les activités sur une longue période. Effets sur l'environnement à court terme ;

⊙ *Panne Majeure* : perte de mission ou endommagement d'un bien ;

⊙ *Panne Mineure* : mission dégradée ;

⊙ *Panne négligeable* : sans effet sensible sur le déroulement de la mission.

I.2. Tolérance aux pannes

La tolérance aux pannes est la capacité d'un système à rendre ses services conformément à ses spécifications fonctionnelles et temporelles en dépit de la survenance d'une panne. D'après [64], la gestion d'une panne dans un système informatique se fait en deux grandes étapes : la détection des erreurs et le rétablissement du système. Les techniques classiques utilisées dans chaque étape sont présentées dans [60]. Elles ne seront pas détaillées dans ce chapitre. Cependant, dans ce paragraphe, nous précisons le rôle et le déroulement de chaque étape, ce qui nous permettra de circonscrire notre périmètre d'étude. Ces étapes sont illustrées par la Figure 3.3.

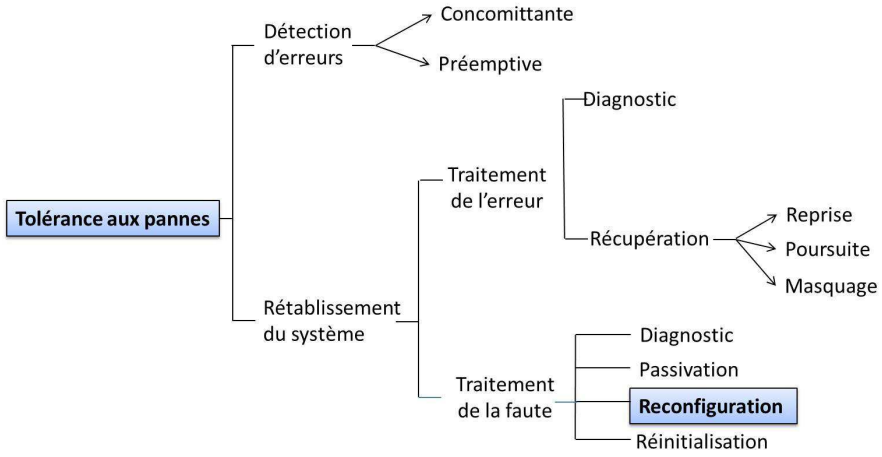

FIGURE 3.3 – Étapes de la tolérance aux pannes

I.2.a. La détection d'erreurs

Il s'agit dans cette étape d'identifier la présence des erreurs dans le système. A cet effet, on distingue la détection *concomitante* qui est réalisée lors de l'exécution normale du système et la détection *préemptive* qui est effectuée par une suspension de l'exécution du système afin de rechercher l'éventuelle présence d'erreurs latentes ou de fautes passives. Dans cette thèse, nous supposons qu'un système de détection de la panne existe et qu'il est indépendant du système ordonnancé.

I.2.b. Le rétablissement du système

Le rétablissement passe par le **traitement de l'erreur** (*Error processing*) qui consiste à éliminer les erreurs des états du système et le **traitement de la faute** (*fault treatment*) qui consiste à éviter que la faute ne se reproduise.
Le traitement des erreurs passe par une phase de **diagnostic** qui consiste à évaluer l'étendue

des dommages et une phase de **récupération** qui consiste à restaurer l'état valide du système. La récupération peut se faire par **reprise** (*backward recovery* : le système retourne au dernier état non erroné sauvegardé), par **poursuite** (*forward recovery* : un nouvel état exempt d'erreur est trouvé) ou par **compensation** (l'état erroné comporte suffisamment de redondance pour permettre de masquer l'erreur). Aujourd'hui, les microprocesseurs modernes sont dotés de dispositifs de traitement d'erreurs au niveau de la mémoire, des caches, des registres etc. [65] [66] [67]
Le traitement de la faute quant à lui passe par les quatre phases suivantes :

- La phase de **diagnostic** qui identifie et enregistre la ou les causes de l'erreur détectée en termes de localisation et de type ;
- La phase de **passivation** qui assure l'exclusion logique et physique du ou des composants fautifs du processus d'exécution ultérieure. En d'autres termes, il s'agit de rendre la faute passive ;
- La phase de **reconfiguration** qui réaffecte les tâches parmi les composants non défaillants ou intègre des composants en réserve ;
- La phase de **réinitialisation** qui contrôle, met à jour et enregistre la nouvelle configuration, puis actualise les tables et enregistrements du système.

I.3. Délimitation de l'étude

I.3.a. Type de panne considéré

De nos jours, les mécanismes rigoureux de vérification, de tests, de validation et de normalisation ont permis de réduire considérablement, voire d'éradiquer les pannes logicielles dans les systèmes temps-réel critiques. L'intérêt est désormais porté sur les pannes matérielles qui sont imprévisibles malgré l'évolution technologique des composants. Parmi les pannes matérielles, les pannes transitoires et intermittentes sont les plus récurrentes [68] et ont fait l'objet de nombreux travaux. Peu de travaux portent sur les pannes permanentes qui, bien que rares, restent susceptibles de survenir avec des conséquences désastreuses sur la quantité de ressources disponibles. C'est pour cette raison que nous focalisons notre étude sur ce type de panne.
Tout au long de ce document, par le mot *panne*, nous désignons la mise hors service d'un ou de plusieurs cœurs du processeur où s'exécutent les tâches du système.

I.3.b. Phase de tolérance impliquée

Compte tenu de ce qui a été exposé dans le paragraphe précédent, notre étude se situe à l'étape de rétablissement du système, plus précisément à la phase de reconfiguration. Nous supposons donc que l'étape de traitement de l'erreur est passée et que c'est l'étape du traitement de la faute qui est en cours. Dans cette étape, les phases de diagnostic et de passivation de la faute sont achevées. Au cours de cette phase de réconfiguration, l'objectif est la poursuite de l'exécution des tâches sur les coeurs fonctionnels, dans le respect des spécifications temporelles

et fonctionnelles, avec une gestion optimale des ressources et du temps d'exécution. Rappelons que notre étude s'appuie sur une politique d'ordonnancement globale avec pour algorithme PD2. La Figure 3.4 circonscrit notre périmètre d'étude.

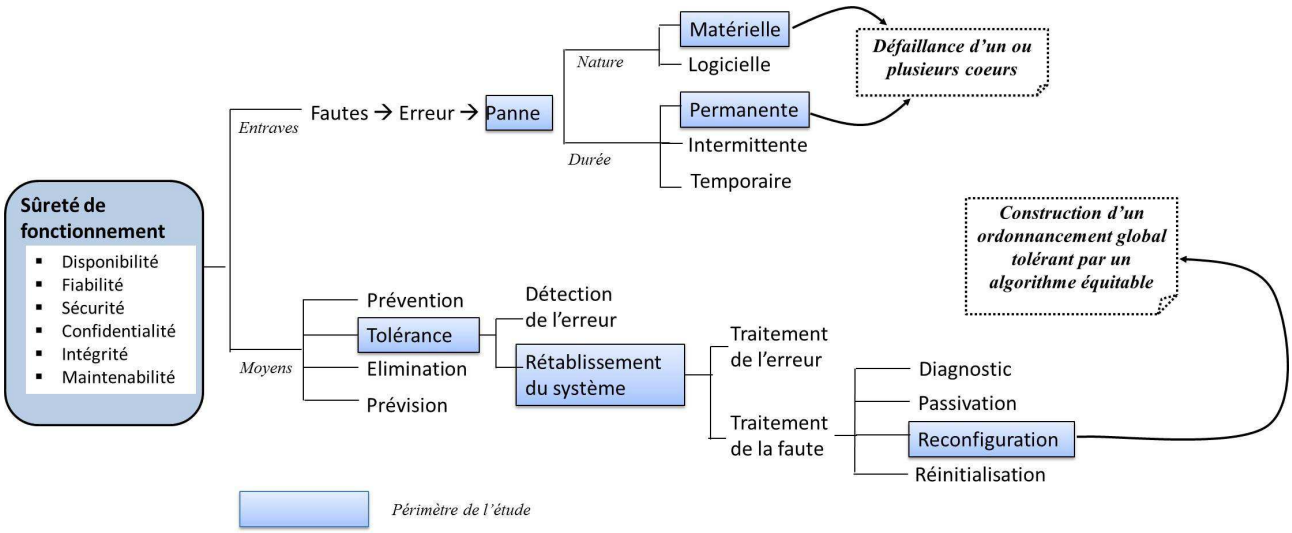

FIGURE 3.4 – Périmètre d'étude

I.4. Utilisation de la redondance

La tolérance aux pannes des systèmes temps-réel repose en général sur l'utilisation de données ou de matériels redondants [69]. L'idée est d'introduire dans le système des copies redondantes de certains éléments aux fins de les utiliser en cas de panne. C'est ainsi que les techniques de tolérance développées reposent essentiellement sur la redondance matérielle, la redondance logicielle et la redondance temporelle qui sont présentées dans les paragraphes qui suivent.

I.4.a. Redondance matérielle

Dans ce type de redondance, un groupe d'unités de traitement (cœurs ou processeurs) redondants est conçu pour tolérer la panne de certains de ses membres. En dépit de la panne de certains membres du groupe, le service offert du point de vue global par le groupe continue en masquant la panne. Les membres du groupe clairement détectés en panne sont isolés, retirés du groupe et les autres membres prennent le relai. Le groupe est ensuite reconfiguré jusqu'à ce que trouver une reconfiguration qui maintient un service acceptable devienne impossible.
On distingue la redondance matérielle passive et la redondance matérielle active [70].

— **redondance matérielle passive**
Une seule unité de traitement (unité primaire) réalise effectivement les traitements et est affectée aux sorties. En cas de panne de l'unité primaire, l'une des unités redondantes

inactives (unités secondaires) est sélectionnée et activée pour prendre en charge le service. La redondance passive pose un problème de synchronisation entre l'unité primaire et l'unité secondaire. En effet, il se pose les problèmes de la détection de la panne de l'unité primaire, de la détermination d'une nouvelle unité primaire parmi les unités secondaires et de la reconstitution d'un contexte d'exécution correct pour cette nouvelle unité primaire. Une solution à ces problèmes consiste à sauvegarder périodiquement les contextes d'exécution des unités primaires.

— **redondance matérielle active**
Toutes les unités (primaire et secondaires) réalisent les traitements. Dans certains cas, une seule unité (primaire) est affectée aux sorties. Dans d'autres cas, un gestionnaire de redondance détermine le résultat à retourner parmi les résultats fournis par toutes les unités (primaire et secondaires). En cas de panne de l'unité primaire, l'unité secondaire choisie prend le contrôle et ce, avec le contexte d'exécution complet de l'activité. Un exemple de redondance passive est la technique "*Triple Modular Redundancy (TMD)*" [71] dans laquelle trois unités de traitement exécutent des copies redondantes du même programme et masquent les erreurs d'exécution par un vote sur les résultats retournés.

I.4.b. Redondance logicielle

Tout comme la redondance matérielle, la redondance logicielle [72] consiste à prévoir plusieurs copies du même logiciel dont une seule (la copie primaire) est exécutée à la fois. Lorsqu'une panne survient, une des copies secondaires prend le relais. Si on veut appliquer le principe de la redondance aux logiciels, on ne va évidemment pas utiliser des copies identiques du même logiciel, puisque leur comportement serait identique. En effet, le code étant le même, la même panne logicielle surviendra au même instant dans les différentes copies. Il faut donc faire développer des programmes ayant les mêmes spécifications par des équipes différentes. Dans ce cas, si on fait fonctionner deux copies du logiciel en même temps avec les mêmes données d'entrée, on peut espérer que quand l'une aura une défaillance, l'autre fonctionnera correctement. Mais l'expérience montre que ce n'est pas aussi simple : les deux équipes peuvent buter sur les mêmes difficultés et avoir tendance à faire des fautes aux mêmes endroits. Toutefois, il reste que la redondance logicielle augmente la fiabilité des logiciels. C'est une méthode souvent utilisée. On peut à cet effet citer l'exemple de Airbus qui utilise la redondance logicielle sur la chaîne de commande et de surveillance des avions. Toutefois, ce type de redondance engendre des surcoûts liés à la multiplication des équipes de développement.

I.4.c. Redondance temporelle

La redondance temporelle permet d'obtenir la tolérance aux pannes en exécutant plusieurs fois une opération. Un exemple est la retransmission des paquets TCP/IP dans un réseau informatique. Cette forme de redondance est utile pour des pannes transitoires ou intermittentes, mais pas vraiment pour des pannes permanentes. Certains travaux basés sur cette forme de

redondance ont été proposés en ordonnancement temps-réel [73] [74] [75]. Ils ont en commun l'implémentation de la tolérance par la duplication du temps d'exécution des tâches. Une même tâche est attribuée à une paire d'unités de traitement (homogènes) et les résultats d'exécution des deux unités sont comparés. S'ils sont identiques (ou bien similaires), le résultat est accepté et retourné à l'utilisateur. Dans le cas contraire, le résultat est rejeté et la tâche est assignée à une autre paire. A chaque activation de tâche, l'algorithme d'ordonnancement recherche une paire d'unités de traitement dont les composantes retournent des résultats identiques ou compatibles. La tolérance est assurée en ce sens que lorsqu'une unité de traitement tombe temporairement ou définitivement en panne, le résultat d'exécution de la tâche affectée produit par l'unité défaillante ne correspondra pas au résultat d'exécution fourni par l'autre unité de la paire. La tâche sera donc assignée à une autre paire d'unités fonctionnelles qui produira un résultat correct retourné à l'utilisateur.

II. Etat de l'art sur la tolérance aux pannes

L'ordonnancement optimal des tâches en temps réel sur un système monoprocesseur ou un système multiprocesseur est déjà un problème NP-complet bien connu. Pour y parvenir, différents travaux ont été consacrés à la conception d'heuristiques dans le but de maximiser certaines mesures de performance comme l'ordonnançabilité, l'équilibrage de charge, l'utilisation du processeur etc. Cependant, très peu d'entre eux traite du problème de tolérance aux pannes au cours de l'ordonnancement de tâches en temps réel [62]. Pourtant, la construction d'un ordonnancement tolérant aux pannes a attiré l'attention des chercheurs depuis la fin des années 80 et le début des années 90 [76] [77] [78] [79] [80]. La plupart des travaux disponibles dans la littérature traitent de l'ordonnancement mono-processeur de tâches périodiques ou apériodiques, indépendantes, avec des hypothèses simplificatrices sur la périodicité des tâches et les coûts de calcul, en présence de pannes transitoires [81] [82] [83]. Notre sujet traitant de la tolérance aux pannes d'un système temps-réel multicoeur, nous proposons de faire un tour d'horizon des travaux relatifs afin de comprendre les idées mises en œuvre. Nombre de ces travaux sont présentés dans [39] [84] [62].

II.1. Pannes matérielles temporaires et intermittentes

La majorité des travaux qui traitent de ce type de pannes porte sur une politique d'ordonnancement partitionnée. Rappelons que cette politique traite le problème d'ordonnancement sur m processeurs comme m problèmes d'ordonnancement monoprocesseur. Ainsi, la tolérance globale du système peut s'obtenir en appliquant les résultats de tolérance obtenus en monoprocesseur à chaque unité d'ordonnancement multiprocesseur.
Les premiers travaux ont porté sur l'algorithme Rate Monotonic (RM). C'est ainsi qu'on peut citer dans un premier temps [85] qui définit une condition nécessaire et suffisante d'ordonnançabilité d'un système de tâches indépendantes et périodiques en dépit des pannes transitoires.

Les auteurs supposent qu'au plus une panne transitoire affecte une unité de traitement toutes les τ_F unités de temps et déterminent le temps de réponse de chaque tâche dans deux cas : le cas où la durée de la panne est négligeable et le cas où elle ne l'est pas. L'hypothèse de départ est que lorsqu'une tâche est préemptée, son état est sauvegardé en mémoire avec suffisamment de code correcteur d'erreurs. De ce fait, seuls les résultats de l'instance de la tâche en cours d'exécution sont affectés par la panne.
En supposant que les tâches sont classées par ordre croissant de période, le pire temps de réponse d'une tâche si la durée de panne est négligeable est donné par l'équation suivante :

$$R_i = \begin{cases} C_i + \left\lceil \frac{R_i}{\tau_F} \right\rceil \times C_i & \text{si i=1} \\ C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j + \left\lceil \frac{R_i}{\tau_F} \right\rceil \times \{max(C_1, C_2, ..., C_i)\} & \text{si i>1} \end{cases}$$

Si la durée de la panne t_{rec} est non négligeable, le pire temps de réponse d'une tâche est donné par l'équation suivante :

$$R_i = \begin{cases} C_i + \left\lceil \frac{R_i}{\tau_F} \right\rceil \times (C_i + t_{rec}) & \text{si i=1} \\ C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j + \left\lceil \frac{R_i}{\tau_F} \right\rceil \times \{max(C_1, C_2, ..., C_i) + t_{rec}\} & \text{si i>1} \end{cases}$$

Un système est donc ordonnançable en dépit des pannes si le pire temps de réponse de toute tâche (calculé par l'une des formules ci-dessus) est inférieur à son délai critique ($R_i < D_i$).

S'inscrivant dans le même registre, [86] définit une condition suffisante d'ordonnançabilité basée sur la charge du système. Les auteurs prouvent que tout système de tâches à échéances implicites de charge inférieure ou égale à 0.5 est ordonnançable par RM sur un processeur, en dépit de la survenue d'une panne transitoire ou intermittente, même en cas de ré-exécution des instances non achevées.

[87] propose un algorithme permettant de construire un système de tâches apériodiques ordonnançable par EDF en dépit de plusieurs pannes. L'algorithme est basé sur le calcul de la durée d'exécution maximale additionnelle (δ_i) qu'une tâche τ_i engendre lorsqu'elle a été affectée par w pannes transitoires durant un intervalle de temps donné. On considère que les dates de fin d'exécution des tâches (f_i) dans le scénario sans panne sont connues. De plus, lorsqu'une tâche est affectée par la panne, on suppose qu'elle engendre une exécution supplémentaire de durée égale son WCET, d'où $\delta_i = w \times C_i$. A chaque activation d'une nouvelle tâche, celle-ci n'est exécutée que si la durée maximale additionnelle qu'elle engendre est inférieure ou égale à la durée de temps qui s'écoule entre sa fin d'exécution prévue (dans le scénario sans panne) et son délai critique (c-à-d $\delta_i \leq D_i - f_i$). Cette condition garantit que si en cours d'exécution une

tâche est impactée par la panne, la durée d'exécution additionnelle qu'elle engendre n'affecte pas l'exécution des autres tâches.

Quelques résultats de tolérance en ordonnancement global sont tout de même rencontrés. On peut citer [88] qui propose un test d'ordonnançabilité basé sur le facteur d'utilisation des tâches suivant une approche probabiliste. [89] et [90] quant à eux proposent des méthodes de tolérance basées sur la technique Primary/Backup (voir §II.2.a). Dans cette technique, une tâche possède une copie primaire et une ou plusieurs copies de secours dont l'une est exécutée en cas de faute sur la copie primaire. C'est ainsi que [90] propose l'algorithme RRFTGS (Resource Reclaim Fault Tolerant Global Scheduling) pour réduire le coût temporel additionnel engendré par la tolérance. L'idée de RRFTGS est de repousser au plus loin l'exécution des copies de secours des tâches afin d'augmenter la disponibilité des ressources pour l'exécution rapide des copies primaires.
D'autres résultats sont publiés dans [91] aussi bien pour les systèmes des tâches apériodiques avec l'algorithme EDF, que pour les systèmes de tâches périodiques avec l'algorithme PF. Ce dernier cas retient notre attention car, tout comme le présent travail, il porte sur la tolérance aux pannes d'un ordonnancement équitable de tâches périodiques. Les auteurs proposent en effet un algorithme (FT-PF : Fault-tolerant Pfair) qui permet de construire un ordonnancement équitable valide en dépit d'une panne transitoire affectant une unité de traitement. L'algorithme est basé sur l'augmentation de la pire durée d'exécution (WCET) des tâches de manière à ce qu'elles puissent terminer leurs exécutions V_i unités de temps avant leurs échéances. Ceci garantit qu'en cas de panne, chaque tâche dispose d'au moins V_i unités de temps pour terminer son exécution sans faute temporelle. On passe donc d'un système de tâches $S : \tau_i < C_i, T_i >$ à un système $S' : \tau'_i < C'_i, T_i >$ avec $C'_i = \left\lceil \frac{C_i T_i}{T_i - V_i} \right\rceil$. Les auteurs prouvent qu'un système de tâches qui satisfait la condition (suffisante) $\sum_{i=1}^{n} (\frac{C'_i}{T_i}) + 1 \leq p$ est ordonnançable avec FT-PF sur p unités de traitement en dépit d'une panne temporaire.

II.2. Pannes matérielles permanentes

Lorsqu'une panne affecte durablement une unité d'exécution, l'ordonnanceur doit pouvoir réaffecter les travaux qui lui étaient destinés aux unités fonctionnelles restantes, ce qui engendre des coûts de migration et pose le problème d'équilibrage des charges au niveau des unités fonctionnelles. La question de la redistribution dynamique de la charge de l'unité défaillante aux unités fonctionnelles restantes a été abordée dans [92] tandis que la réduction des coûts de migration est traitée dans [5].
Tout comme pour les pannes transitoires, la quasi-totalité des travaux abordant la tolérance aux pannes permanentes est basée sur la politique partitionnée. Peu de travaux portent sur l'ordonnancement global et nous n'en avons rencontré aucun sur les algorithme équitables. Les méthodes de tolérance proposées dans ces travaux reposent sur trois principales techniques :

Primary/Backup, l'utilisation d'unités de remplacement et l'utilisation des points de contrôle.

II.2.a. La technique Primary/Backup

C'est la technique de tolérance la plus utilisée dans la littérature [93] [94] [95] [96] [82] [97] [62] [90]. Elle combine redondance matérielle et redondance logicielle. L'idée consiste à disposer pour chaque tâche d'une copie primaire et d'une ou plusieurs copies de secours (backup). La copie de secours n'est pas nécessairement une version identique à la primaire. Il s'agit en général d'une version plus légère, qui fournirait un résultat acceptable. Les deux copies (primaire et secondaire) sont ordonnancées sur des processeurs différents. Dans certaines approches [98] les copies de secours ne doivent pas se chevaucher (c-à-d être allouées dans le même intervalle de temps) dans l'ordonnancement, mais dans la plupart des travaux, elles sont autorisées à se chevaucher [99] car au plus une d'entre elles s'exécutera et donc le chevauchement ne sera jamais effectif. Dans tous les cas, aucun chevauchement n'est autorisé entre les copies primaires (un processeur n'exécute qu'une tâche à la fois) ou entre une copie primaire et une copie de secours (voir Figure 3.5). [77] et [40] proposent des algorithmes d'affectation de tâches qui garantissent ces critères d'ordonnancement.

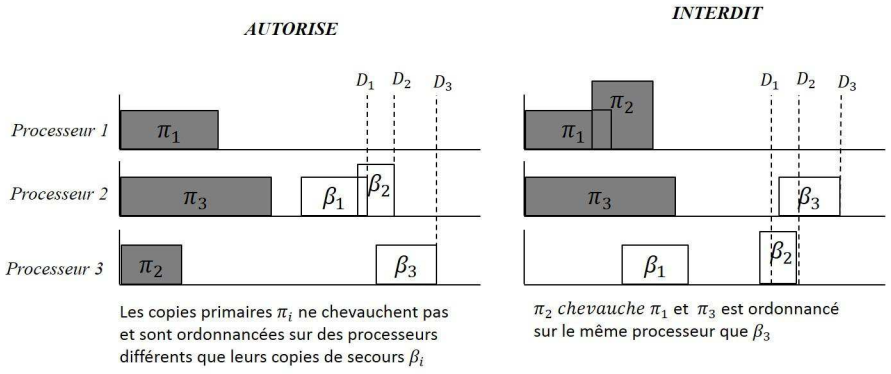

FIGURE 3.5 – Ordonnancement des tâches dans la technique Primary/Backup

Avant la panne, seules les copies primaires sont exécutées. Un test d'acceptation permet de valider l'exécution d'une tâche. Si l'exécution de la copie primaire produit un résultat correct, la copie de secours est désallouée. Si elle échoue, la copie de secours est activée et exécutée avant l'échéance de la tâche. Lorsqu'une panne affecte une unité de traitement, les copies primaires qui y étaient affectées sont perdues. Dans ce cas, leurs copies de secours respectives sont activées et prennent le relai sur les unités fonctionnelles restantes. La désallocation des copies de secours apporte une gestion plus efficace du temps et des ressources. De plus, il a été prouvé [82] qu'avec cette désallocation, le chevauchement entre les copies de secours a peu d'incidence sur l'ordonnançabilité.

II.2.b. Utilisation d'unités de remplacement

Cette technique repose essentiellement sur la redondance matérielle. L'idée est de prévoir des unités de traitement supplémentaires qui prendront le relais des unités défectueuses. Une fois qu'une unité tombe en panne, les travaux restants sont affectés à une unité de substitution qui poursuit l'exécution [39]. Comme travaux s'inspirant de cette technique, on peut citer [79] qui propose d'associer à chaque unité de traitement une unité jumelle à laquelle sont affectées les mêmes tâches. Chaque tâche a une copie primaire et une copie de secours. Une fonction de permutation permet de répartir les tâches entre deux unités jumelles de manière à ce que chaque copie primaire soit affectée à une unité différente de sa copie de secours. Le chevauchement entre les copies de secours est autorisé. Lorsqu'une unité de traitement tombe en panne, sa jumelle prend le relai. [100] propose un circuit multicoeur dans lequel à chaque mémoire cache sont associés deux cœurs : un cœur primaire associé à la portion primaire de la mémoire et un cœur secondaire associé à la portion secondaire. La portion secondaire de la mémoire cache est redondante avec la portion primaire. Une commande du circuit permet d'activer le cœur secondaire lorsqu'une panne est détectée sur le cœur primaire et de prendre le relai de l'exécution des tâches. Le contexte d'exécution étant les mêmes dans les deux portions de la mémoire cache, il n'y a pas de perte d'exécution.

II.2.c. Utilisation des points de contrôle

Les points de contrôle (ckeckpoints) sont une technique utilisée dans plusieurs travaux pour réduire la perte d'exécution engendrée par la panne [101] [102] [103] [40]. Elle est efficace particulièrement dans le cas où la durée d'exécution des tâches est grande. Dans un tel contexte, une panne survenant pendant l'exécution d'une tâche peut entraîner d'énormes pertes de calculs et nécessiter la re-exécution de toute une instance et donc du temps supplémentaire. Le principe est que durant l'exécution d'une tâche, son état ou contexte d'exécution (valeur des variables, contenus des registres, code et.) est sauvegardé dans une mémoire annexe exempte de panne. L'état ainsi sauvegardé est appelé point de contrôle (checkpoint) ou point de récupération (recover point). Lorsqu'une panne affecte une unité de traitement, l'exécution de la copie primaire ou une copie de secours ([104] [105]) se poursuit à partir du dernier point de contrôle. Toutefois, la détermination du nombre de points de contrôle dans une tâche et de l'intervalle de temps entre deux points de contrôle reste une problématique de recherche actuelle [106] [107].

III. Analyse des approches existantes

Pour ce qui est de notre périmètre d'étude, les techniques de tolérance présentées dans la section précédente possèdent des atouts à exploiter et des inconvenients à limiter dans nos approches de tolérance.

III.1. Étude des différentes techniques

III.1.a. La technique des unités de remplacement

L'utilisation des coeurs de remplacement a l'avantage d'être pratique, robuste et de ne pas nécessiter une réconfiguration du système en cas de panne. Cependant, elle pose le problème de synchronisation entre le cœur défaillant et le cœur de remplacement. En effet, dans un circuit où chaque cœur dispose de sa propre mémoire, ceci engendre des coûts de transfert entre les mémoires qui influencent la durée d'exécution des tâches et peut compromettre le respect des contraintes temporelles. De plus, cette technique n'est pas optimale du point de vue de la gestion de ressources et du temps d'exécution. Enfin, si le cœur de remplacement reste libre avant la panne, on n'optimise pas la charge traitée.
Par ailleurs, si à chaque cœur il faut associer un cœur de remplacement, cela introduit trop de redondance matérielle. Pour un système ordonnançable sur m coeurs, il faudra prévoir un processeur à $2 \times m$ coeurs. Or, l'augmentation du nombre de cœurs entraine l'augmentation du volume, du poids et du coût du processeur, ce qui est représente un choix discutable pour les systèmes embarqués.

III.1.b. La technique Primary/Backup

En ordonnancement partitionné la technique Primary/Backup est très efficace parce qu'elle assure que la perte d'un cœur n'entraine pas la perte des tâches qui lui étaient affectées. De plus, l'utilisation des copies de secours (en général plus légères que les copies primaires) a l'avantage d'assurer le respect des échéances des tâches en cas de panne.
Toutefois, puisque la copie primaire et les copies de secours d'une tâche sont toutes ordonnancées, cette technique entraine une augmentation de la charge du système à exécuter et donc du nombre de cœurs à prévoir. Par ailleurs, elle soulève des problèmes d'application en ordonnancement global, où l'affectation des tâches n'est pas statique, comme en ordonnancement partitionné. Dès lors, l'affectation des tâches étant dynamique et les migrations autorisées, il est difficile d'assurer que les copies de secours d'une tâche soient ordonnancées sur des cœurs différents de celui où est ordonnancée la copie primaire. Enfin, il n'est pas toujours possible de disposer des versions légères de chaque tâche en guise de copie de secours.

III.1.c. La technique des points de contrôle

L'intérêt des points de contrôle est indéniable : ils apportent un gain en temps dans la re-exécution des tâches impactées par la panne. Cependant, leur utilité dépend de leur nombre au sein d'une tâche. En effet, la durée de sauvegarde de l'état d'une tâche a une incidence sur sa durée d'exécution. Plus il y a de points de contrôle, plus grande est la durée d'exécution de la tâche. A ce propos, les expérimentations menées dans les travaux de [106] sur des tâches indépendantes révèlent qu'en présence de panne, dans une faible proportion (25%), un ordonnancement avec point de contrôle est meilleur (du point de vue du temps d'exécution) qu'un ordonnancement sans points de contrôle. Inversement, dans une large proportion ($> 25\%$), l'introduction des points de contrôle conduit à un ordonnancement non valide. L'utilisation des points de contrôle doit donc être judicieuse et opportune.

III.1.d. L'algorithme FT-PF

Tout comme notre sujet, l'algorithme FT-PF porte sur l'ordonnancement global équitable. Il a l'avantage de prévoir du temps de récupération pour chaque tâche avant son échéance, ce qui évite des fautes temporelles en cas de panne. Sa difficulté principale réside dans la détermination de la durée V_i de ce temps de récupération. A ce sujet, aucune indication n'a été donnée dans la publication. De plus, l'augmentation de la pire durée d'exécution des tâches entraine l'augmentation de la charge du système. Enfin, son résultat prouvé pour la tolérance aux pannes transitoires reste à étudier pour les pannes permanentes

III.1.e. La redondance temporelle

La redondance matérielle et la redondance logicielle nécessitent de prévoir des unités de traitement additionnelles, ce qui posent des problèmes de coûts, d'espace, de poids et de consommation énergétique. Or ces éléments sont déterminants pour le choix des processeurs, notamment dans des systèmes embarqués automobiles et avioniques [108]. Le but de la redondance temporelle est d'éviter cela en privilégiant plutôt du temps d'exécution additionnel. Cependant, elle entraine à son tour des surcoûts en temps d'exécution dus à des exécutions multiples des tâches. Ces surcoûts sont néfastes pour des applications critiques, où le respect des contraintes temporelles est une exigence forte. Cependant, elle peut être utilisée dans des système où la disponibilité est préférée à la fiabilité, à l'instar des serveurs web ou de certaines applications multimédia. Enfin, le parallélisme d'exécution (ordonnancement de la même tâche sur plusieurs coeurs au même instant) n'étant pas admis en ordonnancement global, la redondance temporelle par des exécutions multiples d'une tâche n'est pas envisageable, à moins de considérer chaque exécution de la tâche comme une tâche distincte.

III.2. Particularités de nos techniques

Les techniques de tolérance que nous proposons s'enrichissent des atouts des techniques existantes tout en essayant de limiter leurs défauts. Tout d'abord, contrairement à la technique des unités de remplacement et à la technique Primary/backup, nous proposons une redondance matérielle limitée au minimum de coeurs nécessaires pour garantir l'ordonnançabilité du système après la panne. De plus, les coeurs additionnels ne restent pas oisifs. Au contraire, avant la panne, les tâches s'exécutent sur l'ensemble des coeurs afin de maximiser la charge de travail traitée avant la panne, ce qui permet de préserver du temps après la panne pour une éventuelle reprise. Pour éviter la surcharge du système et réduire les coûts de conception, il n'est pas fait usage de copies de secours des tâches. Enfin, il n'y a pas de surcoûts en temps d'exécution engendrés par la redondance temporelle. Nous faisons par contre l'hypothèse de l'existence des points de contrôle, car nous supposons que le contexte d'exécution des tâches est sauvegardé à chaque unité de temps. En effet, on peut noter que les multiples préemptions liées à l'algorithme PD2 induisent naturellement l'existence de ces sauvegardes. Ainsi, les tâches impactées par la panne peuvent poursuivre leurs exécutions dans l'état où elles se trouvaient avant la panne. Pour gérer la reprise de l'exécution perdue, FT-PF réserve des unités de temps additionnelles. Nos deux premières techniques procèdent de même et fournissent en plus les méthodes de détermination de ce temps additionnel. De plus, nous proposons une troisième technique qui au lieu de réserver du temps additionnel exploite les unités de temps creux identifiés pour récupérer l'exécution perdue. Enfin, nos techniques prennent en compte le délai de détection de la panne. La prise en compte de ce délai ne ressort pas explicitement dans les travaux étudiés. Pourtant, ce délai permet d'estimer la portion d'exécution perdue et donc d'éviter la reprise complète des tâches impactées par la panne.

Conclusion

Ce chapitre précise notre périmètre d'étude dans le domaine de la sûreté de fonctionnement des systèmes informatiques. Il s'agit de la tolérance aux pannes permanentes d'un système temps-réel critique ordonnancé par un algorithme équitable. L'état de l'art révèle que ce périmètre d'étude demeure non exploré. Toutefois, les techniques classiques de construction d'un ordonnancement temps-réel valide en dépit de la survenue d'une panne en architecture multiprocesseur ou multicoeur ont été présentées. L'analyse de ces techniques met en évidence des atouts que nos techniques exploitent et révèle des écueils qu'elles s'efforcent d'éviter. Ceci met en exergue l'originalité des contributions de cette thèse. En effet, contrairement aux autres techniques qui sont basées sur les redondances matérielle, logicielle et temporelle, nos techniques reposent essentiellement sur la modification en ligne des paramètres temporels des tâches du système. Cela entraine le passage par plusieurs modes d'exécution. Les problèmes de changements de mode des systèmes temps-réels font l'objet du chapitre suivant.

Chapitre

4 Systèmes à criticités multiples et Changements de mode

Sommaire

Résumé

Ce chapitre présente la notion de changement de mode des systèmes temps-réel en général et des systèmes à criticités multiples en particulier sur laquelle reposent les techniques de tolérance proposées dans cette thèse.

Introduction

Afin de limiter les conséquences de la redondance (matérielle, logicielle ou temporelle) qu'on retrouve dans les techniques de tolérances classiques, les techniques proposées dans cette thèse reposent sur les opérations d'ajout ou de suppression des tâches du système, ainsi que de modification de leurs paramètres temporels en cours d'exécution. Ces opérations entrainent le passage du système à travers plusieurs modes d'exécution au cours de l'ordonnancement. Les changements de mode permettent à un système temps-réel de s'adapter aux modifications de son environnement d'exécution. A cet effet, ils doivent s'effectuer sous le contrôle d'un protocole qui garantit le respect des contraintes temporelles. C'est le rôle des protocoles de reconfiguration dynamique définis dans cette thèse dans le cadre de la tolérance aux pannes. Par ailleurs, dans nos travaux, lorsque le nombre de cœurs fonctionnels ne garantit plus l'ordonnançabilité du système, une solution consiste à réduire la charge du système en supprimant des tâches. Plusieurs critères peuvent être utilisés pour déterminer les tâches à supprimer. Parmi ces critères, le niveau de criticité de la tâche retient notre attention.
Le but de ce chapitre est de présenter ces notions de changements de mode (section 1) et de systèmes à criticités multiples (section 2).

I. Changements de mode d'exécution des systèmes

Au cours de l'exécution d'un système informatique, son environnement peut subir des modifications. La modification peut être causée par :

- **Un changement de phase d'exécution**. C'est le cas par exemple, d'un système de gestion de vol d'avion qui est composé de cinq phases principales : roulage, décollage, croisière, atterrissage et à nouveau roulage. C'est aussi le cas d'un système de contrôle automobile qui peut passer par les phases de démarrage (*start-up*), de régulation automatique de vitesse (*cruise control*), de contrôle par le conducteur (*driver control*) et de dépannage (*limp-home*). Dans chacune de ces phases, le système réalise différentes tâches sous différentes contraintes.
- **Des changements de paramètres**. C'est le cas des systèmes thermiquement résilients [109] qui opèrent dans les conditions où la température change dynamiquement et celui des systèmes nécessitant une longue autonomie [110] qui doivent collecter l'énergie de l'environnement et la stocker. Leur fonctionnement ainsi que leur gestion des ressources doivent s'ajuster à la quantité d'énergie disponible.
- **Des pannes matérielles**. C'est le cas de tout système temps-réel qui fait face à la diminution des ressources d'exécution. Le système doit poursuivre son exécution quitte à fournir un service dégradé. **C'est à ce type de cause que s'intéressent nos travaux**.

Pour s'adapter à de telles modifications, plusieurs modes d'exécution sont à prévoir dès la conception du système [111, 112, 113].

I.1. Définitions et exemple

I.2. Notion de mode

Un *mode* représente un état de fonctionnement du système. Il regroupe différentes tâches qui concourent ensemble à la réalisation d'un service. Une tâche est un code fonctionnel s'exécutant pour la réalisation d'un résultat. Des tâches du même mode peuvent s'exécuter concurremment en s'échangeant les données et en partageant des ressources. D'après [114], les modes d'un système peuvent être regroupés en trois catégories :

- **Les modes de fonctionnement normal :** le système passe par un certain nombre de phases différentes planifiées et exécutées régulièrement. C'est le cas des modes « *driver control* » et « *cruise control* » d'une automobile dans lesquels le système commute dans son fonctionnement ordinaire selon que la vitesse est contrôlée de manière automatique ou par le conducteur.
- **Les modes de fonctionnement exceptionnels** (parfois appelés modes opérationnels) **:** ils sont causés par de rares évènements qui entrainent l'exécution d'un code qui n'aurait pas été exécuté si l'évènement n'était pas survenu. La réponse à un tel évènement est planifiée, mais le changement de mode résultant peut ne pas se produire. C'est par exemple le cas du mode « *prepare for crash* » dans lequel entre un système automobile lorsque le système de surveillance détecte qu'un impact est sur le point de se produire. Dès lors, il ferme les fenêtres, resserre les ceintures de sécurité, assouplit les freins et se prépare à déployer l'airbag. Si un tel évènement n'était pas sur le point de se produire, le système resterait dans son mode de fonctionnement normal ;
- **Les modes de fonctionnement dégradé** (parfois simplement appelé dégradation progressive) **:** ils sont causés par une panne ou une erreur qui exige que la charge du système soit réduite et que priorité soit donnée aux tâches de sécurité et aux tâches des fonctionnalités basiques. Les réponses générales aux évènements de changement de mode sont prévues, mais l'ensemble des conditions d'erreurs peut ne pas être connu à l'avance. Un exemple est le mode « *limp home* » dans lequel bascule un système d'automobile après la défaillance du capteur moteur.

Ainsi donc, un système peut avoir plusieurs modes en fonctionnement normal qui évoluent suivant un ordre séquentiel statique. Il peut y aussi avoir en plus un petit nombre de modes en fonctionnement exceptionnel et des modes en fonctionnement dégradé. Dans des systèmes plus complexes, ces modes peuvent être organisés de manière hiérarchique.

I.3. Notion de module

Les *modules* sont des entités autonomes permettant de décomposer l'application et de l'exécuter de façon distribuée. Chaque module comporte plusieurs modes dont un seul est exécuté à un instant donné. Le mode initial est celui qui est exécuté lors du démarrage du système. En

d'autres termes, un système se compose d'un ou de plusieurs modules comprenant chacun des modes d'exécution. Si les modules peuvent s'exécuter de manière concomitante, un seul mode à la fois est exécuté dans un module. Pour des raisons de simplification, les systèmes considérés dans cette thèse sont constitués d'un seul module.

I.4. Notion de changement de mode

Le changement de mode désigne le passage du système d'un mode d'exécution à un autre provoqué par un évènement interne ou externe. Ce changement est supposé s'effectuer en un temps négligeable. L'évènement déclencheur (*trigger*) peut être déclaré au sein d'un mode à une fréquence donnée sous forme d'une condition à vérifier périodiquement. A chaque période, la condition est testée ; si elle est vérifiée le système passe au mode suivant. L'évènement peut aussi être lié à un changement interne du système ou de l'environnement via une lecture d'entrée. Par exemple, dans un système automobile lorsque le conducteur actionne la pédale de frein cela génère un évènement qui va faire passer le système du mode « *cruise control* » au mode « *driver control* ». L'évènement peut aussi être un déclencheur chronométré (timed-trigger) si le changement de mode est dirigé par une horloge. C'est le cas des systèmes de contrôle de vol qui ont un mode jour et un mode nuit entre lesquels ils basculent à des instants particuliers. Enfin, l'évènement déclencheur peut provenir de la plateforme matérielle ou d'un sous-système de surveillance de l'état de santé du système. C'est ce qui se produit lors du passage du système automobile dans le mode « *limp home* » après la défaillance du capteur moteur ou bien (comme cela est supposé dans cette thèse) lors de la détection de la défaillance d'un cœur du processeur.

Les opérations effectuées lors d'un changement de mode peuvent être de deux types (voir [115]) :

1. Les opérations qui augmentent la charge du système : l'ajout d'une tâche, l'augmentation de la durée d'exécution d'une tâche ou l'augmentation de la fréquence d'exécution d'une tâche.
2. Les opérations qui réduisent la charge du système : la suppression d'une tâche, la réduction de la durée d'exécution d'une tâche ou la réduction de la fréquence d'exécution d'une tâche.

Toutes ces opérations sont utilisées dans nos techniques de tolérance.

I.5. Exemple

Soit un régulateur qui délivre un signal de commande calculé à partir de deux grandeurs physiques mesurées par des capteurs. Les valeurs des mesures sont transmises au régulateur par un réseau du type TDMA (*Time Division Multiple Access*) de sorte que :

— La valeur courante du premier capteur est transmise dans les créneaux [0, 10], [40, 50], [80, 90], . . .

— La valeur courante du second capteur est transmise dans les créneaux [20, 30], [60, 70], [100, 110], . . .
— Le signal de commande doit être mis à jour toutes les 40 unités de temps.

Le Régulateur a donc deux capteurs (leurs identifiants dans le réseau) et un actuateur (valeur du signal de commande). Lors du démarrage, les valeurs du capteur doivent être initialisées afin que le calcul du signal de commande soit possible. La pire durée de lecture d'un capteur est de 10 unités de temps et la pire durée de calcul de la valeur du signal est de 20 unités de temps.

Du point de vue conceptuel, ce régulateur peut être modélisé par un module comprenant deux modes : le mode *Initial* pour l'initialisation des valeurs des capteurs au cours des 40 premières unités de temps et le mode *Normal* pour le calcul de la valeur du signal de commande toutes les 40 unités de temps.
Le mode *Initial* comporte deux tâches périodiques $\tau_1 < 10, 40 >$ et $\tau_2 < 20, 40 >$ qui reçoivent les valeurs courantes des capteurs en établissant la communication avec eux par le réseau TDMA. Le mode *Normal* comporte en plus, une tâche périodique $\tau_3 < 20, 40 >$ qui calcule et met à jour la valeur du signal de commande à partir des valeurs transmises par τ_1 et τ_2. Le changement de mode se fait donc lors du calcul de la première valeur du signal et est déclenchée par une horloge interne, 40 unités de temps après le début de l'exécution.

La Figure 4.1 représente l'exécution du module Régulateur sur une architecture multicœurs pendant 120 unités de temps à partir de son démarrage. Dans l'intervalle [0, 40] les tâches τ_1 et τ_2 sont exécutées dans le mode *Initial*. A l'instant 40, le système passe au mode *Normal* et la tâche τ_3 est ajoutée.

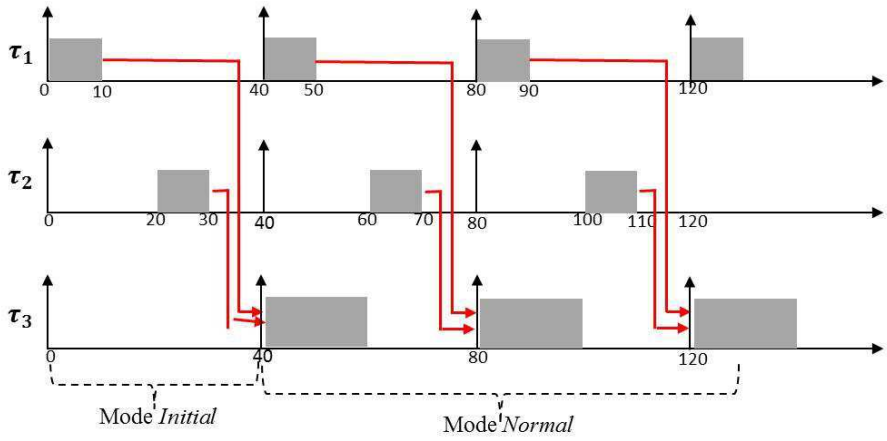

FIGURE 4.1 – Illustration d'un changement de mode : basculement à l'instant 40 du mode Initial qui contient τ_1 et τ_2 au mode Normal où τ_3 est ajoutée

I.6. Types de tâches impliquées dans un changement de mode

Du point de vue de l'ordonnancement, le changement de mode, s'opère par l'ajout, la suppression ou la modification des paramètres des tâches. C'est ainsi qu'on peut distinguer cinq types de tâches (voir Figure 4.2) rangées suivant deux critères :

— **Selon le niveau d'exécution**

⊙ *Tâche abandonnée* : tâche de l'ancien mode dont l'exécution est immédiatement interrompue à l'instant du changement de mode. Cette tâche ne s'exécute que dans l'ancien mode ;

⊙ *Tâche terminée* : tâche de l'ancien mode dont on autorise la poursuite de l'exécution de l'instance courante parce que le résultat produit est vital pour le bon fonctionnement du système. Cette tâche s'exécute dans l'ancien et dans le nouveau mode ;

⊙ *Tâche nouvelle* : Tâche dont l'exécution débute pour la première fois dans le nouveau mode. Elle est activée dans le nouveau mode et ne figurait pas dans l'ancien mode. Dans certains protocoles, l'exécution d'une telle tâche peut être retardée si certaines instances des tâches de l'ancien mode requièrent du temps pour terminer leur exécution ;

— **Selon les paramètres temporels**

⊙ *Tâche inchangée* : tâche présente dans l'ancien mode et dans le nouveau mode sous les mêmes paramètres temporels ;

⊙ *Tâche changée* : tâche qui s'exécute dans l'ancien mode et dans le nouveau mode, mais sous des paramètres temporels différents.

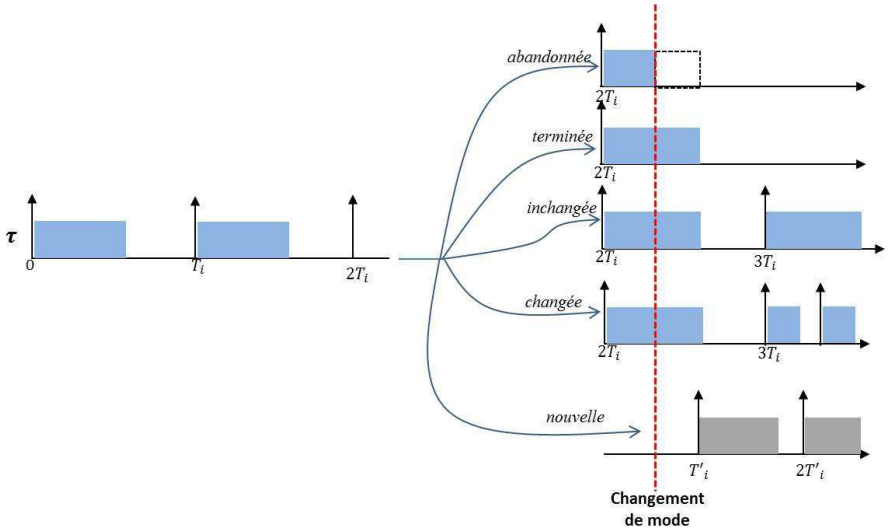

FIGURE 4.2 – Types de tâches rencontrées dans un changement de mode

Les nouvelles versions des tâches inchangées ou changées sont déclenchées immédiatement à la fin de la période des dernières instances activées dans l'ancien mode ou après un certain délai suivant cet instant. Si les schémas d'exécution des tâches inchangées ne sont pas affectés par le changement de mode on dit que la périodicité des tâches est préservée.

I.7. Protocoles de changement de mode

Un protocole de changement de mode définit la manière dont les tâches de l'ancien mode mettent un terme à leurs exécutions, dont celles du nouveau mode sont introduites, et dont celles qui sont communes aux deux modes traversent la phase transitoire. Les protocoles de changement de modes les plus connus sont présentés dans l'article de Jorge Real [116]. On distingue trois familles : les protocoles *immédiats* dans lesquels tous les travaux en cours du mode courant sont suspendus au profit des travaux du nouveau mode qui débutent immédiatement ; les protocoles *synchrones* qui retardent l'instant du lancement des tâches dans le nouveau mode jusqu'à ce que toutes les tâches de l'ancien mode soient terminées ; les protocoles *asynchrones* qui permettent que les tâches de ces deux modes s'exécutent simultanément pendant la phase transitoire. **C'est à cette famille de protocole qu'appartiennent les protocoles définis dans nos techniques de tolérance.**

L'analyse d'ordonnançabilité permet de vérifier qu'un protocole garantit ou non le respect des échéances des tâches. En effet, du point de vue de l'ordonnancement, à des modes différents correspondent des exigences temporelles différentes. Donc l'ordonnançabilité d'un mode n'entraine pas l'ordonnançabilité du mode suivant. L'analyse d'ordonnançabilité doit donc porter sur chaque mode et sur la phase transitoire. Pour le changement de mode immédiat, il n'y a pas de problème d'ordonnancement spécifique. Cependant, il y a une obligation pour le RTOS et/ou pour le langage de programmation de faciliter la suspension rapide et immédiate des travaux. Si le travail suspendu utilisait une ressource sollicitée dans le nouveau mode, alors des mesures sont à prendre pour libérer la ressource ou pour autoriser le travail à suspendre à terminer son exécution jusqu'à ce que la ressource soit libérée (ce qui se ramène au cas synchrone). Pour les changements de mode synchrones, des analyses d'ordonnancement ont été produites [117], lesquelles calculent pour chaque tâche du nouveau mode un délai minimal de décalage d'activation. A chaque fois que le changement de mode est déclenché, ces décalages assurent que les travaux de l'ancien mode se terminent avant leurs échéances et que les tâches du nouveau mode démarrent leur exécution le plus tôt possible. L'ordonnancement se complique lorsqu'il y a des tâches qui changent de période.

Plusieurs auteurs [118, 115] classent les protocoles immédiats dans la famille des protocoles synchrones, ce qui ramène le nombre de familles à deux. Dans la suite, nous adoptons cette classification. Dans un premier temps, nous présentons les protocoles synchrones et asynchrones

utilisés en architecture monoprocesseur. Puis, nous présentons quelques protocoles en architecture multiprocesseur.

I.7.a. Protocoles synchrones

Dans ces protocoles, les tâches du nouveau mode ne sont activées que lorsque toutes les tâches de l'ancien mode sont achevées. Cette séparation entre tâches des deux modes successifs fait de l'ordonnançabilité de chacun des modes du système une condition nécessaire et suffisante de l'ordonnançabilité du système entier. Outre la simplification de l'analyse d'ordonnançabilité, les protocoles de cette classe sont faciles à implanter et assurent la cohérence entre tâches des différents modes. Leur paramètre caractéristique est le délai maximal entre la demande de changement de mode et le lancement du nouveau mode. Bien qu'ils sont moins utilisés dans les travaux, à cause de leur faible réactivité par rapport aux protocoles asynchrones, les protocoles synchrones constituent la base des protocoles multicoeurs actuels. En architecture monoprocesseur, les protocoles synchrones les plus connus sont le protocole des instants oisifs et le protocole du délai minimal.

— **Le protocole des instants oisifs**
Ce protocole a été proposé par Tindell et Alonso en 1996 [119]. Son principe consiste à repousser l'activation des tâches du nouveau mode au prochain instant oisif, c'est-à-dire, au moment où le processeur aura achevé d'exécuter toutes les tâches courantes et sera libre. Malgré les avantages apportés par le synchronisme, ce protocole présente une relativement faible réactivité. D'après [118], dans le pire cas, l'ensemble des tâches ordonnées sous EDF occupe 100% de temps du processeur et une demande de changement de mode intervenue juste après l'instant critique ne pourra être effectuée qu'à l'instant critique suivant.

— **Le protocole du délai minimal (Minimum Single Offset : MSO)**
Proposé par J. Real en 2000 [120], ce protocole existe en deux versions : avec et sans périodicité. Contrairement à ce qu'autorise le protocole des instants oisifs, les nouvelles instances de tâches de l'ancien mode ne sont plus activées dès que la demande de changement de mode est émise. Bien entendu, les instances déjà activées s'achèvent normalement. Ce protocole se caractérise par le calcul du délai après lequel les tâches du nouveau mode peuvent être introduites. Sa faiblesse réside dans le fait qu'il se peut que certaines tâches inchangées échangent des données, essentielles à leur fonctionnement, avec les tâches de l'ancien mode arrêtées durant la phase transitoire, et avec leurs successeurs dans le nouveau mode lesquelles ne seront actives qu'après ce délai. La cohérence du système peut être alors mise en danger. Les protocoles multicoeurs présentés en §$I.7.b$ sont une émanation de MSO.

I.7.b. Protocoles asynchrones

La meilleure réactivité de ces protocoles est due au fait que les tâches du nouveau mode sont déclenchées pendant que celles de l'ancien mode sont encore en train de s'exécuter. Cependant, la superposition momentanée de deux modes dans la phase transitoire produit temporairement une charge de processeur plus élevée. Il est donc primordial, afin d'assurer l'ordonnançabilité du système, d'évaluer correctement la charge supplémentaire engendrée par la transition. Les protocoles monoprocesseurs asynchrones sont présentés dans ce paragraphe sur la base de l'analyse d'ordonnançabilité utilisée. C'est ainsi que nous distinguerons les protocoles basés sur les algorithmes à priorités fixes et les protocoles basés sur les algorithmes à priorités dynamiques.

— **Priorité fixe**
Les protocoles de Sha [115] et Lehoczky [121] ont tout d'abord été analysés sous le critère du calcul du taux d'utilisation. Cependant, il s'est avéré que ce calcul ne permet pas, dans la phase transitoire dans laquelle la régularité des activations des tâches est temporairement perturbée, de fournir une estimation fiable de la charge effective du système. Ce critère a donc été abandonné au profit du calcul du pire temps de réponse qui offre le moyen d'évaluer, sur un intervalle de temps s'étendant sur deux modes différents, les interférences que subit une tâche moins prioritaire imputables à l'exécution des tâches plus prioritaires des modes différents. Parmi les protocoles analysés sous ce critère, on peut citer le protocole de Pedro et Burns [117, 122], celui de Tindell [123] et celui de Real et Crespo [116]. Ces protocoles se différencient entre eux par les comportements adoptés par les tâches au regard du changement de mode. Leur analyse est construite, sur la base de fenêtres temporelles qui permettent d'observer à différents moments de passage d'un mode à un autre, quelles tâches, à partir de quand et jusqu'à quand, sont exécutées.

— **Priorité dynamique**
Pour l'analyse de la phase transitoire, la plupart des protocoles porte sur l'algorithme EDF, à l'instar de [124] dont l'analyse d'ordonnancement porte sur la demande processeur. Ce critère qui paraît être un choix pertinent soulève plusieurs questions dont celle de la détermination de la durée de l'intervalle d'étude. Par contre, le taux d'utilisation, qui ne permet pas de garantir l'ordonnançabilité dans le cas des priorités fixes, peut constituer dans le cas d'EDF, sous certaines conditions, un test que sa simplicité rend applicable en ligne. C'est sur ce constat qu'est construit le protocole d'Andersson [125]. Reprenant l'idée du protocole de Sha [115], l'auteur propose un test d'ordonnançabilité pour les systèmes de tâches utilisant le même mécanisme de changement de mode sous EDF. Le principe est le suivant : si la demande du changement de mode est observée à l'instant t_r, l'ancienne version d'une tâche τ_i ne peut être remplacée par une nouvelle version τ_i' qu'à la prochaine période de la tâche $Next(T_i)$. L'instant de remplacement de la tâche notée t_{ri} doit donc vérifier la condition $t_r \leq Next(T_i) \leq t_{ri}$ (La deuxième inégalité est stricte si la nouvelle version de la tâche est retardée). Un système de tâches est ordonnançable pendant un changement de mode se produisant à l'instant t_r, si dans

tous les intervalles de temps $\Delta = [t_0, t_1]$ tels que $t_0 \leq t_r < t_1$, la demande de ressource processeur ne dépasse pas sa capacité.

I.7.c. Des Protocoles pour multiprocesseur

Dans une architecture multiprocesseur, il n'est pas simple, compte tenu de la migration de tâches entre processeurs, de déterminer quels processeurs sont alloués aux tâches au moment du changement de mode. Les protocoles de changement de mode existants sont des extensions du protocole du délai minimal (Minimum Single Offset : MSO) présenté précédemment. Nelis et al [126, 127, 128] ont proposé deux versions (synchrone et asynchrone) de MSO pour le multiprocesseur : *Synchronous Multiprocessor MSO* (SM-MSO) et *Asynchronous Multiprocessor MSO (AM-MSO).*

Le principe de ces deux protocoles est le même que celui du protocole MSO original : à partir de l'instant du changement de mode, les nouvelles instances des tâches ne sont plus activées et celles déjà en cours se terminent normalement. Les tâches de nouveau mode sont déclenchées différemment dans chacun des protocoles. Dans SM-MSO, elles ne sont introduites qu'après la fin de toutes les tâches de l'ancien mode. Dans AM-MSO, une première tâche du nouveau mode peut commencer son exécution, sur un des processeurs libres, lorsqu'il n'y a plus de tâche de l'ancien mode à ordonnancer.
L'analyse de ces deux protocoles ne porte sur aucune politique d'ordonnancement spécifique, et fait toujours l'hypothèse de l'allocation de tâches au moment du changement de mode la pire possible. Elle cherche simplement à obtenir des bornes sur les délais après lesquels les tâches de nouveau mode sont activées.
Les améliorations apportées à ces deux protocoles sont de trois ordres. Tout d'abord, Nelis et al. [128] ont intégré les tâches inchangées dans l'analyse. Ensuite ils ont réévalué les bornes avec global EDF comme politique d'ordonnancement. Finalement, Yomsi et al. [129] ont intégré à l'analyse des facteurs déterminant les vitesses auxquelles chacun des processeurs exécute les tâches.

II. Système à criticités multiples

Les premiers travaux sur les systèmes à criticités multiples (*MCS : Mixed Criticality Systems*) ont été publiés par Vestal en 2007 [130]. La présentation qui est faite dans cette section s'appuie sur les travaux de Burns et Davis [131], lesquels sont plus récents et détaillés.

II.1. Définition et modélisation

II.1.a. Définition

La *criticité* désigne le niveau d'assurance contre les pannes nécessaire pour un composant d'un système. Un système à criticités multiples est un système qui a deux ou plusieurs niveaux de criticité distincts. La plupart des systèmes embarqués complexes que l'on trouve, par exemple, dans les industries automobiles (standard AUTOSAR) et avioniques (standard ARINC), évoluent en systèmes à criticités multiples afin de satisfaire certaines exigences strictes non fonctionnelles relatives au coût, à l'espace, au poids, à la production de chaleur et à la consommation d'énergie. La problématique majeure est de combiner, de manière rigoureuse, l'exigence de partitionnement pour une assurance de sécurité et l'exigence de partage pour une utilisation efficace des ressources. Par exemple, dans l'avionique, le standard ARINC contenu dans la norme DO-178B définit 5 niveaux de criticité, notés de A à E. Une défaillance (ie. le non respect d'une échéance) d'une tâche de criticité **A** peut avoir des conséquences catastrophiques sur le système tout entier (par exemple, crash d'un avion), tandis qu'une défaillance d'une tâche de criticité **E** n'a pas de conséquences notables sur le système. Le tableau 4.1 résume les conséquences d'une défaillance d'une tâche en fonction de son niveau de criticité.
Toutefois, il faut souligner que toutes les normes et documents sur MCS n'attribuent pas la même signification au terme «criticité». Cette question a été explorée par Graydon et Bate [132], Esper et al. [133] et Paulitsch et al. [134].

Niveau	Condition	Description
A	Catastrophique	Une défaillance peut provoquer un crash.
B	Dangereuse	Une défaillance a un large impact négatif sur la sécurité ou les performances, ou réduit les possibilités de l'équipage à gérer l'avion à cause de souffrances physiques ou une charge de travail trop importante, ou provoque des blessures sérieuses voire fatales parmi les passagers.
C	Majeure	Une défaillance est significative, mais a moins d'impact qu'une défaillance dangereuse (par exemple, provoque un inconfort des passagers plutôt que des blessures).
D	Mineure	Une défaillance est notable, mais a moins d'impact qu'une défaillance majeure (par exemple, provoquant des désagréments auprès des passagers ou un changement de plan de vol).
E	Sans effet	Une défaillance n'a pas d'impact sur la sécurité, les opérations ou la charge de travail de l'équipage.

TABLE 4.1 – Niveaux de criticité dans la norme DO-178B

II.1.b. Modélisation

Signalons d'emblée que tous les documents traitant des systèmes à criticités multiples n'utilisent pas la même modélisation. Nous adoptons ici le modèle le plus utilisé [135] que l'on retrouve dans les travaux présentés dans cette section.

Un système est modélisé comme un ensemble fini de composants. Chaque composant a un niveau de criticité L (défini par l'ingénieur système responsable de l'ensemble du système) et contient un ensemble fini de tâches périodiques ou sporadiques. Chaque tâche τ_i est définie par quatre paramètres : sa période T_i , son délai critique D_i, sa pire durée d'exécution C_i et son niveau de criticité L_i. Les tâches donnent lieu à une séquence illimitée de travaux.

Deux exigences sont fondamentales dans la mise en œuvre d'un MCS. La première est la séparation ou l'isolation des composants, d'où le partitionnement. Les tâches de différents composants ne doivent pas être autorisées à interférer les unes avec les autres. En particulier, des mécanismes doivent être mis en place pour éviter qu'un travail ne s'exécute sur une durée supérieure au WCET de sa tâche et pour assurer qu'une tâche ne génère pas de travaux dont la durée d'exécution se rapproche de sa période. L'obligation de protéger l'exécution d'un composant contre les fautes d'exécution d'un autre composant est déjà présente dans tous les systèmes qui hébergent plusieurs applications. Elle revêt plus d'importance si les composantes ont des niveaux de criticité différents. En effet, sans une telle protection, tous les composants devraient être conçus selon les normes strictes du niveau de criticité le plus élevé, ce qui augmente massivement les coûts de développement.
Après les préoccupations de partitionnement vient la nécessité d'une utilisation efficace des ressources. Ceci est facile dès lors qu'on comprend que les paramètres d'une tâche ne sont pas indépendants. En particulier, l'estimation du WCET doit dériver d'un processus dicté par le niveau de criticité. Le pire temps d'exécution C_i est remplacé par une fonction $C_i : N^+ \rightarrow R^+$, qui spécifie le pire temps d'exécution pour les différents niveaux de criticité. Nous pouvons donc noter que C_i n'est plus une constante, mais une fonction dépendant du niveau de criticité. Aussi, dans le cadre du modèle à criticités multiples, le pire temps d'exécution d'une tâche τ_i pour le niveau de criticité ℓ sera noté par $C_i(\ell)$.
Plus le niveau de criticité est élevé, plus le processus de vérification sera conservatif et donc la valeur du WCET sera plus grande. Telle a été l'observation au cœur des travaux de Baruah et Vestal [136] exprimée en ces termes : "'*the more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to be in pratice (Plus on a besoin de confiance dans le temps d'exécution d'une tâche, plus grande et plus conservative elle tend à être en pratique)*"'. Par exemple, une même tâche aura un WCET plus élevé si elle est définie comme étant *safety critical* (car un niveau d'assurance plus élevé est nécessaire), que si elle était simplement considérée *mission critical* ou même non critique. Cette estimation du WCET en fonction de la criticité influence considérablement les résultats classiques d'ordonnancement.

II.2. Ordonnancement

Les premiers travaux depuis Vestal en 2007 [130] ont porté sur l'ordonnancement monoprocesseur. C'est en 2009 qu'apparaissent les premières publications sur l'ordonnancement multiprocesseur.

II.2.a. En architecture monoprocesseur

Dans ses premiers travaux, Vestal utilise une extension de la théorie standard d'ordonnancement temps-réel en priorité fixe (FP) en architecture monoprocesseur basée sur l'analyse des temps de réponse [137]. Il montre que ni l'algorithme Rate Monotonic ni l'algorithme Deadline Monotonic ne sont optimaux pour les MCS. Cependant, l'algorithme d'Audsley [138] a été prouvé optimal. La preuve a été apportée par Dorin et al en 2010 [139]. La publication de Vestal a été suivie de celle de Baruah [136] qui généralise le modèle de Vestal en utilisant un modèle de tâches sporadiques ordonnancées suivant un algorithme à priorités fixes au niveau des travaux et suivant un algorithme à priorités dynamiques. Deux résultats se dégagent de ces travaux. Le premier est que l'algorihme EDF (Early Deadline First) ne domine pas les algorithmes à priorités fixes lorsque les niveaux de criticité sont introduits. Le deuxième résultat montre qu'il existe des systèmes faisables qui ne peuvent pas être ordonnancés par EDF.
D'autres résultats d'ordonnancement des MCS en monoprocesseur publiés dans [140, 141, 142, 143, 144, 145, 146] ont porté sur l'ordonnancement monoprocesseur d'un ensemble fini de travaux à criticités multiples avec des durées d'exécution dépendantes de la criticité. Ces recherches ont été largement remplacées par des travaux sur la criticité multiples des tâches qui est plus largement applicable.

II.2.b. En architecture multiprocesseur

Les premiers travaux sont ceux d'Anderson et al. en 2009 [147] étendus par Mollison et al. en 2010 [148]. Cinq niveaux de criticité ont été identifiés, allant du niveau A (le plus haut) au niveau-E (le plus bas). Les auteurs ont envisagé un schéma de mise en œuvre, qu'ils appellent MC2, qui utilise un ordonnancement cyclique pour le niveau A, EDF partitionné pour le niveau B, EDF global pour les niveaux C et D et enfin *Global Best-Effort* pour Niveau-E. Chaque processeur dispose d'un container pour chaque niveau de criticité et d'un ordonnanceur hiérarchique à deux niveaux. Ce schéma MC2 a inspiré plusieurs autres travaux à l'instar de [149, 150, 151] qui évaluent les surcoûts induits par l'OS associé aux plateformes multiprocesseurs. Ils ont également expérimenté des techniques d'isolement pour les caches de dernier niveau (LLC Last Level Cache) et DRAM et ont démontré, en utilisant MC2, les avantages d'avoir différentes techniques d'isolement (séparation) pour chaque niveau de criticité [151]. Le schéma MC2 est également utilisé par Bommert [152] pour supporter les tâches distribuées à criticités multiples.
D'autres résultats d'ordonnancement multiprocesseur des MCS peuvent être trouvés dans [153,

154, 155, 156, 157] pour la politique partitionnée, [158] pour la politique semi-partionnée et [159, 160] pour la politique globale.

II.3. Application aux changements de mode

D'après [114] , l'exécution d'un MCS peut se dérouler dans plusieurs modes définis en fonction des niveaux de criticité. De plus, un système peut combiner changement de mode général (voir section 1) et changement de mode de criticité. Pour ce dernier type, la plupart des articles se limitent à deux niveaux de criticité seulement : élevé (HI) et faible (LO) avec $HI > LO$. Ces systèmes sont appelés systèmes à double criticité (*dual-criticity systems*). Le système est alors soit dans un mode de criticité LO ou soit dans un mode de criticité HI et les paramètres d'une tâche sont $(T_i, D_i, C_i(HI), C_i(LO), Li)$ où $C_i(HI)$ et $C_i(LO)$ désignent les WCET de la tâche dans chaque mode. À l'extrême, les modèles présentées par Ekberg et al. [161, 162, 163] autorisent un nombre quelconque de modes et le changement entre les modes est représenté par un graphe acyclique orienté. Deux modèles de changement de mode de criticité sont rencontrés [114] : le modèle standard (SM : Standard Model) et le modèle adaptatif (AM : Adaptative Model). Les tâches sont supposées indépendantes, sans contraintes de précédence et ne partagent pas d'autres ressources que le processeur. N niveaux de criticité $L_0, L_1, ...L_{N-1}$ sont considérés.

II.3.a. Modèle standard

Dans ce modèle, l'exécution du système se déroule de la manière suivante :

- Le système démarre dans le mode de criticité le plus faible L_0 et reste dans ce mode aussi longtemps que les travaux terminent leur exécution avant leurs WCET $C(L_0)$ définis pour ce niveau de criticité.
- Si un travail épuise ses $C(L_0)$ unités de temps sans terminer son exécution, un changement de mode de criticité se produit : le système passe au mode de criticité suivant (L_1).
- Lorsque le système passe au mode L_1, toutes les tâches de criticité L_0 sont abandonnées. Aucun travail d'une tâche de criticité L_0 n'est exécuté.
- Le système reste dans le mode L_1 jusqu'à ce qu'un travail épuise ses $C_i(L_1)$ unités de temps sans terminer son exécution. Le système passe immédiatement au mode de criticité suivant et les travaux du niveau de criticité L_1 sont supprimées.
- Ce processus continue jusqu'à ce que le plus haut mode de criticité L_{N-1} soit atteint, ce mode contenant uniquement les travaux de niveau de criticité $N-1$.

Ce model standard est très utile pour décrire les propriétés fondamentales que doit avoir un MCS. Cependant, dans certains travaux [164, 165, 166, 167, 168, 169, 170], il a été nécessaire de l'étendre pour définir des propriétés plus souples telle que la possibilité d'autoriser certains travaux de criticité inférieure à poursuivre leur exécution dans le mode de criticité supérieure ou bien la possibilité d'un mode de criticité inférieure de se rétablir dans des conditions appropriées. **Nos travaux reposent sur ce modèle standard étendu.**

Le modèle standard définit donc un chemin du mode L_0 au mode L_{N-1}. Le protocole de changement de mode dans ce modèle se décrit comme suit :

- L_0 est le mode initial ;
- L_{N-1} est un mode final ;
- Tous les modes sont en sens unique ;
- Les transitions entre les modes sont immédiates (ou retardées dans certains modèles où certains travaux de criticité inférieure sont autorisés à terminer leur exécution) ;
- Suite à un changement de mode, certaines tâches ne s'exécutent que dans l'ancien mode (tâches abandonnées) ;
- Certains tâches s'exécutent dans les deux modes, mais leurs durées d'exécution sont modifiées ;
- Il n'y a pas de nouvelles tâches dans un mode.

Dans les premiers travaux [130, 136] il est supposé que le mode L_0 est le seul mode normal dans lequel le système s'exécute. Le changement vers un autre mode ne se produit que de façon exceptionnelle lorsque les ressources d'exécution sont réduites.

II.3.b. Modèle Adaptatif

Alors que le modèle standard n'autorise les changements de mode que dans une seule direction (d'un mode de criticité inférieure vers un mode supérieur), le modèle adaptatif autorise des changements dans les deux sens. Le protocole de changement de mode est le suivant :

- L_0 est le mode initial ;
- Il n'y a pas de mode final ;
- Tous les modes sont ré-entrants ;
- Les transitions entre les modes sont bornées ;
- Suite à un changement de mode, certaines tâches ne s'exécutent que dans l'ancien mode ;
- Certains tâches s'exécutent dans les deux modes, mais leur durée d'exécution (et même leurs périodes) sont variables ;
- Il y a de nouvelles tâches lors du passage d'un mode en direction du mode initial L_0.

Conclusion

Les changements de mode en général et les changements de mode de criticité spécifiquement sont des mécanismes qui permettent de doter un système de la capacité à s'adapter aux modifications de son environnement d'exécution. En ordonnancement multicoeur en particulier, ces mécanismes peuvent permettre à un système temps-réel de poursuivre son exécution en dépit des pannes affectant un ou plusieurs cœurs du processeur. Qu'il s'agisse d'un changement de mode général (présenté en section 1) ou d'un changement de mode de criticité (présenté en section 2), la transition d'un mode à l'autre doit s'opérer sous le contrôle d'un protocole qui garantit le respect des contraintes temporelles. Parmi les protocoles de changement de mode

étudiés, nous n'en avons rencontré aucun qui traite des pannes permanentes en ordonnancement global multicoeur. Les techniques de tolérance proposées dans la deuxième partie de cette thèse apportent une contribution dans ce sens.

Deuxième partie

Contributions

Préambule

Cette partie traite de la construction d'une séquence d'ordonnancement valide d'un système temps-réel multicoeur en dépit de pannes matérielles mettant hors service un ou plusieurs cœurs. Dans ce préambule, nous présentons la modélisation, la démarche, les hypothèses et notations sur lesquelles reposent nos contributions.

Le modèle de tâches

Comme nous l'avons mentionné dans le chapitre 2 (§II.2), en ordonnancement temps-réel multicoeur suivant une politique globale, l'optimalité est obtenue au moyen des algorithmes équitables pour les systèmes composés de tâches périodiques, indépendantes, à départs simultanés et à échéances sur requêtes [42]. Motivés par ce résultat d'optimalité et forts de ce que la condition nécessaire et suffisante ($\sum_{i=1}^{n}(\frac{C_i}{T_i}) \leq m$) est facile à calculer, nous focalisons notre étude sur cette catégorie de systèmes. Rappelons la modélisation temporelle : une tâche τ_i d'un tel système est caractérisée par quatre paramètres dont :

- une date de première activation r_i ;
- une pire durée d'exécution C_i ;
- une échéance relative ou délai critique D_i ;
- une période T_i.

Puisque les tâches du système sont à départs simultanés, elles sont toutes activées au même instant que nous supposons être l'instant 0 (donc $r_i = 0$). Une tâche à échéance sur requête a un délai critique égal à sa période ($D_i = T_i$) et donc toute instance de cette tâche doit terminer son exécution avant l'activation de la prochaine instance. La notation $\tau_i < C_i, T_i >$ est adoptée pour une tâche à échéance sur requête (implicitement $r_i = 0$ et $D_i = T_i$) et la notation $\tau_i < C_i, D_i, T_i >$ pour une tâche à échéance contrainte (implicitement $r_i = 0$ et $D_i \leq T_i$). Nous supposons que le WCET C_i d'une tâche tient compte des coûts de préemption et de migration.

Rappelons par ailleurs que le système est ordonnancé par l'algorithme PD2 qui procède à une division de chaque instance d'une tâche τ_i en C_i sous-tâches unitaires τ_i^j ($j \geq 0$). L'ordre de priorité entre les tâches est dynamiquement défini en fonction des fenêtres d'exécution de leurs sous-tâches. Suivant l'heuristique d'affectation que nous avons choisie, les sous-tâches élues sont affectées dans l'ordre croissant de numérotation des cœurs.

Le modèle de panne

Nous supposons que la plate-forme matérielle d'exécution est composée du nombre minimal m de cœurs qu'il faut pour que le système satisfasse la condition nécessaire et suffisante d'ordonnançabilité. En l'absence de panne, ce nombre minimal peut être obtenu par la formule $m = \lceil U \rceil$. Notre hypothèse est justifiée par le fait que pour des besoins de réduction de coûts financiers, de consommation d'énergie, de poids et de taille, les concepteurs de systèmes temps-réel préfèrent une plate-forme matérielle strictement adaptée à leurs besoins [171, 17].

Soit donc un système multicoeur S s'exécutant sur m cœurs. A tout instant de son exécution, pour des raisons évoquées en introduction générale (§$II.4$), une panne peut affecter temporairement ou durablement un ou plusieurs cœurs. Nous nous intéressons aux pannes permanentes. Dès lors que la panne se produit, les tâches affectées au(x) cœur(s) défaillant(s) ne s'exécutent pas : on dit qu'elle sont impactées. De plus, le nombre de cœurs fonctionnels ne garantit plus l'ordonnançabilité du système, ce qui peut compromettre la validité de la séquence d'ordonnancement. La plupart des travaux cités dans le Chapitre 2 considèrent que la panne est détectée instantanément, ce qui n'est pas toujours possible. Ici, nous supposons que la panne est détectée après un délai de x unités de temps. La détection des pannes est un domaine de recherche à part entière qui sort du cadre de cette thèse. Plusieurs travaux y ont été consacrés à l'instar de [172, 173, 174].

Par hypothèse, le système de détection de la panne et le cœur sur lequel il s'exécute ne font pas partie de notre modèle d'analyse. Nous supposons ainsi l'existence d'une tâche de monitoring qui s'exécute sur un cœur indépendant supposé robuste et infaillible. A chaque unité de temps, cette tâche réalise deux opérations. Dans un premier temps, elle enregistre dans une file de taille x (x étant le délai de détection de la panne) les sous-tâches affectées à chaque cœur. Ensuite, elle met à jour des registres qui renseignent l'ordonnanceur sur l'état de chaque cœur. A cet effet, elle peut utiliser un mécanisme similaire à la fonction "*ping*" connue en réseau informatique, pour vérifier à chaque fois l'état de fonctionnement d'un cœur. Des "ping" sont donc effectués sur chaque cœur à intervalle de temps régulier. Au bout d'un nombre y de tentatives infructueuses, le cœur est considéré comme défaillant. Le délai x de détection de la panne est donc fonction de la fréquence des "ping" sur les cœurs et du nombre y. Dans la suite, nous notons t_p l'instant où la panne se produit, t_d l'instant où elle est détectée et nous nous plaçons dans le pire cas où $t_d = t_p + x$.

En première analyse, nous considèrerons qu'un seul cœur tombe en panne durant toute l'exécution. Comme le montre la Figure 4.3, les sous-tâches affectées à ce cœur entre l'instant t_p et l'instant t_d ne sont pas exécutées : on dit qu'elles sont perdues. Dans le pire des cas, elles sont au nombre de x. Ces sous-tâches peuvent appartenir à la même instance d'une tâche ou à des instances différentes d'une ou de plusieurs tâches. L'ordonnanceur peut accéder à ces sous-tâches via la mémoire locale du cœur de monitoring et procéder à la reconfiguration. On note x_i le nombre de sous-tâches perdues de la tâche τ_i. Soulignons que lorsqu'une instance est impactée par la panne (et donc certaines de ses sous-tâches sont perdues), le compteur ordinal n'évolue pas ; il reste sur la dernière instruction exécutée, car la tâche n'a pas eu accès au cœur défaillant et donc aucune instruction de cette instance n'a été exécutée. C'est pourquoi, les sous-tâches perdues représentent des pertes en temps d'exécution et non des instructions non exécutées.

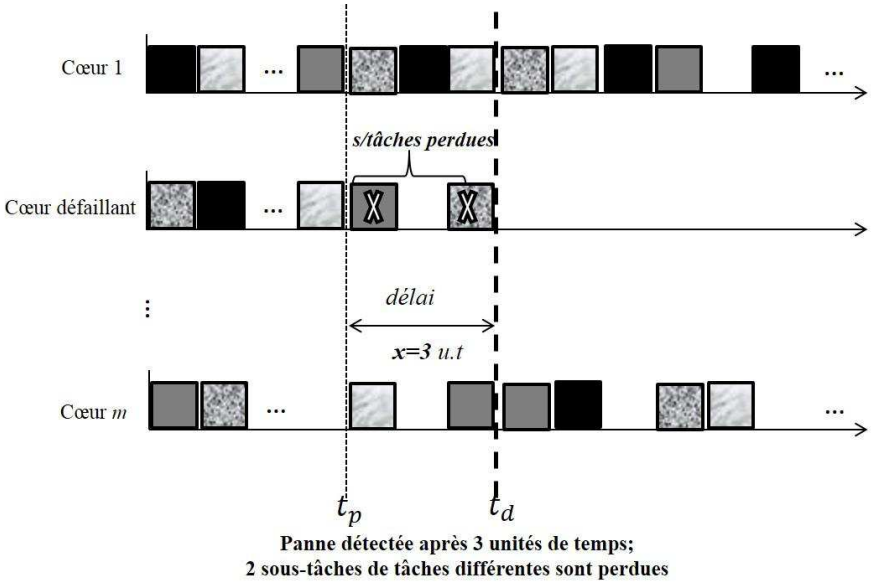

FIGURE 4.3 – Perte d'exécution due à la détection d'une panne.

L'exécution d'une instance affectée reprend après la reconfiguration du système. A cet effet, selon la nature de la tâche et des résultats d'exécution attendus, les concepteurs peuvent envisager l'une des actions suivantes, illustrées par la Figure 4.4 :

- **aucune action** : c'est le cas lorsqu'aucune sous-tâche de l'instance n'est perdue ;
- **interrompre l'instance** : l'exécution de l'instance est arrêtée. Les sous-tâches non exécutées sont abandonnées et les sous-tâches perdues ne sont pas rattrapées. La durée effective d'exécution de l'instance est alors égale au nombre de sous-tâches exécutées avant la panne. L'ordonnancement de la tâche reprendra avec les prochaines instances ;
- **exécuter partiellement l'instance** : l'exécution de l'instance se poursuit jusqu'à la fin du temps qui lui était initialement attribué. Les sous-tâches restantes sont exécutées,

mais les x_i sous-tâches perdues sont abandonnées. L'exécution de l'instance dure donc $C_i - x_i$ unités de temps.

— **exécuter totalement l'instance** : Les sous-tâches restantes sont exécutées et les sous-tâches perdues sont rattrapées pour que l'exécution de l'instance soit complète, c'est-à-dire qu'elle dure effectivement C_i unités de temps comme prévu.

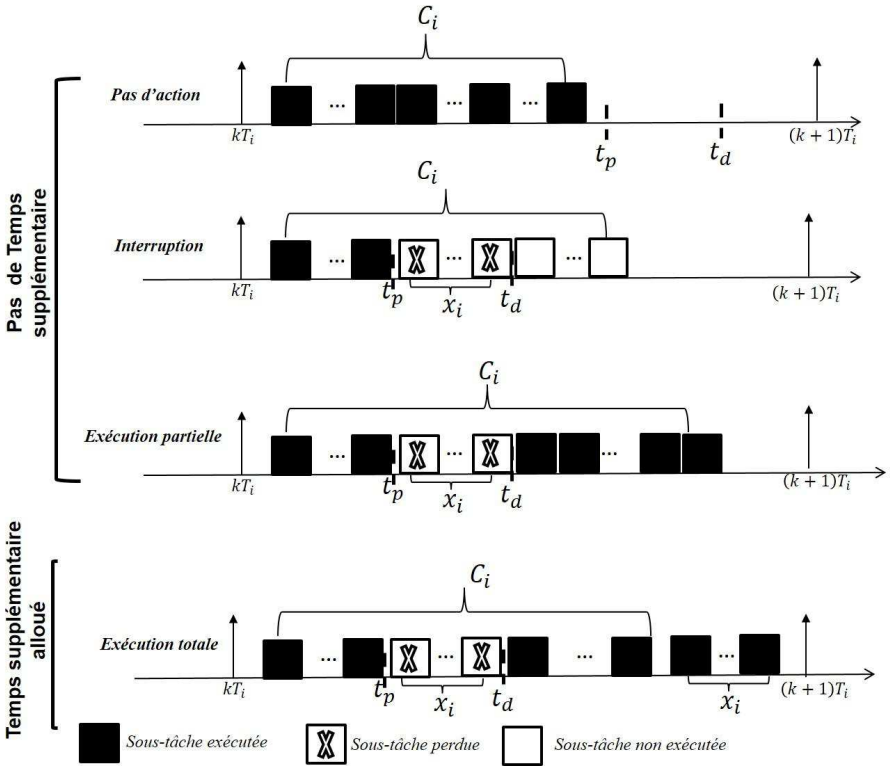

FIGURE 4.4 – Actions envisageables pour une instance impactée par la panne.

Dans les trois premières actions ci-dessus, l'exécution de l'instance dure moins de C_i unités de temps. S'il y a des sous-tâches perdues, aucune allocation de temps supplémentaire n'est envisagée pour les récupérer. On se contente donc d'une exécution inachevée de l'instance.
Par contre, dans la dernière action, x_i unités de temps supplémentaires sont allouées pour rattraper les sous-tâches perdues afin que l'instance termine son exécution.

Dans cette partie, nous étudions dans un premier temps le cas où aucune allocation de temps supplémentaire n'est faite pour récupérer les sous-tâches perdues : on parle d'ordonnancement sans reprise. Ce cas est traité dans le Chapitre 4. Puis, nous étudions le cas où du temps supplémentaire est alloué pour la récupération : on parle d'ordonnancement avec reprise des sous-tâches perdues. Ce cas est traité dans le Chapitre 5. Soulignons qu'il peut arriver que le

délai de détection de la panne soit supérieur à la période d'une tâche impactée $x > T_i$. De ce fait, les sous-tâches perdues peuvent appartenir à plusieurs instances de cette tâche. Dans nos travaux, si le problème se pose, nous supposons que seules les sous-tâches perdues de la dernière instance impactée sont à récupérer.

En dernière analyse, nous considérons que plusieurs cœurs tombent (simultanément ou successivement) en panne. Ce cas est traité dans le Chapitre 6.

Démarches

Les résultats présentés dans cette partie suivent une démarche comportant les trois étapes suivantes :

1. Définition d'une technique de tolérance : la démarche à suivre par l'ordonnanceur Pfair pour produire une séquence valide en dépit de la panne est décrite et l'architecture matérielle nécessaire est déterminée.
2. Expérimentation : la technique est appliquée sur un échantillon de systèmes pour évaluer son efficacité.
3. Preuve : lorsque l'expérimentation est concluante, nous énonçons un résultat de validité dont nous donnons une preuve (totale ou partielle selon les cas).

L'expérimentation des techniques se fait sur un prototype que nous avons conçu, le prototype FTA (FTA : Failure Tolerance Analyser). Ce prototype est composé de trois modules :

— Un générateur de systèmes. Ce module génère des systèmes dont les paramètres des tâches sont choisis de manière aléatoire. Dans un premier temps, la période T_i (et donc le délai critique $D_i = T_i$) est définie à l'aide d'une matrice issue des travaux de Goossens [175] qui permet de limiter l'hyperpériode et donc faciliter l'analyse d'ordonnancement. Ensuite, le WCET T_i des tâches lourdes est choisi selon une loi uniforme dans l'intervalle $[\lceil \frac{T_i}{2} \rceil, T_i]$ et celui des tâches légères dans l'intervalle $[1, \lfloor \frac{T_i}{2} \rfloor - 1]$. Quant à la date de réveil r_i, elle est égale à 0 pour toutes les tâches. Le nombre de tâches dans un système est choisi uniformément entre 5 et 10. A chaque génération, 550 systèmes sont ainsi construits, regroupés en 11 catégories de 50 systèmes. Chaque catégorie correspond à un taux de tâches lourdes (c-à-d telles que $U_i \geq 0.5$) dans le système. La première catégorie correspond à des systèmes où il n'y a aucune tâche lourde, la deuxième à 10% de tâches lourdes, la troisième à 20%, etc., et la onzième à 100% de tâches lourdes. Cette catégorisation permet d'étudier l'influence du facteur d'utilisation des tâches sur le résultat d'ordonnancement.

— Un simulateur. Pour chaque système généré, ce module calcule le nombre minimal de cœurs nécessaires et applique l'ordonnancement PD2 adapté. Au cours de l'ordonnancement, une panne est simulée et le protocole de reconfiguration à tester est appliqué à

l'instant de détection de la panne. L'instant de la panne et le cœur affecté sont aléatoirement choisis.
— Un analyseur. Ce module étudie la séquence générée par le simulateur sur chaque système et détermine si elle est valide (c-à-d toutes instances ont respecté leurs échéances) et si elle est équitable (c-à-d toutes les sous-tâches ont été ordonnancées dans leurs fenêtres d'exécution).

La simulation et l'analyse sont répétées à plusieurs reprises sur l'ensemble des systèmes générés afin de faire varier l'instant de survenue de la panne, le cœur affecté et donc les tâches impactées. Si toutes les séquences générées à l'issue de plusieurs simulations sont valides, la technique est retenue et on passe à l'étape de preuve. Dans le cas contraire, les séquences non valides sont étudiées pour dégager les conditions d'application de ce protocole et l'expérimentation est reprise sous ces conditions.

Notations

Les notations suivantes sont utilisées dans les chapitres de cette partie :

— t_p : instant de survenue de la panne ;
— t_d : instant de détection de la panne ;
— x : délai de détection de la panne (donc $t_d = t_p + x$) ;
— $Next_H$: prochaine hyperpériode après l'instant de détection de la panne ;
— m : nombre de cœurs nécessaire pour l'ordonnançabilité du système ;
— μ : nombre de cœurs additionnels.
— r_i^j : pseudo-date de réveil d'une sous-tâche τ_i^j ;
— d_i^j : pseudo-échéance d'une sous-tâche τ_i^j ;
— $Seq_{[k]}^{(S)}$: Séquence d'ordonnancement du système S sur k cœurs ;
— $Seq_{[k]}^{(S)}[t_1, t_2]$: Séquence d'ordonnancement du système S sur k cœurs durant l'intervalle de temps $[t_1, t_2]$;
— $Seq_{[k \longmapsto_{t_d} r]}^{(S)}$: Séquence générée par l'ordonnancement du système S sur k cœurs au début et puis sur r cœurs après une panne détectée à l'instant t_d, avec $r < k$.
— $Exec(\tau_i^j, Seq)$: instant où la sous-tâche τ_i^j est ordonnancée dans la séquence Seq ;
— $Reveillees(S, t)$: Liste des sous-tâches du système S réveillées à l'instant t (τ_i^j, $r_i^j = t$) ;
— $Pretes_{[k]}(S, t)$: Liste des sous-tâches du système S prêtes à l'instant t dans un ordonnancement sur k cœurs. Les éléments de cette liste sont supposés rangés par ordre de priorité PD2. Une sous-tâche τ_i^j est dite prête à un instant t si elle est réveillée ($r_i^j \leq t$) et non exécutée. De plus, la sous-tâche qui la précède dans l'instance a déjà été exécutée : $Exec(\tau_i^{j-1}, Seq) < t$;
— $Rang(\tau_i^j, L)$: rang occupé par la sous-tâche τ_i^j dans la liste ordonnée L ;
— $Elues_{[k]}(S, t)$: ensemble des sous-tâches du système S élues à l'instant t dans un ordonnancement sur k cœurs ;

— $NonElues_{[k]}(S,t)$: Liste des sous-tâches prêtes du système S non élues à l'instant t dans un ordonnancement sur k cœurs. Dans la liste des sous-tâches prêtes, le rang de toute sous-tâche de cette liste est supérieur au nombre de cœurs ($Rang(\tau_i^j, Pretes_k(S,t)) > k$) ;

Hypothèses

Les résultats présentés dans cette partie s'appuient sur les hypothèses suivantes :

1. La plateforme matérielle initiale (c-à-d en l'absence de panne) est composée de m cœurs identiques ;
2. La condition $\sum_{i=1}^{n}(\frac{C_i}{T_i}) \leq m$ est vérifiée pour tout système. Les systèmes étudiés sont donc ordonnançables sur m coeurs et la séquence $Seq_{[m]}^{(S)}$ est valide et équitable ;
3. Il existe un système de monitoring, indépendant du système étudié et de la plateforme matérielle considérée, qui détecte la panne après un délai maximal de x unités de temps et renseigne l'ordonnanceur PD2 sur les cœurs affectés et les sous-tâches perdues ;
4. Le délai de détection de la panne est inférieur aux périodes des tâches : $\forall \tau_i,\ x < T_i$;
5. A l'instant de détection de la panne, la reconfiguration du système s'opère en un temps négligeable.

En définitive, en s'appuyant sur les modèles, les hypothèses et la démarche ci-dessus présentée, nous étudions dans cette partie les mécanismes de reconfiguration après panne d'un système temps-réel et nous définissons la plateforme matérielle à prévoir pour permettre cette reconfiguration.

Chapitre

5 Ordonnancement sans reprise

Sommaire

Résumé

Ce chapitre traite de la tolérance à la défaillance d'un cœur du processeur, dans le cas où la portion d'exécution éventuellement perdue n'est pas rattrapée. Nous montrons que dans ce cas, l'ajout d'un seul cœur au nombre minimal nécessaire permet de construire une séquence d'ordonnancement valide.

Introduction

Nous avons vu dans le préambule de cette partie que lorsqu'une panne est détectée, une reconfiguration est nécessaire pour construire une séquence d'ordonnancement valide et équitable. Au cours de cette réconfiguration, les concepteurs du système doivent décider d'allouer ou pas du temps supplémentaire pour que l'exécution des instances affectées soit complète. Ce chapitre est consacré au scénario où il n'est pas nécessaire d'allouer du temps supplémentaire et donc de réordonnancer les sous-tâches perdues. Cette option peut être retenue selon la nature de la tâche ou selon ses paramètres temporels. Du point de vue des paramètres temporels, ceci se justifie lorsque l'on sait par exemple que le WCET (pire temps d'exécution) calculé est une borne supérieure très pessimiste de la durée réelle moyenne de la tâche et que dans la plupart des cas on ne l'atteint pas. Dans ce cas, rattraper une sous-tâche qui de toute façon a une grande probablilité d'être vide n'est pas nécessaire. Du point de vue de la nature de la tâche, on peut se satisfaire d'une exécution partielle s'il est estimé que la perte de tout ou partie de l'exécution d'une instance de tâche n'altère pas considérablement la qualité du résultat. A titre d'exemple, on peut citer le cas d'une tâche de lecture d'un flux multimédia dont on estime que la perte d'une instance ne dégrade pas le rendu visuel. On peut aussi citer le cas des tâches itératives qui calculent un résultat en procédant par plusieurs itérations. Au bout d'un nombre d'itérations donné, on considère que le résultat obtenu est moins précis que si la tâche avait été jusqu'au bout de son calcul, mais qu'il est acceptable et donc l'exécution partielle de la tâche ne causera pas de dysfonctionnement.

Lorsque toute l'instance affectée est abandonnée (c-à-d qu'on abandonne les sous-tâches restantes et les sous-tâches perdues), les sous-tâches non encore exécutées de cette instance ne sont pas ordonnancées. Elles donnent lieu à des unités de temps creux supplémentaires qui peuvent être allouées aux autres sous-tâches prêtes du système qui s'exécuteront alors plus tôt que si l'exécution de ladite instance était poursuivie. Les dépassements d'échéance sont ainsi évités. Par contre, si seules les sous-tâches perdues de l'instance sont abandonnées, on a une marge de manœuvre plus étroite car les sous-tâches restantes de l'instance affectées sont ordonnancées. Il pourrait dans ce cas y avoir risque de dépassement d'échéance. Nous allons montrer qu'il n'en est rien.

Dans ce chapitre, nous nous plaçons dans ce cas et donc nous considérons que seules les sous-tâches perdues d'une instance sont abandonnées. Cependant, l'approche proposée s'applique aussi au cas de l'abandon complet de l'instance. C'est ainsi que dans la première section, nous proposons la technique de redondance matérielle limitée qui consiste à ordonnancer le système sur une architecture matérielle comprenant un cœur de plus que le minimum nécessaire m. Nous montrons alors dans la deuxième section que cette technique permet d'obtenir une séquence d'ordonnancement valide.

I. Technique de la redondance matérielle limitée

Soit un système temps-réel multicoeur ordonnançable sur m coeurs. Dans cette partie, nous supposons que $m = \lceil U \rceil$. Si une panne permanente met hors service l'un des coeurs, la validité de la séquence produite est compromise car le nombre de coeurs fonctionnels ne gararantit plus l'ordonnançabilité du système. Pour pallier une telle éventualité, la *Technique de la Redondance Matérielle Limité (TRML)* consiste à prévoir un coeur de plus que nécessaire (donc $\mu = 1$). De ce fait, comme l'illustre la Figure 5.1, l'ordonnancement du système débute non pas sur m coeurs, mais sur $m + 1$ coeurs. Lorsque la panne survient, l'ordonnancement se poursuit sur les m cœurs restants. Cette technique repose sur une redondance active où le cœur additionnel ne reste pas inactif au début de l'ordonnancement et donc exécute les tâches qui lui sont affectées. Son intérêt est double : d'une part, elle assure que la condition nécessaire et suffisante d'ordonnançabilité reste satisfaite après la panne et d'autre part, elle permet d'optimiser la charge traitée (c-à-d le nombre de sous-tâches ordonnancées) avant la panne et donc d'alléger la charge restante du système après la panne.

Étant donné que la réconfiguration s'opère uniquement au niveau matériel et qu'aucune modification n'est opérée au niveau des tâches, le système s'exécute dans le même mode avant et après la panne : il n'y a pas de changement de mode.
Le choix de la technique *TRML* par des concepteurs d'un système temps-réel multicoeur peut donc être motivé par le souci de pouvoir supporter la défaillance d'un cœur en engageant des coûts financiers et énergétiques réduits.

I.1. Algorithme d'ordonnancement

En nous appuyant sur la description formelle introduite au Chapitre 2 (§III.2), nous proposons ci-dessous l'algorithme d'ordonnancement PD2 avec panne sans reprise des sous-tâches impactées par la technique *TRML*. L'ordonnancement étant cyclique (tâches périodiques à départs simultanés), cet algorithme défini pour la première hyperpériode est itéré sur les hyperpériodes suivantes. La liste des primitives est complétée par la primitive *detection*(). A chaque appel, cette primitive consulte les registres du système de détection où sont enregistrés les états des cœurs du processeur et retourne le numéro du cœur défaillant s'il y en a un et -1 sinon. Rappelons qu'ici, on considère un seul cœur défaillant.

PRIMITIVES

- $sousTaches(S)$: retourne la liste des sous-tâches du système S sur une hyperpériode ;
- $pretes(ST, t)$: retourne la liste des sous-tâches de S prêtes à l'instant t, ST étant l'ensemble des sous-tâches de S ;
- $card(L)$: retourne le nombre d'éléments de la liste L ;
- $ranger(L)$: range les sous-tâches de la liste L par ordre de priorité PD2 décroissante ;
- $init(Seq)$: initialise la séquence Seq par des unités de temps creux.

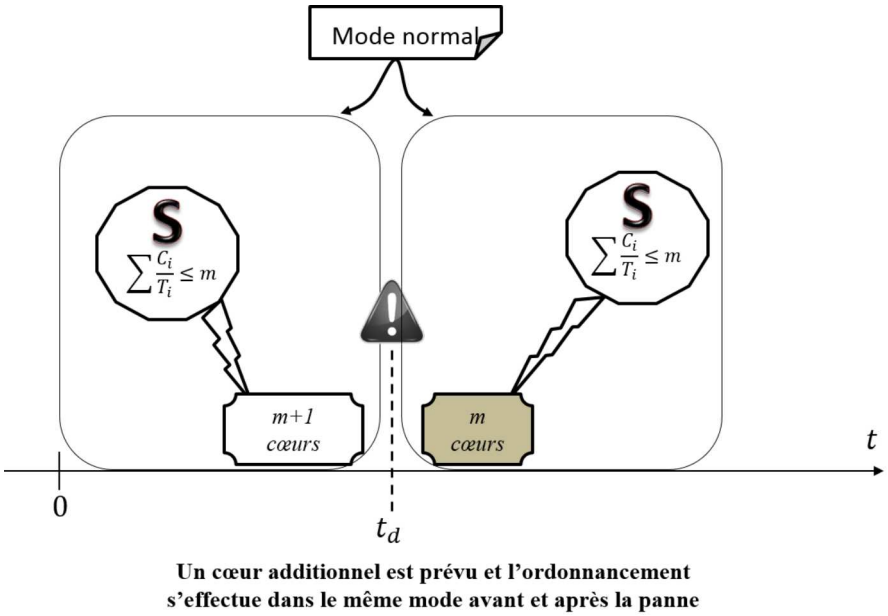

FIGURE 5.1 – Illustration de la technique de la redondance matérielle limitée.

— *detection*() : retourne le cœur défaillant ou -1 s'il n'y en a aucun.

I.2. Exemple

Nous considérons le système Ψ du chapitre 2 ($\S III.2$)
$\Psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$.
La charge du système est $U = 26/12 = 2.16$ (donc $m = 3$ cœurs) et l'hyperpériode $H = ppcm(3, 4, 6, 12) = 12$. En application de la technique de la redondance matérielle limitée, l'ordonnancement débutera sur 4 cœurs. Supposons à présent qu'une panne affecte le coeur $C2$ à l'instant $t_p = 1$. La panne est détectée, après $x = 2$ unités de temps, donc à l'instant $t_d = 3$. Les sous-tâches τ_5^1 et τ_4^1 affectées au coeur $C2$ respectivement aux instants $t_1 = 1$ et $t_2 = 1$ n'ont pas été exécutées. Comme le montre la Figure 5.2, l'ordonnancement se poursuit sur les 3 cœurs restants et les sous-tâches perdues ne sont pas reprises.

Le tableau 5.1 rappelle les fenêtres d'exécution de chaque sous-tâche et présente les instants d'exécution de chacune d'elles. On peut constater que chaque sous-tâche s'exécute dans sa fenêtre et donc la séquence construite est valide et équitable.

Algorithm 2: Ordonnancement PD2 avec panne sans reprise des sous-tâches perdues

Inputs : $S : \tau_i < C_i, T_i >$: système à ordonnancer ;
$C = (C_1, C_2, .., C_{m+1})$: liste des cœurs du processeur.

Output: $Seq = Seq^{(S)}_{[(m+1)\longmapsto t_d(m)]}$: Séquence d'ordonnancement du système S sur $m+1$ puis sur m cœurs.

```
init(Seq);
t ← 0 ;
ST ← sousTaches(S);
while t < H do // i.e à tout instant t
    P ← pretes(ST, t) ;
    ranger(P);
    num ← detection() ;
    k ← 1 ;
    j ← 1 ;
    while (k ≤ m + 1) ∧ (j ≤ card(P)) do
        if k ≠ num then
            Seq(C(k), t) ← P(j) ;
            j ← j+1 ;
        k ← k+1 ;
    t ← t+1 ;
```

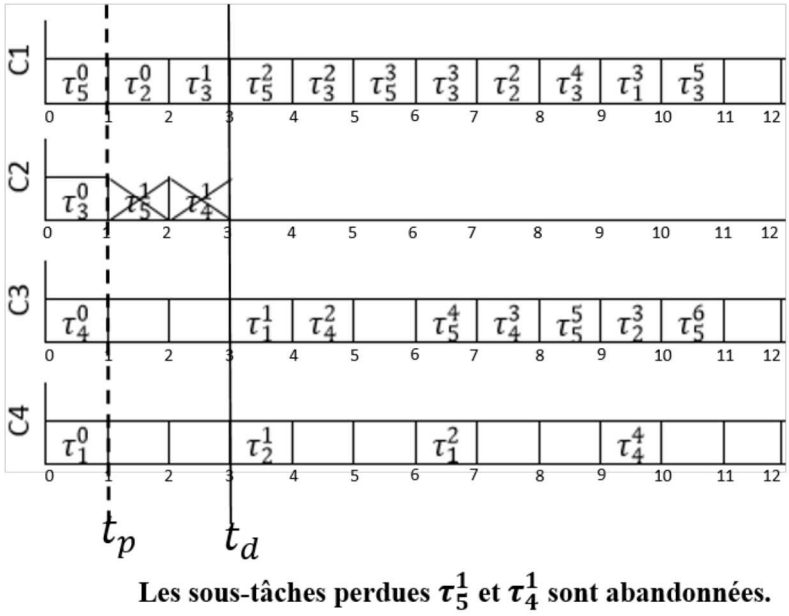

FIGURE 5.2 – Séquence d'ordonnancement de ψ avec $t_p = 1$ et $x = 2$.

I.3. Expérimentation

Afin de valider la technique *TRML*, une étude expérimentale a été conduite avec le prototype FTA. Comme mentionné dans le préambule, 550 systèmes générés aléatoirement ont été soumis 10 fois à la simulation (afin de faire varier l'instant de la panne et le cœur affecté) pour un délai de détection $x = 2$ unités de temps. Les résultats obtenus montrent que quels que soient la charge du système et le taux de tâches lourdes le constituant, peu importent l'instant de survenue de la panne, le cœur affecté ou la tâche impactée, l'utilisation de la redondance matérielle limitée assure la construction d'une séquence d'ordonnancement valide. Dans la section suivante, nous prouvons ce résultat.

TABLE 5.1 – Instants d'ordonnancement des sous-tâches du système Ψ par la technique TRML.

τ_i	τ_i^j	r_i^j	d_i^j	$Exec(\tau_i^j)$
τ_1	τ_1^0	0	3	0
	τ_1^1	3	6	3
	τ_1^2	6	9	6
	τ_1^3	9	12	9
τ_2	τ_2^0	0	3	1
	τ_2^1	3	6	3
	τ_2^2	6	9	7
	τ_2^3	9	12	9
τ_3	τ_3^0	0	2	0
	τ_3^1	2	4	2
	τ_3^2	4	6	4
	τ_3^3	6	8	6
	τ_3^4	8	10	8
	τ_3^5	10	12	10
τ_4	τ_4^0	0	3	0
	τ_4^1	2	5	-
	τ_4^2	4	8	4
	τ_4^3	7	10	7
	τ_4^4	9	12	9
τ_5	τ_5^0	0	2	0
	τ_5^1	1	4	-
	τ_5^2	3	6	3
	τ_5^3	5	7	5
	τ_5^4	6	9	6
	τ_5^5	8	11	8
	τ_5^6	10	12	10

II. Preuve de la validité de la technique

Dans cette section, nous prouvons que pour tout système ordonnançable avec PD2 sur m cœurs, si du temps supplémentaire n'est pas alloué pour rattraper l'exécution perdue du fait de la panne, l'utilisation de la technique de la redondance matérielle limitée assure un ordonnancement valide et équitable.

II.1. Résultat d'ordonnancement

Notre résultat principal est présenté dans le théorème suivant :

Théorème 1. ***Soit un système*** S ***composé de n tâches indépendantes et périodiques, à départs simultanés et à échéances sur requête ordonnançable avec l'algorithme PD2 sur*** m ***coeurs. Si*** S ***est exécuté sur*** $m+1$ ***cœurs et qu'un cœur tombe en panne, dans le cas où les sous-tâches perdues ne sont pas récupérées, la séquence d'ordonnancement construite*** $Seq^{(S)}_{[(m+1)\longmapsto_{tp} m]}$ ***est valide.***

$$\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{tp}(m)]}) < d_i^j \ \blacksquare$$

II.2. Lemmes utilisés

Le principe de la preuve du théorème consiste à montrer qu'une tâche n'est jamais ordonnancée plus tard avec $m+1$ puis m cœurs qu'avec m coeurs. Pour cela, deux lemmes sont nécessaires. Dans un premier temps, on montre dans le Lemme 1 que, à tout instant t, si une sous-tâche est prête dans l'ordonnancement du système sur $m+1$ cœurs, alors dans un ordonnancement sur m cœurs, elle n'a pas déjà été ordonnancée. Ensuite ce lemme est utilisé pour démontrer la même propriété dans le Lemme 2 pour un ordonnancement du système sur $(m+1) \longmapsto_{t_d} (m)$ cœurs.

Lemme 1. ***A tout instant*** t***, toute sous-tâche prête dans l'ordonnancement d'un système S sur*** $m+1$ ***cœurs n'a pas déjà été ordonnancée dans*** $Seq^{(S)}_{[m]}$***.***

$$\forall t,\ \tau_i^j \in Pretes_{[m+1]}(S,t) \Rightarrow Exec(\tau_i^j, Seq^{(S)}_{[m]}) \geq t \ \blacksquare$$

Démonstration. Nous procédons par récurrence sur l'instant t.

— Le lemme est vérifié à l'instant $t = 0$.
En effet, à l'instant $t = 0$, les sous-tâches prêtes sont les sous-tâches réveillées et donc : $Pretes_{[m+1]}(S,0) = Pretes_{[m]}(S,0) = Reveillees(S,0)$.
Par conséquent, à l'instant 0, toute sous-tâche prête dans l'ordonnancement du système sur $m+1$ cœurs n'a pas encore été ordonnancée dans $Seq^{(S)}_{[m]}$, car elle est nouvellement réveillée.

— Supposons qu'à un instant $t-1 \geq 0$ le lemme soit vérifié et donc que :
$\tau_i^j \in Pretes_{[m+1]}(S, t-1) \Rightarrow Exec(\tau_i^j, Seq_{[m]}^{(S)}) \geq t-1$.

— Montrons que le lemme est vérifié à l'instant t.
Soit $\tau_i^j \in Pretes_{[m+1]}(S,t)$. Trois cas sont possibles : soit τ_i^j est juste réveillée à l'instant t et sa précédente τ_i^{j-1} a déjà été ordonnancée dans $Seq_{[m+1]}^{(S)}$, soit τ_i^j était prête à l'instant $t-1$ mais n'a pas été élue, soit enfin à l'instant $t-1$, τ_i^j était réveillée mais sa précédente τ_i^{j-1} n'avait pas encore été ordonnancée. Nous montrons que le lemme est vérifié dans chacune de ces éventualités.

✓ Premier cas : τ_i^j est réveillée à l'instant t et τ_i^{j-1} a déjà été ordonnancée dans $Seq_{[m+1]}^{(S)}$. τ_i^j est aussi réveillée à l'instant t dans l'ordonnancement du système sur m cœurs et donc n'a pas déjà été ordonnancée dans $Seq_{[m]}^{(S)}$ à l'instant t. Le lemme est donc vérifié.

✓ Deuxième cas : à l'instant $t-1$, τ_i^j était prête et n'avait pas été ordonnancée dans $Seq_{[m+1]}^{(S)}$.
Dans ce cas, $Rang(\tau_i^j, Pretes_{[m+1]}(S,t-1)) > m+1$ et donc il y a $m+1$ sous-tâches $\tau_{u_1}^{v_1}\tau_{u_2}^{v_2}..\tau_{u_{m+1}}^{v_{m+1}}$ élues dans $Seq_{[m+1]}^{(S)}$ qui sont plus prioritaires que τ_i^j.
D'après l'hypothèse de récurrence, à l'instant $t-1$, puisque τ_i^j est prête dans l'ordonnancement du système sur $m+1$ cœurs, τ_i^j n'a pas déjà été ordonnancée dans $Seq_{[m]}^{(S)}$. A cet instant, dans l'ordonnancement du système sur m cœurs, les $m+1$ sous-tâches τ_u^v n'ont pas encore été ordonnancées (hypothèse de récurrence). Elles sont donc soit prêtes, et alors plus prioritaires que τ_i^j, soit dans l'attente de l'ordonnancement de leurs précédentes τ_u^{v-1} qui alors sont prêtes et plus prioritaires que τ_i^j. Donc, à l'instant $t-1$, il existe au moins $m+1$ sous-tâches plus prioritaires que τ_i^j et donc τ_i^j sera ordonnancée au plus tôt à l'instant t dans $Seq_{[m]}^{(S)}$. Le lemme est vérifié.

✓ Troisième cas : à l'instant $t-1$, τ_i^j est réveillée mais τ_i^{j-1} n'a pas encore été ordonnancée dans $Seq_{[m+1]}^{(S)}$.
Dans l'ordonnancement du système sur m cœurs, à l'instant $t-1$, τ_i^j est aussi réveillée et d'après l'hypothèse de récurrence, τ_i^{j-1} n'a pas déjà été ordonnancé. Par conséquent, τ_i^j n'est pas une sous-tâche prête à l'instant $t-1$ et donc sera ordonnancée dans $Seq_{[m]}^{(S)}$ au plus tôt à l'instant t. Le lemme est vérifié.

□

Lemme 2. ***A tout instant*** $t \geq t_d$***, toute sous-tâche prête dans l'ordonnancement d'un système*** S ***sur*** $(m+1) \longmapsto_{t_p} (m)$ ***cœurs n'a pas déjà été ordonnancée dans*** $Seq_{[m]}^{(S)}$**.**

$$\forall t \geq t_d,\ \tau_i^j \in Pretes_{[(m+1)\longmapsto_{t_d}(m)]}(S,t) \Rightarrow Exec(\tau_i^j, Seq_{[m]}^{(S)}) \geq t \ \blacksquare$$

Démonstration. Nous utilisons le même raisonnement de récurrence que dans le Lemme 1.

— Ce lemme est vérifié à l'instant $t = t_d$.
En effet, soit $\tau_i^j \in Pretes_{[(m+1)\longmapsto_{t_d}(m)]}(S,t)$. Avant l'instant t_d, on a $Elues_{[(m+1)\longmapsto_{t_d}(m)]}(S,t) = Elues_{[m+1]}(S,t)$ et donc on a $\tau_i^j \in Pretes_{[(m+1)\longmapsto_{t_d}(m)]}(S,t_d) = Pretes_{[m+1]}(S,t_d)$. On déduit du Lemme 1 que τ_i^j n'a pas déjà été ordonnancée dans $Seq_{[m]}^{(S)}$.

— Supposons qu'à un instant $t-1 \geq t_d$ le lemme soit vérifié :
$\tau_i^j \in Pretes_{[(m+1)\longmapsto_{t_d}(m)]}(S,t-1) \Rightarrow Exec(\tau_i^j, Seq_{[m]}^{(S)}) \geq t-1$.

— Pour montrer qu'à l'instant t ce lemme est vérifié, nous utilisons exactement la même preuve que pour le Lemme 1.

Le Lemme 2 est donc vérifié à l'instant t.

□

II.3. Preuve du résultat

Démonstration. (Théorème 1)
Montrons que : $\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_p}m]}^{(S)}) < d_i^j$.
Pour cela, nous procédons par disjonction de cas, puis par l'absurde.

Rappelons que le système S étant ordonnançable par hypothèse sur m cœurs, il l'est aussi sur (m+1) coeurs et donc les séquences $Seq_{[m]}^{(S)}$ et $Seq_{[m+1]}^{(S)}$ sont valides et équitables.
De plus, une sous-tâche n'est ordonnancée que si elle est reveillée et donc
$\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)})$. Il reste à montrer que $\forall \tau_i^j, Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_p}m]}^{(S)}) < d_i^j$.

A cet effet, considérons une sous-tâche quelconque τ_i^j. Dans la séquence $Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)}$, deux cas sont envisageables : soit τ_i^j est ordonnancée avant l'instant t_d, soit τ_i^j est ordonnancée à partir de l'instant t_d.

— Cas 1 : τ_i^j est ordonnancée avant la panne : $Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)}) < t_d$.
A chaque instant $t < t_d$, il y a jusqu'à $m+1$ sous-tâches élues dans les deux ordonnancements et on a $Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)}[0,t_d] = Seq_{[m+1]}^{(S)}[0,t_d]$ qui est valide et équitable. D'où $Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)}) = Exec(\tau_i^j, Seq_{[m+1]}^{(S)}) < d_i^j$.
Le Théorème 1 est vérifié.

— Cas 2 : τ_i^j est ordonnancée après la détection de la panne $Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)}) \geq t_d)$.
Raisonnons par l'absurde en supposant que $d_i^j \leq Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}m]}^{(S)})$.
D'après le Lemme 2, $\tau_i^j \in Pretes_{[(m+1)\longmapsto_{t_d}(m)]}(S,t) \Rightarrow Exec(\tau_i^j, Seq_{[m]}^{(S)}) \geq t$.

Par conséquent, si $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_d} m]}) = t$ alors $Exec(\tau_i^j, Seq^{(S)}_{[m]}) \geq t$ et donc, $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_d} m]}) \leq Exec(\tau_i^j, Seq^{(S)}_{[m]})$.
Il en résulte que $d_i^j \leq Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_d} m]})$ entraine $d_i^j \leq Exec(\tau_i^j, Seq^{(S)}_{[m]})$, ce qui signifie que τ_i^j n'est pas ordonnancée dans sa fenêtre dans $Seq^{(S)}_{[m]}$ (qui pourtant est valide et équitable) et donc cela contredit l'hypothèse de départ.
Conclusion : $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_p} m]}) < d_i^j$. Le Théorème 1 est vérifié.

□

Conclusion

En définitive lorsqu'il n'est pas nécessaire d'allouer du temps supplémentaire pour rattraper la portion d'exécution perdue, l'ordonnancement du système avec un cœur de plus que nécessaire génère une séquence valide et équitable. Cette technique de redondance matérielle limitée présentée et illustrée dans ce chapitre a été expérimentée et prouvée. La démonstration repose sur le fait que dans l'ordonnancement d'un système avec la technique *TRML*, à tout instant t, une sous-tâche prête n'a pas déjà été ordonnancée dans la séquence d'ordonnancement dudit système sur m cœurs. La preuve de la technique *TRML* sert de base à la validation de certaines techniques de tolérance proposées dans les chapitres suivants qui traitent du cas où la reprise de l'exécution perdue est obligatoire.

Chapitre

6

Ordonnancement avec reprise

Sommaire

Résumé

Ce chapitre propose trois techniques d'ordonnancement d'un système temps-réel multi-coeur qui subit en cours d'exécution la défaillance d'un cœur et dont la portion d'exécution perdue est rattrapée.

Introduction

Lorsqu'au cours de l'ordonnancement du système une portion du temps prévue pour l'exécution peut être perdue du fait de la détection tardive d'une panne, les concepteurs doivent décider en fonction de l'importance des tâches, s'il faut ou non allouer du temps supplémentaire pour récupérer cette perte. Le cas où cette allocation n'est pas nécessaire a été traité au chapitre précédent. Nous avons alors prouvé que la technique de la redondance matérielle limitée garantit la validité de la séquence d'ordonnancement. Le présent chapitre traite du cas où une telle allocation s'impose. Ce scénario est envisageable pour des systèmes dont les tâches réalisent des traitements précis. Une exécution incomplète des tâches produit des résultats incorrects qui, dans certains cas, peuvent s'avérer catastrophiques. Par exemple, pour un système automobile composé de tâches de contrôle de vitesse et des tâches de contrôle du conducteur, une exécution partielle de ces tâches peut entrainer des erreurs d'appréciation et donc une prise de décision pouvant provoquer un accident. De manière générale, dans les systèmes ordonnancés par un algorithme équitable où une exécution partielle des tâches n'est pas admise, les sous-tâches perdues suite à une panne doivent être réordonnancées afin que toutes les instances des tâches s'exécutent complètement. De ce fait, on se retrouve face à une double problématique, à savoir : comment assurer que le système reste ordonnançable après la panne et comment garantir la validité et l'équité de l'ordonnancement étant donné que la reprise des sous-tâches perdues peut engendrer des fautes temporelles ?

Pour répondre à la première question, la technique de la redondance limitée est une fois de plus utilisée, et donc toutes les approches développées dans ce chapitre reposent sur cette technique. Un nombre limité de cœurs additionnels est prévu, donc la tolérance s'opère par une action au niveau matériel. Pour répondre à la deuxième question, il faut ajouter des actions au niveau du système. Ces actions visent à assurer que le réordonnancement des sous-tâches perdues est possible et qu'il n'engendre pas de dépassement de pseudo-échéance par les autres sous-tâches du système. A cet effet, nous proposons trois techniques. La première technique, dénommée *Technique des Sous-tâches de Substitution (TSS)*, consiste à prévoir dans chaque instance de tâche, des sous-tâches additionnelles qui pourront être utilisées en remplacement des sous-tâches perdues. Cette technique est présentée à la Section I. La deuxième technique (présentée en Section II) est la *Technique Contraindre puis Relâcher (TCR)*. Elle vise à créer une marge de temps dans chaque instance qui pourra être utilisée pour réordonnancer les sous-tâches perdues. Cette marge est créée entre le délai critique et la période de la tâche. Pour déterminer les nouveaux délais critiques des tâches deux méthodes sont proposées : la méthode des sous-tâches fantômes et la méthode de distribution des temps creux. La Section III est dédiée à la présentation de la *Technique du Flux Apériodique (TFA)* inspirée des travaux de Choquet-Geniet et Malo [176]. Elle consiste à réordonnancer à l'aide d'un flux apériodique les sous-tâches perdues dans les unités de temps creux identifiées.

Les deux premières techniques (*TSS* et *TCR*) reposent sur la modification des paramètres temporels des tâches du système. C'est ainsi que les WCET (C_i) sont augmentés dans TSS alors que les délais critiques (D_i) sont réduits dans TCR. A la différence de ces deux techniques, la technique *TFA* conserve les paramètres des tâches. Cependant, elle ajoute au système un flux apériodique. De plus, dans la technique *TSS* le système s'exécute dans le même mode tout au long de l'ordonnancement alors que dans les techniques *TCR* et *TFA*, des changements de mode s'opèrent au moment de la détection de la panne pour adapter l'exécution du système à la nouvelle configuration matérielle. Ces changements de mode apportent plus de flexibilité. Enfin, les deux premières techniques visent à assurer le respect des échéances des tâches en dépit de la panne. Cependant, elles ne s'appliquent pas à tous les systèmes ordonnançables. La technique *TFA* s'applique à tout système ordonnançable mais autorise un nombre borné de dépassements d'échéances. Nous montrons dans ce cas que la récupération s'effectue en un temps borné.
La comparaison entre ces trois techniques présentée à la Section IV résume les caractéristiques de chacune d'elles, ce qui permet d'orienter le choix des concepteurs. Enfin, le chapitre se termine par la présentation d'une technique mixte qui permet de différencier les traitements en fonction de la nature des tâches.

I. La Technique des Sous-tâches de Substitution

Soit $S : \{\tau_i < C_i, T_i >, i = 1..n\}$ un système temps-réel ordonnançable avec l'algorithme PD2 sur m coeurs, avec $m = \lceil U \rceil$. Considérons une panne affectant l'un des coeurs détectée à un instant t_d, x unités de temps après sa survenue. Du fait de ce délai de détection, au plus x sous-tâches d'une même tâche peuvent être perdues. En prévision du ré-ordonnancement de ces sous-tâches, l'idée principale de cette technique est d'ajouter x sous-tâches à chaque instance de tâche. Ces sous-tâches sont appelées *sous-tâches de substitution*, car elle seront utilisées en remplacement des sous-tâches perdues. Comme le montre la Figure 6.1, les paramètres temporels du système sont ainsi modifiés et donc chaque tâche a désormais un WCET égal à $C_i + x$ au lieu de C_i comme à l'origine. Ceci correspond à C_i sous-tâches réelles et x sous-tâches de substitution par instance. Nous notons $\tau_i^j (j \geq 0)$ une sous-tâche réelle d'une tâche τ_i et $s_i^{k,h} (h \in [1, x])$ les sous-tâches de substitution de sa k^e instance. Les deux types de sous-tâches (réelles et de substitution) sont ordonnancées au cours de l'exécution du système. L'ajout de sous-tâche de substitution n'est possible que si les paramètres temporels du système respectent la condition :

$$\forall \tau_i,\ C_i + x \leq T_i. \tag{6.1}$$

Il en résulte que lorsqu'une panne est détectée, les sous-tâches réelles perdues d'une tâche appartiennent nécessairement à une même instance.

Avant la panne, les instructions d'une instance d'une tâche τ_i sont complètement exécutées après l'allocation de C_i unités de temps processeur à la tâche (c-à-d après l'exécution de toutes

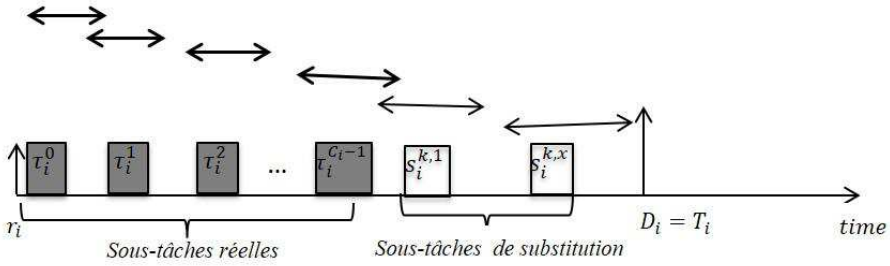

FIGURE 6.1 – Prévision de sous-tâches de substitution pour remplacer les sous-tâches perdues

ses sous-tâches réelles). De ce fait, ses sous-tâches de substitution représentent des unités de temps creux (c-à-d des temps d'inactivité) pour les cœurs auxquels elles sont affectées, car aucune instruction ne s'exécute lors de leur ordonnancement. Après la détection de la panne, si x_i sous-tâches réelles sont perdues dans une instance, les x_i prochaines sous-tâches de substitution se transforment en sous-tâches réelles pour achever l'exécution de l'instance avant son échéance. En ce qui concerne les instances non affectées des tâches impactées et les instances des autres tâches, les sous-tâches de substitution restent des unités de temps creux.
Ainsi, comme le présente la Figure 6.2, le système initial à exécuter $S : \{\tau_i < C_i, T_i >, i = 1..n\}$ est transformé en un système de substitution $S_x : \{\tau_i^+ < C_i + x, T_i >, i = 1..n\}$ qui s'exécute dans un même mode avant et après la panne. Pour être ordonnançable, S_x doit satisfaire la condition nécessaire et suffisante de l'équation (1.2) de la page 25. Il faut donc que $U_x = \sum_{i=1}^{n} \left(\frac{C_i + x}{T_i} \right) \leq m_x$. Étant donné que nous considérons une plateforme d'exécution minimale, on a $m_x = \lceil U_x \rceil$. En application de la redondance matérielle limitée, le système S_x débutera son exécution sur $m_x + 1$ cœurs et la poursuivra sur m_x cœurs si une panne survient. Comme nous pouvons le constater, le nombre de cœurs additionnels utilisés par cette technique est $\mu = m_x + 1 - m$.

I.1. Algorithme d'ordonnancement

L'Algorithme 3 effectue l'ordonnancement PD2 d'un système par la technique TSS.

I.2. Exemple

Considérons le système $\psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$ avec $x = 2$ et $t_d = 3$. Le système de substitution obtenu par la technique *TSS* est
$\psi_x : \tau_1^+ < 3, 3 >, \tau_2^+ < 4, 6 >, \tau_3^+ < 4, 4 >, \tau_4^+ < 7, 12 >, \tau_5^+ < 9, 12 >$.
Il a une charge de $U_{\psi_x} = 4$ et donc débutera son exécution sur 5 cœurs en application de la redondance matérielle limitée. La Figure 6.3 montre la séquence d'ordonnancement obtenue dans laquelle les sous-tâches réelles τ_5^1 et τ_5^2 appartenant à la première instance de la tâche τ_5

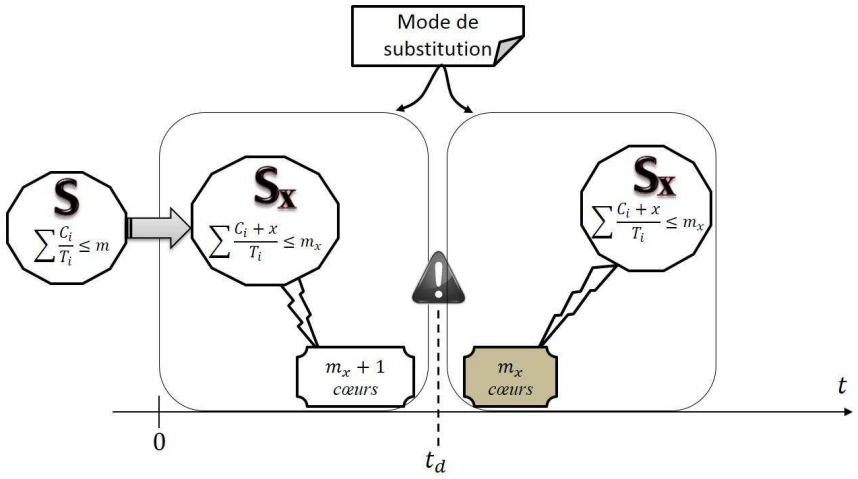

FIGURE 6.2 – Ordonnancement PD2 avec panne et avec reprise par la technique TSS

Algorithm 3: Ordonnancement PD2 avec panne et reprise par la technique TSS

Inputs : $S : \tau_i < C_i, T_i >$: système à ordonnancer ;
x : délai de détection de la panne.

1. Construire à partir de S le système $S_x : \tau_i < C_i + x, T_i >$;
2. $m_x \leftarrow \left\lceil \sum_{i=1}^{n} \left(\frac{C_i + x}{T_i} \right) \right\rceil$;
3. Ordonnancer S_x sur m_x cœurs avec l'Algorithme 2 (qui démarrera donc sur $m_x + 1$ cœurs);

sont perdues. Elles sont remplacées par les sous-tâches de substitution $s_5^{1,1}$ et $s_5^{1,2}$, ce qui permet à toutes les instructions de cette instance de s'exécuter.

I.3. Expérimentation et Preuve

Au cours de l'expérimentation de cette technique avec le prototype FTA aucune faute temporelle n'a été enregistrée dans les systèmes générés. Cela se justifie par le fait qu'en réalité, l'ordonnancement du système $S : \{\tau_i < C_i, T_i >, i = 1..n\}$ avec panne d'un cœur et ré-exécution de x sous-tâches perdues se ramène à l'ordonnancement du système $\{S_x : \tau_i^+ < C_i + x, T_i >, i = 1..n\}$ avec panne et sans reprise des sous-tâches perdues. En effet, dans le système S_x, les sous-tâches perdues sont tout simplement abandonnées et l'exécution complète des tâches impactées est réalisée par les sous-tâches de substitution dont l'ordonnancement était déjà prévu. Tout se passe donc comme si les sous-tâches de substitution étaient renommées en sous-tâches perdues.

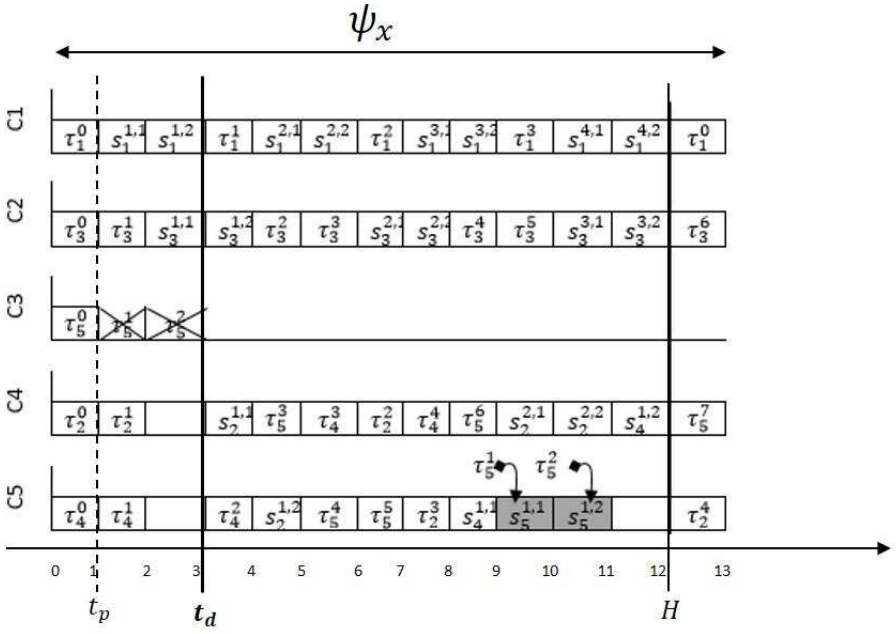

FIGURE 6.3 – Exemple d'ordonnancement PD2 par la Technique TSS

La preuve élaborée dans le chapitre précédent pour l'ordonnancement sans reprise s'applique à la présente technique. On conclut alors que l'ordonnancement du système S_x et par conséquent celui du système S par la technique TSS sont valides.

L'intérêt de la technique TSS est qu'elle est facile d'utilisation et d'implémentation. Elle assure que si un cœur tombe en panne au cours d'exécution, il y a toujours suffisamment de temps additionnel pour rattraper l'exécution perdue sans causer de faute temporelle. Le bon fonctionnement de l'application est donc garanti. La technique peut s'appliquer à tout système ordonnançable respectant la condition de l'équation (5.1). Cependant, le prix à payer est la surcharge du système à ordonnancer qui est due à la prévision de plus de sous-tâches additionnelles que nécessaire. En effet, il y a au total $N = \sum_{i=1}^{n}(\frac{H}{T_i})$ instances de tâches dans une hyperpériode et donc $x \times N$ sous-tâches sont ajoutées pour la tolérance alors qu'il y a au plus x sous-tâches perdues. Ce qui fait une marge de $x \times (N-1)$ sous-tâches ordonnancées qui ne sont pas utilisées pour la tolérance et qui ne représentent rien d'autre que des temps creux. Dans l'exemple du système Ψ, 22 sous-tâches de substitution sont ajoutées pour seulement 2 sous-tâches perdues, ce qui entraine le passage d'une charge à traiter de 2.16 (avec le système S) à une charge traitée de 4.0 (avec le système S_x) et nécessite 5 cœurs au lieu de 3. Comme nous l'avons déjà mentionné, certains concepteurs préfèrent limiter la redondance matérielle. De plus, continuer d'ordonnancer les sous-tâches de substitution après la récupération des sous-tâches perdues n'est pas efficace. La technique TCR présentée dans la section suivante vise à

répondre à ces préoccupations.

II. La Technique Contraindre puis Relâcher

Cette technique vise à utiliser un seul cœur supplémentaire (c-à-d $\mu = 1$) et un nombre de sous-tâches additionnelles correspondant au nombre effectif de sous-tâches perdues. Pour faire face à la panne, les fenêtres d'exécution des sous-tâches sont dynamiquement reconfigurées afin de permettre le réordonnancement des sous-tâches perdues et d'adapter l'exécution à la nouvelle configuration matérielle. L'idée principale de cette technique est de contraindre les échéances des tâches en début d'exécution et de les relâcher progressivement après la détection de la panne. Contraindre les échéances des tâches permet de créer dans chaque tâche un intervalle de temps, appelé *marge de tolérance*, qui sera utilisé comme temps additionnel pour réordonnancer les sous-tâches perdues. Cette marge est donc construite en passant d'un système de tâches à échéances sur requêtes $S : \{\tau_i < C_i, T_i >, i = 1..n\}$ à un système de tâches à échéances contraintes $S' : \{\tau_i < C_i, D'_i, T_i >, i = 1..n\}$. La marge de tolérance d'une tâche τ_i correspond à l'intervalle de temps $[D'_i, T_i]$ (voir Figure 6.4). Lorsque la panne est détectée, si une seule sous-tâche est perdue, la marge de tolérance lui est attribuée comme fenêtre d'exécution pour son réordonnancement. Cependant, s'il y a un nombre x_i de sous-tâches perdues ($x_i > 1$), cette marge devra être divisée en x_i fenêtres d'exécution pour le ré-ordonnancement.

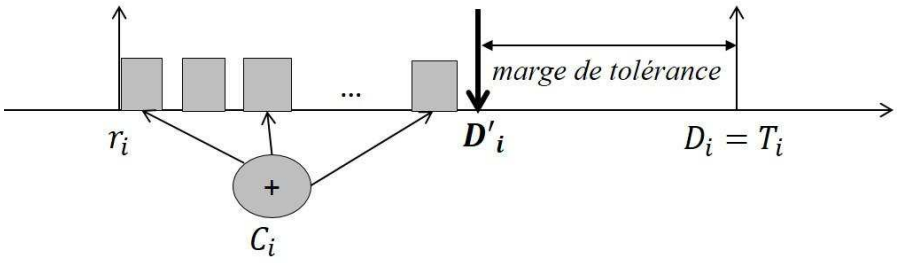

FIGURE 6.4 – Construction de la marge de tolérance d'une tâche

L'utilisation de la technique *TCR* nécessite de déterminer les échéances contraintes D'_i d'une tâche de telle sorte que la marge de tolérance construite soit suffisante pour permettre le réordonnancement de toutes les sous-tâches perdues. Cette détermination est traitée dans le paragraphe II.2. Mais avant, nous présentons dans le paragraphe suivant comment la réconfiguration du système s'opère pour la reprise des sous-tâches perdues.

II.1. Reconfiguration dynamique des fenêtres d'exécution

Pour rendre l'ordonnancement tolérant à une panne, la technique TCR procède à des changements de mode afin d'adapter l'exécution du système à la modification de la configuration

matérielle. Comme illustré par la Figure 6.5, l'exécution du système débute dans le mode "contraint" sur $m+1$ cœurs. Dans ce mode, on se prépare à l'éventualité d'une panne. C'est pourquoi les échéances sont contraintes pour prévoir une marge de tolérance. Lorsqu'une panne est détectée, le système bascule en mode "récupération" dans lequel la marge de tolérance des tâches impactées est utilisée pour construire des fenêtres d'exécution supplémentaires à affecter aux sous-tâches perdues. Les tâches non impactées retrouvent immédiatement leurs paramètres initiaux. Lorsque toutes les sous-tâches perdues ont été réordonnancées, le système bascule dans le mode "normal" où toutes les tâches ont leurs paramètres temporels initiaux. A chaque changement de mode, les fenêtres d'exécution des sous-tâches sont dynamiquement reconfigurées pour les adapter au nouveau mode. C'est ainsi que dans le mode "contraint", les fenêtres d'exécution des sous-tâches sont calculées à partir du système de tâches à échéances contraintes $S' : \{\tau_i < C_i, D'_i, T_i >, i = 1..n\}$ obtenu à partir du système initial $S : \{\tau_i < C_i, T_i >, i = 1..n\}$ par l'une des méthodes de détermination des échéances contraintes présentées au paragraphe II.2. En mode "récupération", les fenêtres d'exécution sont calculées à partir des paramètres d'un système intermédiaire $S(I) : \{\tau_i < C_i + x_i, T_i >, i = 1..n\}$, où $I = (x_1, x_2, .., x_n)$ dans lequel x_i sous-tâches supplémentaires sont ajoutées à chaque tâche. Leurs fenêtres d'exécution serviront à réordonnancer les x_i sous-tâches perdues. Toutefois, cela n'est envisageable que si on dispose de temps additionnel. En effet, étant donné que dans ce mode il reste m cœurs fonctionnels, si $U = m$, il n'est pas possible de disposer de temps d'exécution additionnel car le système est saturé. Le pire cas c'est quand on a $U = m$ alors que la tâche de plus grande période a besoin de x unités de temps supplémentaire. C'est pour cette raison que considérons ici que $m = \left\lceil U + Max(\frac{x}{T_i}) \right\rceil$.
Le mode de récupération dure jusqu'à la prochaine hyperpériode $Next_H$ qui est l'instant où on est sûr que toutes les tâches ont récupéré leurs sous-tâches perdues. A partir de cet instant, le système bascule en mode normal où les fenêtres d'exécution sont calculées avec les paramètres initiaux du système S.

Eu égard à ce qui précède, à l'instant de la détection de la panne, la reconfiguration des fenêtres d'exécution du système s'opère en suivant le protocole (P_{TCR}) ci-dessous, où $r_i^j(\Omega)$ et $d_i^j(\Omega)$) désignent respectivement la pseudo-date de réveil et la pseudo-échéance d'une sous-tâche τ_i^j calculés avec les paramètres temporels du système Ω.

Protocole P_{TCR}

- **Tâche non affectée par la panne** ($\{\tau_i / x_i = 0\}$).
  Retour aux fenêtres initiales : $[r_i^j(S'), d_i^j(S')) \longmapsto [r_i^j(S), d_i^j(S))$
- **Tâche dont l'instance k_0 est affectée par la panne** ($\{\tau_{i_0} / x_{i_0} \neq 0\}$).
  - ✓ Les sous-tâches des instances suivantes retrouvent leurs fenêtres initiales :
    $j \geq k_0 \times C_{i_0}$: $[r_{i_0}^j(S'), d_{i_0}^j(S')) \longmapsto [r_{i_0}^j(S), d_{i_0}^j(S))$.

- ✓ Les sous-tâches non-ordonnancées de l'instance k_0 gardent leurs fenêtres contraintes : $j_0 < j < k_0 \times C_{i_0} : [r^j_{i_0}(S'), d^j_{i_0}(S')) \longmapsto [r^j_{i_0}(S'), d^j_{i_0}(S'))$.
- ✓ chaque sous-tâche perdue $\tau^{j_0}_{i_0}$ de l'instance k_0 prend la fenêtre d'une sous-tâche de substitution $s^{j_s}_{i_0}$:
$[r^{j_0}_{i_0}(S'), d^{j_0}_{i_0}(S')) \longmapsto [r^{j_s}_{i_0}(S(I)), d^{j_s}_{i_0}(S(I)))$.

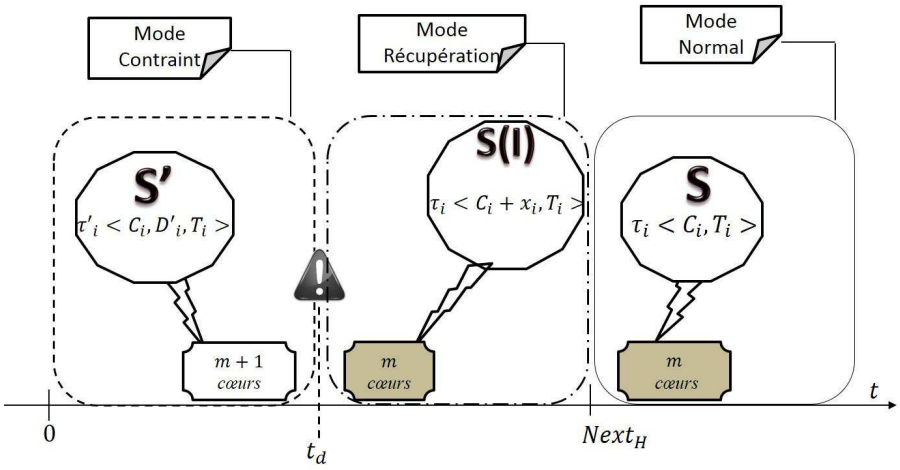

FIGURE 6.5 – Ordonnancement PD2 avec panne et reprise par la technique TCR

Puisque $D'_i < T_i \ \forall \tau_i$, on a $r^j_i(S') \leq r^j_i(S)$ et $d^j_i(S') \leq d^j_i(S)$ et donc, lorsqu'on contraint les échéances des tâches du système, les fenêtres d'exécution de leurs sous-tâches sont avancées et de taille plus réduite. Dans la Figure 6.7 par exemple, les fenêtres d'exécution des sous-tâches de la tâche τ_2 dans le système S' sont avancées et de taille plus réduite que dans le système S. C'est ainsi que dans le système S', la sous-tâche τ^1_2 est réveillée à l'instant 1 et sa fenêtre est de taille 2 alors que dans le système S, elle est réveillée à l'instant 3 et a une fenêtre de taille 3.

La technique *TCR* utilise donc les systèmes suivants pour déterminer les fenêtres d'exécution des sous-tâches au cours de l'ordonnancement :

- Le système à ordonnancer ou *système initial* $S : \{\tau_i < C_i, T_i >, i = 1..n\}$, composé de tâches à échéances sur requêtes ;
- Le *système contraint* $S' : \{\tau'_i < C_i, D'_i, T_i >, i = 1..n\}$ composées des mêmes tâches que S mais avec des échéances contraintes ;
- Le *système intermédiaire* $S(I) : \{\tau_i < C_i + x_i, T_i >, i = 1..n\}$ composé de tâches ayant par instance x_i sous-tâches de plus que dans S. $"I"$ désigne le n-uplet $(x_1, x_2, ..., x_n)$ dont les éléments sont les nombres de sous-tâches perdues de chaque tâche, avec $x_i \geq 0 \ \forall i$ et $\sum x_i \leq x$.

Nos hypothèses sur ces systèmes sont les suivantes :

1. La condition (2.2) est satisfaite et donc le système S est ordonnançable sur m coeurs, ce qui est garanti par le choix de $m = \left\lceil U + Max(\frac{x}{T_i}) \right\rceil$. Cette valeur de m assure par ailleurs que $S(I)$ est ordonnançable sur m cœurs pour tout ensemble I.
2. La condition (6.1) est vérifiée : chaque tâche de S dispose d'au moins x unités de temps entre son WCET et sa période.
3. S' est ordonnançable sur $m+1$ coeurs. Ici on utilise la condition suffisante d'ordonnançabilité définie dans [56] :

$$\sum_{i=1}^{n}(C_i/D'_i) \leq m+1 \tag{6.2}$$

Dans la suite, nous désignons par *séquence TCR*, la séquence d'ordonnancement construite avec la technique *TCR*.

II.2. Calcul des échéances contraintes

Dans ce paragraphe, nous proposons deux méthodes pour obtenir un système de tâches à échéances contraintes à partir d'un système de tâches à échéances sur requêtes : la *méthode de distribution des temps creux* et la *méthode des sous-tâches fantômes*.

II.2.a. Méthode de distribution des temps creux

Cette méthode calcule le nombre d'unités de temps creux par hyperpériode et le répartit en parts égales entre les tâches du système, puis entre leurs instances. La valeur obtenue par instance est ensuite déduite du délai critique initial pour obtenir le nouveau délai critique. Cependant, il faut garantir que la marge de tolérance soit au moins égale à x unités de temps et que le nouveau délai critique ne soit pas inférieur au WCET, raison pour laquelle un ajustement est nécessaire.

Soit donc un système $S : \tau_1 < C_1, T_1 >, \tau_2 < C_2, T_2 >, ..., \tau_n < C_n, T_n >$ composé de n tâches à échéances sur requêtes ordonnançable sur m cœurs. Le calcul des échéances contraintes par la méthode de distribution des temps creux se fait en suivant les étapes suivantes :

1. Calculer le nombre d'unités de temps creux sur par hyperpériode : $IT = (m-U) \times H$;
2. Diviser IT par le nombre de tâches. On obtient $Nb = \left\lfloor \frac{IT}{n} \right\rfloor$ unités par tâche ;
3. Calculer le nombre d'instance par tâche $\forall \tau_i \; n_i = \frac{H}{T_i}$;
4. Déterminer les nouveaux délais critiques : $\forall \tau_i, D'_i = D_i - max(x, \left\lceil \frac{Nb}{n_i} \right\rceil)$;
5. Ajuster les nouveaux délais critiques : $\forall \tau_i$, si $D'_i < C_i$ alors $D'_i \longleftarrow C_i$.

Pour illustrer cette méthode, déterminons le système de tâches à échéances contraintes du système $\psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$ pour $x = 2$. Nous avons
$IT = (3 - \frac{26}{12}) \times 12 = 10$; $Nb = \frac{10}{5} = 2$; $n_1 = \frac{12}{3} = 4$, $n_2 = \frac{12}{6} = 2$, $n_3 = \frac{12}{4} = 3$, $n_4 = n_5 = \frac{12}{12} = 1$. Les nouveaux délais critiques sont les suivants :
$D'_1 = 3 - max(2, \left\lceil \frac{4}{4} \right\rceil) = 1$; $D'_2 = 6 - max(2, \left\lceil \frac{4}{2} \right\rceil) = 4$; $D'_3 = 4 - max(2, \left\lceil \frac{4}{3} \right\rceil) = 2$;
$D'_4 = D'_5 = 12 - max(2, \left\lceil \frac{4}{1} \right\rceil) = 8$.
L'ajustement n'est pas nécessaire car les nouveaux délais sont supérieurs ou égaux aux WCET. Le système à échéances contraintes obtenu est :
$\psi' : \tau_1 < 1, 1, 3 >, \tau_2 < 2, 4, 6 >, \tau_3 < 2, 2, 4 >, \tau_4 < 5, 8, 12 >, \tau_5 < 7, 8, 12 >$.

II.2.b. Méthode des sous-tâches fantômes

Etant donné qu'à un délai de détection de x unités de temps correspondent au plus x sous-tâches perdues par instance, la présente méthode détermine le nouveau délai critique en simulant l'ajout de x sous-tâches à chaque instance des tâches. Ensuite, la pseudo-échéance de la dernière sous-tâche réelle est considérée comme nouveau délai critique. La différence avec la technique *TSS* est que les sous-tâches additionnelles ne sont pas effectives car elles ne seront pas ordonnancées. Elles sont uniquement utilisées pour déterminer le nouveau délai critique, c'est pour cette raison qu'elle sont qualifiées de sous-tâches *fantômes*. Cette méthode présente un double avantage : d'une part, elle garantit une marge de tolérance suffisante pour le réordonnancement des sous-tâches perdues et d'autre part, elle offre une indication pour construire des fenêtres d'exécution supplémentaires dont le nombre est fonction du nombre de sous-tâches à réordonnancer.
Soit τ_i une tâche ayant un WCET égal à C_i. Pour déterminer son nouveau délai critique, on considère un WCET égal à $C_i + x$ dans le calcul des fenêtres d'exécution. Cette tâche est alors supposée avoir, par instance, C_i sous-tâches réelles, numérotées de 0 à $C_i - 1$ et x sous-tâches fantômes. On se place dans la première instance et on retient la pseudo-échéance de la dernière sous-tâche réelle $\tau_i^{C_i-1}$ comme délai critique. D'où la formule :

$$D'_i = d_i^{C_i-1}. \tag{6.3}$$

On construit ainsi un *système fantôme* $S_x : \{\tau_i < C_i + x, T_i >, i = 1..n\}$ dans lequel chaque tâches a x sous-tâches additionnelles qui sont les sous-tâches fantômes utilisées pour déterminer les nouveaux délais critiques.
La Figure 6.6 illustre cette méthode pour un délai de détection $x = 2$ unités de temps. Lorqu'une seule sous-tâche est perdue, une seule fenêtre d'exécution est construite dans la marge de tolérance. Par contre, si deux sous-tâches sont perdues, deux fenêtres d'exécution sont construites.

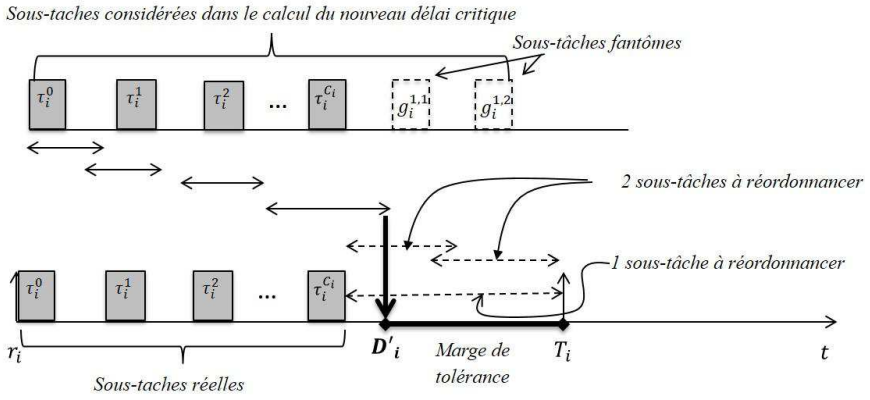

FIGURE 6.6 – Calcul des échéances contraintes par la méthode des sous-tâches fantômes

Appliquons cette méthode au système $\psi : \tau_1 < 1,3 >, \tau_2 < 2,6 >, \tau_3 < 2,4 >, \tau_4 < 5,12 >, \tau_5 < 7,12 >$ pour $x = 2$. Pour cela, nous considérons le système
$\psi_x : \tau_1^+ < 3,3 >, \tau_2^+ < 4,6 >, \tau_3^+ < 4,4 >, \tau_4^+ < 7,12 >, \tau_5^+ < 9,12 >$. L'application de la formule (5.3) fournit les nouveaux délais critiques suivants :
$D'_1 = d_1^{+0} = \lceil \frac{0+1}{\frac{3}{3}} \rceil = 1$; $D'_2 = d_2^{+1} = \lceil \frac{1+1}{\frac{4}{6}} \rceil = 3$; $D'_3 = d_3^{+1} = \lceil \frac{1+1}{\frac{4}{4}} \rceil = 2$; $D'_4 = d_4^{+4} = \lceil \frac{4+1}{\frac{7}{12}} \rceil = 9$;
$D'_5 = d_5^{+6} = \lceil \frac{6+1}{\frac{9}{12}} \rceil = 10$.
Le système à échéances contraintes résultant est :
$\psi' : \tau'_1 < 1,1,3 >, \tau'_2 < 2,3,6 >, \tau'_3 < 2,2,4 >, \tau'_4 < 5,9,12 >, \tau'_5 < 7,10,12 >$.

II.2.c. Comparaison des deux méthodes

La technique TCR a été expérimentée à l'aide du prototype FTA pour les deux méthodes de calcul des délais critiques du système à échéances contraintes S'. Dans un premier temps, les 550 systèmes aléatoires ont été testés avec des paramètres quelconques (instant de la panne, délai de détection, cœur affecté) et un nombre minimum de cœurs $m = \lceil U \rceil$. L'expérience a été répétée 30 fois et le nombre de cas de non validité recensé dans chaque méthode est consigné dans le Tableau 6.1. Il ressort de ces résultats que les délais critiques calculés par la méthode des sous-tâches fantômes (Méthode 2) garantissent la validité de la séquence d'ordonnancement. Par contre, dans de rares cas, les délais critiques calculées avec la méthode de distribution des temps creux (Méthode 1) soulèvent des cas de non validité. Les systèmes non valides par la méthode 1 ont été étudiés et nous avons pu constater que leurs charges respectives étaient telles que $|U - m| < 0.1$. En effet, lorsque la charge U est proche du nombre minimal de cœurs m, il n y a pas suffisamment de temps creux à distribuer entre les instances des tâches. De ce fait, les délais critiques calculés ne permettent pas de construire une marge de tolérance suffisante pour la ré-exécution des sous-tâches perdues. Dans un second temps, nous avons testé ces systèmes non valides avec la méthode 2 sur les mêmes paramètres (instant de panne, délai

de détection, cœur affecté) et la validité a été obtenu pour tous les systèmes. Enfin, nous avons repris toute l'expérimentation avec la Méthode 1 en considérant comme nombre minimal de cœur $m = \lceil U_{max} \rceil$, où U_{max} est la charge maximale qu'on obtiendrait si on ajoute x sous-tâches au système. Le pire cas se produit quand elles sont ajoutées à la tâche de plus petite période. $U_{max} = max_k \left(\sum_{i \neq k} \left(\frac{C_i}{T_i} \right) + \frac{C_k + x}{T_k} \right) = \sum_{i=1}^{n} \left(\frac{C_i}{T_i} \right) + max_k \left(\frac{x}{T_k} \right) = U + \frac{x}{min(T_i)}$.

Avec cette nouvelle valeur de m, toutes les séquences obtenues ont été valides et équitables par la Méthode 1.

TABLE 6.1 – Expérimentation de TCR : 550 systèmes aléatoires simulés 30 fois avec $x = 1$

avec $m = \lceil U \rceil$		avec $m = \lceil U_{max} \rceil$	
Dist. temps creux	**S/tâches fantômes**	**Dist. temps creux**	**S/tâches fantômes**
9 cas de non validité	0 cas de non validité	0 cas de non validité	0 cas de non validité

Il ressort de cette expérimentation que la méthode des sous-tâches fantômes garantit la validité de l'ordonnancement pour $m = \lceil U \rceil$. Pour obtenir la validité avec la méthode de distribution des temps creux, il faut considérer $m = \lceil U_{max} \rceil$. La méthode des sous-tâches fantômes est donc plus flexible et permet d'utiliser moins de cœurs que la méthode de distribution des temps creux. Puisque la valeur $m = \left\lceil U + Max(\frac{x}{T_i}) \right\rceil$ a été retenue précédemment pour éviter la saturation du système (voir page 101), les deux méthodes sont applicables. Néanmoins, la méthode de distribution des temps creux trouve sa limite si dans l'hyperpériode, il y a moins d'unités de temps creux à distribuer que d'instances de tâches. Son utilisation requiert donc d'imposer la contrainte supplémentaire $IT \geq \sum_{i=1}^{n} (\frac{H}{T_i})$ au système. Compte tenu de cet aspect, bien que cette méthode soit plus simple à implémenter, nous retenons la méthode des sous-tâches fantômes pour calcul des délais critiques du système contraint S' de la technique TCR.

II.3. Justification de la technique

Nos choix dans cette technique visent à surcharger le système le moins possible afin de pouvoir ordonnancer le plus grand nombre de systèmes possible avec un seul cœur additionnel. C'est pourquoi les fenêtres d'exécution sont calculées à partir des paramètres temporels de trois systèmes différents. Premièrement nous utilisons le système S' avec des échéances contraintes ($D'_i < T_i$) et les WCET initiaux C_i. En effet, ordonnancer les sous-tâches additionnelles (c-à-d les sous-tâches fantômes) comme dans TSS n'est pas optimal, car elles ne représentent rien d'autre que des unités de temps creux. Nous gagnons plutôt à utiliser ces unités de temps pour ordonnancer d'autres sous-tâches réelles en attente. De plus, exécuter S' plutôt que S_x a l'avantage d'induire une charge de traitement plus réduite, ce qui est avantageux pour obtenir

l'ordonnançabilité du système sur $m+1$ cœurs en mode "contraint". Ensuite, nous utilisons le système intermédiaire $S(I)$ où toutes les tâches non impactées retrouvent leurs paramètres du système initial S dont les fenêtres d'exécution sont plus large que dans S'. Cela évite des fautes temporelles et offre plus de flexibilité pour le ré-ordonnancement des sous-tâches perdues. Enfin, après la phase de récupération, le système le moins contraint des trois, à savoir le système S est utilisé pour poursuivre l'ordonnancement sur les m cœurs restants avec des fenêtres d'exécution plus larges.

II.4. Algorithme d'ordonnancement

L'Algorithme 4 (voir page suivante) effectue l'ordonnancement PD2 d'un système par la technique *TCR*. On suppose l'existence des primitives des suivantes :

- $sousTaches(S)$: retourne la liste des sous-tâches du système S ;
- $pretes(ST, t)$: retourne la liste des sous-tâches de ST prêtes à l'instant t ;
- $card(L)$: retourne le nombre d'éléments de la liste L ;
- $ranger(L)$: range les sous-tâches de la liste L par ordre de priorité PD2 décroissante ;
- $init(Seq)$: initialise la séquence Seq par des unités de temps creux ;
- $detection()$: retourne le cœur défaillant ou -1 s'il n'y en a aucun ;
- $contraindre(S, x)$: retourne le système de tâches à échéances contrainte issu de S par la méthode des sous-tâches fantômes ;
- $reconfigurer(ST, S, L)$: reconfigure les fenêtres d'exécution des sous-tâches de ST à l'aide des paramètres du système S et de la liste des sous-tâches perdues L suivant le protocole P_{TCR}.

II.5. Exemple

Considérons à nouveau le système
$\psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$ avec $x = 2$ et $t_d = 3$.
Nous avons $m = \left\lceil 2.16 + \frac{2}{3} \right\rceil = 3$. Le système contraint obtenu par la méthode des sous-tâches fantômes est :
$\psi' : \tau'_1 < 1, 1, 3 >, \tau'_2 < 2, 3, 6 >, \tau'_3 < 2, 2, 4 >, \tau'_4 < 5, 9, 12 >, \tau'_5 < 7, 10, 12 >$ et son facteur de charge $CH_\psi = \sum_{i=1}^{n} (C_i/D'_i) = 3.92 < 4 = m + 1$.
Le Tableau 6.2 présente les paramètres des fenêtres d'exécution des sous-tâches de ce système contraint calculés à l'aide de la formule (2.4). Nous nous limitons à la première hyperpériode.

Supposons qu'une panne est détectée sur le coeur C2. Les sous-tâches τ_3^1 et τ_5^2 affectées à ces cœurs aux instants $t_1 = 1$ et $t_2 = 2$ sont perdues et on a $I = (0, 0, 1, 0, 1)$. Le système intermédiaire est $\psi(I) : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_{3_0} < 3, 4 >, \tau_4 < 5, 12 >, \tau_5 < 8, 12 >$ avec

Algorithm 4: Ordonnancement PD2 avec panne et reprise par la technique TCR

Inputs : $S : \tau_i < C_i, T_i >$: système à ordonnancer ;
$C = (C_1, C_2, .., C_{m+1})$: liste des cœurs du processeur ;
x : délai de détection de la panne.

Output: $Seq = Seq^{(S)}_{[(m+1)\longmapsto t_d(m)]}$: Séquence d'ordonnancement de S sur une hyperpériode.

$S' \leftarrow contraindre(S, x)$;
$ST' \leftarrow sousTaches(S')$;
t $\leftarrow 0$;
$t_d \leftarrow 0$;
$init(Seq)$;

// Affectation des sous-tâches avant la panne

while $detection() = -1$ **do** *// i.e tant qu'aucun cœur n'est défaillant*
    P $\leftarrow pretes(ST', t)$;
    $ranger(P)$;
    k $\leftarrow 1$;
    j $\leftarrow 1$;
    while $(k \leq m+1) \wedge (j \leq card(P))$ **do**
        $Seq(C(k), t) \leftarrow P(j)$;
        j $\leftarrow$ j+1 ;
        k $\leftarrow$ k+1 ;
    t $\leftarrow$ t+1 ; $t_d \leftarrow$ t ;

$num \leftarrow detection()$;

// Panne détectée: réconfiguration et poursuite de l'ordonnancement

$Perdues \leftarrow \emptyset$;
for $i = t_d - x$ *to* $t_d - 1$ **do** *// constitution de la liste des sous-tâches perdues*
    $Perdues \leftarrow Perdues \bigcup Seq(C(num), i)$;

$STI \leftarrow reconfigurer(ST', S, Perdues)$;
$Next_H \leftarrow \lfloor \frac{t_d}{H} \rfloor \times H + H$;

while $t < Next_H$ **do**
    P $\leftarrow pretes(STI, t)$;
    $ranger(P)$;
    k $\leftarrow 1$;
    j $\leftarrow 1$;
    while $(k \leq m+1) \wedge (j \leq card(P))$ **do**
        if $k \neq num$ **then**
            $Seq(C(k), t) \leftarrow P(j)$;
            j $\leftarrow$ j+1 ;
        k $\leftarrow$ k+1 ;
    t $\leftarrow$ t+1 ;

TABLE 6.2 – Paramètres des sous-tâches des premières instances des tâches du système ψ'

τ_i	τ_i^j	$r_i'^j$	$d_i'^j$	$b_i'^j$	$D_i'^j$
τ_1	τ_1^0	0	1	0	0
	τ_1^1	3	4	0	4
	τ_1^2	6	7	0	7
	τ_1^3	9	10	0	10
τ_2	τ_2^0	0	2	1	3
	τ_2^1	1	3	0	3
	τ_2^2	6	8	1	9
	τ_2^3	7	9	0	9
τ_3	τ_3^0	0	1	0	1
	τ_3^1	1	2	0	2
	τ_3^2	4	5	0	5
	τ_3^3	5	6	0	6
	τ_3^4	8	9	0	9
	τ_3^5	9	10	0	10
τ_4	τ_4^0	0	2	1	3
	τ_4^1	1	4	1	5
	τ_4^2	3	6	1	7
	τ_4^3	5	8	1	9
	τ_4^4	7	9	9	
τ_5	τ_5^0	0	2	1	4
	τ_5^1	1	3	1	4
	τ_5^2	2	5	1	6
	τ_5^3	4	6	1	6
	τ_5^4	5	8	1	10
	τ_5^5	7	9	1	10
	τ_5^6	8	10	0	10

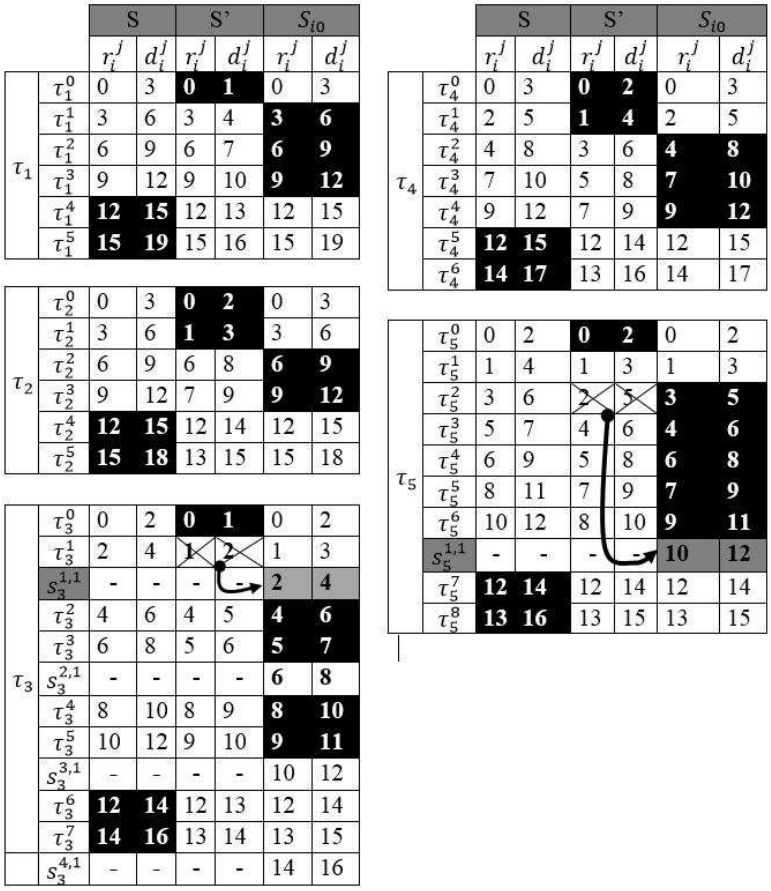

FIGURE 6.7 – Fenêtres d'exécution utilisées par les sous-tâches de ψ (cases en noir)

$U_{\psi(I)} = 2.50 < m$. Une sous-tâche supplémentaire a été ajoutée à chaque instance des tâches impactées $\tau 3$ et $\tau 5$. Puisque les sous-tâches perdues appartiennent aux premières instances de ces tâches, les fenêtres d'exécution des sous-tâches $s_3^{1,1}$ et $s_5^{1,1}$ sont utilisées pour les réordonnancer.

La Figure 6.7 montre les fenêtres d'exécution utilisées par chaque sous-tâche au cours de l'ordonnancement. La séquence TCR du système ψ est présentée par la Figure 6.8.

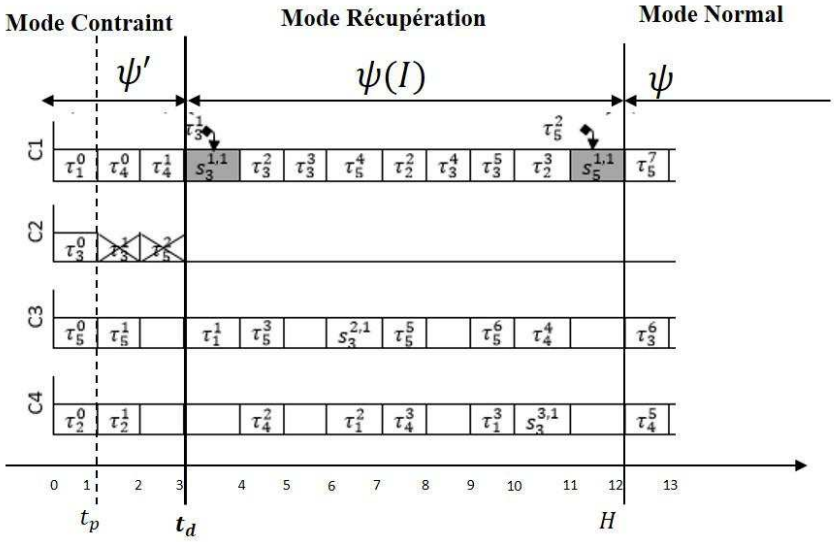

FIGURE 6.8 – Exemple d'ordonnancement PD2 par la technique TCR

II.6. Problème d'inversion de l'ordre de priorité

L'algorithme PD2 attribue les priorités entre sous-tâches en fonction de leurs fenêtres d'exécution. Etant donné que ces fenêtres ne sont pas nécessairement les mêmes dans les systèmes S, S' et $S(I)$, la technique TCR peut engendrer des inversions de priorités lorsque le système passe d'un mode d'exécution à un autre. En effet, si on considère le passage du mode "contraint" au mode "récupération", on peut constater qu'une sous-tâche τ_u^v peut être plus prioritaire qu'une sous-tâche τ_k^q dans le système S' mais être moins prioritaire qu'elle dans le système $S(I)$. De plus, une sous-tâche peut être réveillée dans le système S' alors qu'elle ne l'est pas encore dans $S(I)$. Par exemple, en observant les Figures 6.8 et 6.10, on peut constater que la sous-tache τ_1^0 est plus prioritaire que τ_3^0 dans le système ψ' alors que dans le système $\psi(I)$, τ_3^0 est plus prioritaire que τ_1^0.

L'inversion de l'ordre de priorité entre les sous-tâches a une incidence sur l'analyse d'ordonnancement de la séquence TCR. En effet, à l'instant t_d (où se produit le changement de mode), il reste m cœurs fonctionnels et les sous-tâches ont pour fenêtres d'exécution celles calculées dans le système $S(I)$. On s'attend alors à ce qu'à partir de cet instant, la séquence TCR construite dans le mode "récupération" soit identique à la séquence système $S(I)$ sur m coeurs. Ce n'est pas toujours le cas à cause de deux facteurs. En effet, comme l'illustre la Figure 6.9 , si on place à l'instant t_d où doit se produire le changement de mode et qu'on observe la séquence d'ordonnancement de S' sur $(m+1)$ cœurs et celle de $S(I)$ sur m cœurs, on constate :

— qu'il y a des sous-tâches déjà ordonnancées dans la séquence de S' (à des instants $t < t_d$) mais pas encore dans $S(I)$ (elles sont prévues à des instants $t \geq t_d$). Nous les appelons

sous-tâches anticipées. Ces sous-tâches ont été aussi déjà ordonnancées dans la séquence TCR et donc ne font plus partie des tâches prêtes dans le mode "récupération" ;

— inversément, il peut y avoir des sous-tâches déjà ordonnancées dans $S(I)$ (à des instants $t < t_d$) mais dont l'ordonnancement est prévu après (à des instants $t \geq t_d$) dans S'. Nous les appelons *sous-tâches retardées*. Ces sous-tâches n'ont pas encore été ordonnancées dans la séquence TCR et donc font partie des sous-tâches prêtes du mode "récupération".

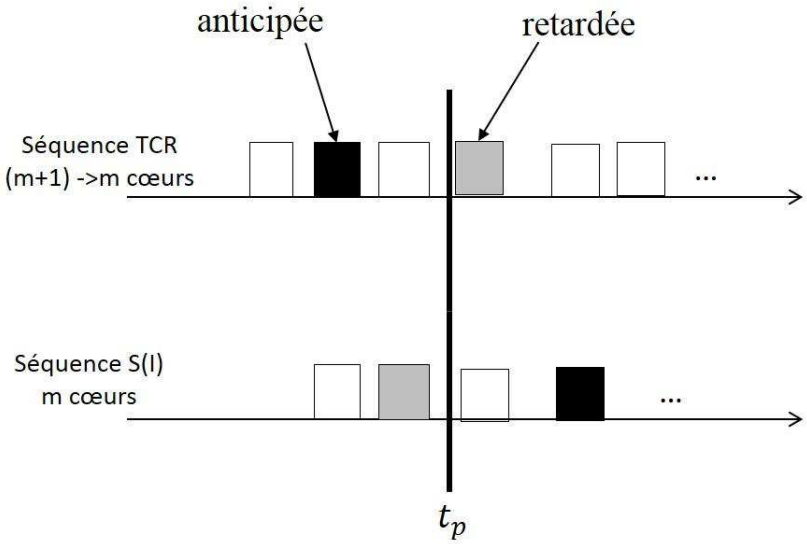

FIGURE 6.9 – Mise en évidence des sous-tâches anticipées et retardées

La présence des sous-tâches anticipées et des sous-tâches retardées explique donc que la séquence TCR dans mode "récupération" ne soit pas identique à la séquence $S(I)$ dans le même intervalle de temps. Dans la suite, nous désignons par $S' \longmapsto S(I)$ le système constitué des sous-tâches de S ordonnancées de l'instant 0 à l'instant $Next_H$ dans les modes "contraint" et "récupération". La présence des sous-taches anticipées permet d'avancer l'exécution de certaines sous-tâches dans le mode "récupération". En effet, puisque ces sous-tâches ont déjà été ordonnancées en mode "contraint", d'autres sous-tâches prêtes seront ordonnancées à leurs places et donc plus tôt qu'elles ne l'auraient été dans la séquence $S(I)$. Les sous-tâches anticipées n'entrainent pas de fautes temporelles. A contrario, dans le mode "récupération", la présence de sous-tâches retardées recule l'ordonnancement de certaines sous-tâches. En effet, elles sont prêtes et prioritaires dans l'ordonnancement TCR, les fenêtres d'exécution sont les mêmes que dans $S(I)$ où elles ont déjà été ordonnancées. Donc, ces sous-tâches retardées sont ordonnancées dans TCR à la place d'autres sous-tâches prêtes (ordonnancées à cet instant dans $S(I)$) qui doivent attendre les instants suivants. Par conséquent, l'ordonnancement de ces sous-tâches reportées aura lieu dans la séquence TCR plus tard qu'elles ne le sont dans la séquence $S(I)$,

ce qui pourrait entrainer des fautes temporelles.
Pour étudier la validité de la technique TCR, nous discutons donc en particulier sur la présence ou non de sous-tâches retardées à l'instant de changement de mode. Il y a absence de sous-tâche retardée si, à l'instant t_d, toutes les sous-tâches déjà ordonnancées dans $Seq_m^{S(I)}$ ont été aussi déjà ordonnancées dans $Seq_{[(m+1)\longrightarrow_{t_p} m]}^{(S'\longmapsto S(I))}$ ce qui se traduit par :

$$\forall \tau_i^j, Exec(\tau_i^j, Seq_m^{S(I)}) < t_d \Longrightarrow Exec(\tau_i^j, Seq_{(m+1)}^{S'}) < t_d$$

Afin d'illustrer ce qui précède, étudions l'incidence de l'inversion de l'ordre de priorité sur l'ordonnancement TCR à la lumière de deux exemples. Comme premier exemple, comparons entre les instants $t_d = 3$ et $Next_H = 12$, la séquence TCR du système Ψ de la Figure 6.8 à la séquence $Seq_3^{\Psi(I)}$ de l'ordonnancement du système $\Psi(I)$ sur 3 cœurs de la Figure 6.10

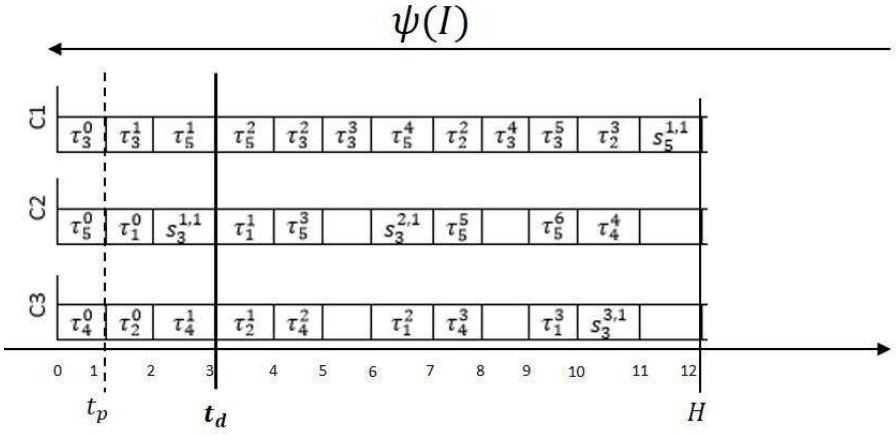

FIGURE 6.10 – Ordonnancement du système intermédiaire $\Psi(I)$ sur 3 cœurs

On constate qu'à l'instant $t_d = 3$ où se produit le changement de mode, la sous-tâche τ_2^1 est ordonnancé dans $Seq_4^{\Psi'}$ à $t = 2$ (et donc avant t_d) alors qu'elle ne l'est pas encore dans $Seq_3^{\Psi(I)}$. C'est donc une sous-tâche anticipée. Il n'y a pas de sous-tâche retardée, car toutes les sous-tâches ordonnancées dans $Seq_3^{\Psi(I)}$ avant t_d sont aussi déjà ordonnancées dans $Seq_4^{\Psi'}$ et donc dans le mode "contraint" de TCR. Dans le mode "récupération", les sous-tâches sont ordonnancées au même instant que dans $Seq_{[3]}^{(\Psi(I))}$ et les sous-tâches perdues τ_3^1 et τ_5^2 sont substituées par $s_3^{1,1}$ et $s_5^{1,1}$.

Considérons un deuxième exemple avec des sous-tâches retardées.
Soit le système Υ suivant composé de 48 tâches. $\tau_{i-j} < C, T >$ désigne une liste de tâches $\tau_i, \tau_{i+1} ... \tau_j$ ayant pour paramètres temporels $< C, T >$.
$\Upsilon : \tau_{0-3} < 1, 20 >, \tau_{4-7} < 1, 36 >, \tau_{8-47} < 2, 38 >$. $U = 2.41$. On suppose un délai de détection de $x = 1$ unité de temps.

Nous avons $m = \lceil 2.41 + \frac{1}{20} \rceil = 3$ et comme système contraint $\Upsilon' = \tau_{0-3} < 1, 10 >, \tau_{4-7} < 1, 18 >, \tau_{8-47} < 2, 26 >$ qui sera exécuté sur $m+1 = 4$ cœurs. Notons qu'avec la technique TSS, on aurait $m_x = 6$ et donc $\mu = 3$.

Supposons une panne détectée à l'instant $t_d = 1$ sur le coeur C4 (avec $x = 1$). La sous-tâche τ_3^0 est affectée et donc $I = (0, 0, 0, 1, 0, ..., 0)$. Les fenêtres d'exécution des premières instances de ces tâches dans les différents systèmes de la technique TCR sont :
$\Upsilon : \tau_{0-3}^0 : [0, 20), \tau_{4-7}^0 : [0, 36), \tau_{8-47}^0 : [0, 19), \tau_{8-47}^1 : [19, 38)$.
$\Upsilon' : \tau_{0-3}^0 : [0, 10), \tau_{4-7}^0 : [0, 18), \tau_{8-47}^0 : [0, 13), \tau_{8-47}^1 : [13, 26)$.
$\Upsilon(I) : \tau_{0-2}^0 : [0, 20), \tau_3^0 : [0, 10), \tau_3^1 : [10, 20), \tau_{4-7}^0 : [0, 36), \tau_{8-47}^0 : [0, 19), \tau_{8-47}^1 : [19, 38)$.

Il y a une inversion de l'ordre de priorité entre les sous-tâches τ_{0-3}^0 et les sous-tâches τ_{8-47}^0. En effet, dans les systèmes Υ et $\Upsilon(I)$, les sous-tâches τ_{8-47}^0 sont plus prioritaires que les sous-tâches τ_{0-3}^0 (car $d_{0-3}^0 = 10 < 13 = d_{8-47}^0$), alors que dans le système Υ', c'est l'inverse (car $d_{8-47}^0 = 19 < 20 = d_{0-3}^0$).

La Figure 6.11 présente dans sa partie supérieure la séquence d'ordonnancement obtenue avec TCR et dans sa partie inférieure la séquence d'ordonnancement de $\Upsilon(I)$ sur 3 cœurs. Les sous-tâches anticipées et les sous-tâches retardées sont mises en évidence.

En mode "récupération", l'ordonnancement du système Υ par la technique TCR se déroule de la façon suivante :

— A l'instant $t = 0$.
Toutes les tâches sont réveillées dans les deux ordonnancements TCR et $\Upsilon(I)$ (car $r_i = 0$) et donc leurs premières sous-tâches sont prêtes. Nous avons deux sous-tâches retardées τ_8^0 et τ_9^0 (ordonnancées à $t = 0$ dans $Seq_3^{\Upsilon(I)}$ mais pas dans $Seq_4^{\Upsilon'}$). Inversement, nous avons trois sous-tâches anticipées τ_0^0, τ_1^0 et τ_2^0 (ordonnancées dans $Seq_4^{\Upsilon'}$ mais pas dans $Seq_3^{\Upsilon(I)}$). La sous-tâche perdue est τ_3^0.

— A l'instant $t_d = 1$.
Dans l'ordre de priorité, les sous-tâches prêtes dans $\Upsilon(I)$ sont τ_{10}^0 τ_{11}^0 τ_{12}^0... et dans TCR τ_8^0 τ_9^0 τ_{10}^0 τ_{11}^0 τ_{12}^0.... Dans $Seq_3^{\Upsilon(I)}$, τ_{10}^0, τ_{11}^0 et τ_{12}^0 sont élues et donc ordonnancées. Dans la séquence TCR, τ_8^0 τ_9^0 et τ_{10}^0 sont ordonnancées alors que τ_{11}^0 et τ_{12}^0 qui pourtant sont prêtes ne sont pas élues à cet instant comme dans $Seq_3^{\Upsilon(I)}$ à cause des sous-tâches retardées qui sont plus prioritaires. τ_{11}^0 et τ_{12}^0 s'exécuteront donc dans TCR plus tard que dans $Seq_3^{\Upsilon(I)}$.

— A l'instant $t = 2$.
Les sous-tâches prêtes dans $\Upsilon(I)$ sont τ_{13}^0 τ_{14}^0 τ_{15}^0... et dans TCR τ_{11}^0 τ_{12}^0 τ_{13}^0 τ_{14}^0 τ_{15}^0.... Dans $Seq_3^{\Upsilon(I)}$, τ_{13}^0, τ_{14}^0 et τ_{15}^0 sont élues et donc ordonnancées. Dans la séquence TCR, τ_{11}^0, τ_{12}^0, τ_{13}^0 qui sont ordonnancées alors que τ_{14}^0 et τ_{15}^0 pourtant prêtes ne sont pas élues

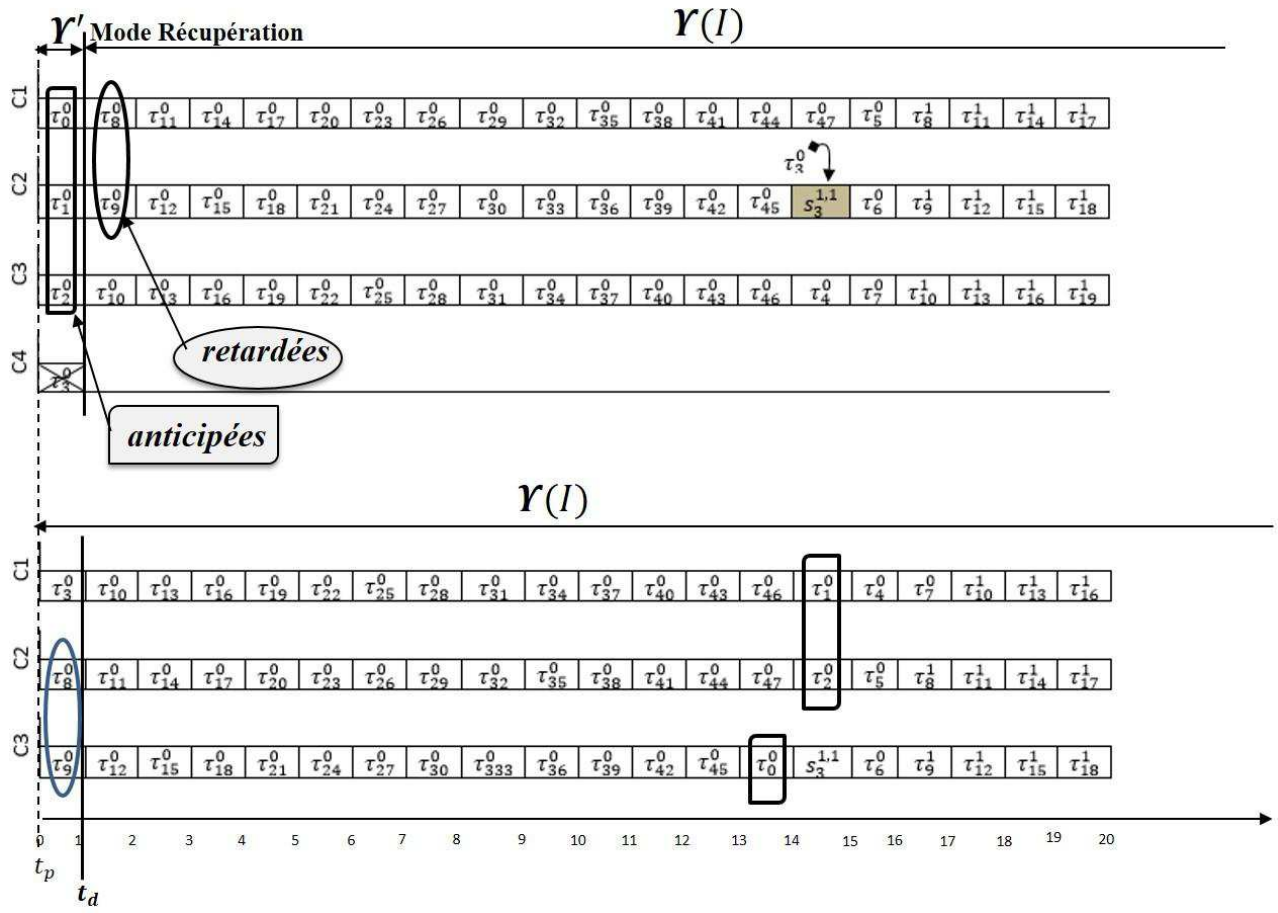

FIGURE 6.11 – Mise en évidence de l'inversion de priorités dans la technique TCR

à cet instant à cause des sous-tâches τ_{11}^0 et τ_{12}^0 reportées à l'instant 1. Donc τ_{14}^0 et τ_{15}^0 s'exécuteront dans TCR plus tard que dans $Seq_3^{\Upsilon(I)}$.

— De l'instant $t = 3$ à l'instant $t = 13$.
Le report d'exécution des sous-tâches continue : deux sous-tâches ordonnancées à l'instant t dans $Seq_3^{\Upsilon(I)}$ sont ordonnancées dans la séquence TCR à $t + 1$ et engendrent à leur tour des reports.

— A l'instant $t = 14$.
Les sous-tâches prêtes dans $\Upsilon(I)$ sont τ_{46}^0 τ_{47}^0 τ_0^0... et dans TCR τ_{44}^0 τ_{45}^0 τ_{46}^0 τ_{47}^0... mais pas τ_0^0 qui est une tâche anticipée et donc déjà ordonnancée en mode "contraint". Dans $Seq_3^{\Upsilon(I)}$, τ_{46}^0, τ_{47}^0 et τ_0^0 sont élues et donc ordonnancées. Dans la séquence TCR, c'est τ_{44}^0 τ_{45}^0 τ_{46}^0. Seule τ_{47}^0 s'exécutera plus tard que dans $Seq_3^{\Upsilon(I)}$.

— A l'instant $t = 15$.
Les sous-tâches prêtes dans $\Upsilon(I)$ sont τ_1^0 τ_2^0 $s_3^{1,1}$ τ_4^0... et dans TCR τ_{47}^0 $s_3^{1,1}$ τ_4^0... mais pas

τ_1^0 et τ_2^0 qui sont des sous-tâches anticipées. Dans $Seq_3^{\Upsilon(I)}$, τ_1^0, τ_2^0 et $s_3^{1,1}$ sont élues et donc ordonnancées. Dans la séquence TCR, ce sont τ_{47}^0 $s_3^{1,1}$ et τ_4^0. Il n'y a plus de report de sous-tâche. Au contraire, la sous-tâche τ_4^0 s'exécute plus tôt que dans $Seq_3^{\Upsilon(I)}$. A partir de cet instant, les sous-tâches sont ordonnancées dans la séquence TCR soit au même instant que dans $Seq_3^{\Upsilon(I)}$, soit plus tôt.

Les exemples ci-dessus nous conduisent aux définitions et remarques suivantes.

Définition 3. *Sous-tâche retardée*
Sous-tâche ordonnancée dans $Seq_{[m]}^{S(I)}$ avant l'instant t_d mais ordonnancée à ou après l'instant t_d dans $Seq_{[m+1]}^{S'}$.

Définition 4. *Sous-tâche anticipée*
Sous-tâche ordonnancée dans $Seq_{[m+1]}^{S'}$ avant l'instant t_d mais ordonnancée à ou après l'instant t_d dans $Seq_{[m]}^{S(I)}$.

Définition 5. *Sous-tâche reportée*
Sous-tâche ordonnancée à un instant $t_1 \geq t_d$ dans $Seq_{[m]}^{S(I)}$ mais ordonnancée plus tard dans la séquence TCR (c-à-d un instant $t_2 > t_1$).

Par ailleurs, on constate qu'en mode "récupération" :

1. A partir de l'instant où toutes les tâches ordonnancées dans la séquence $S(I)$ l'ont aussi été dans la séquence TCR, les sous-tâches s'exécutent soit au même instant que dans $S(I)$, soit plus-tôt. S'il n'y a pas de sous-tâches retardées, cet instant correspond à l'instant t_d. Dans le cas contraire, cet instant correspond à l'instant où le nombre de sous-tâches anticipées ordonnancées dans $S(I)$ est supérieur ou égal au nombre de sous-tâche retardées.
2. Si la panne intervient au moment où toutes les tâches sont réveillées (c-à-d à l'instant 0 ou à une hyperpériode), s'il y a n_r sous-tâches retardées, alors il y a au moins $n_r + 1$ sous-tâches anticipées.
3. Les n_r sous-tâches retardées s'exécutent à l'instant t_d. Leur exécution peut engendrer des reports d'exécution en cascade qui cessent à l'instant où n_r des sous-tâches anticipées ont été ordonnancées dans $S(I)$.
4. Une sous-tâche reportée à un instant t est exécutée à l'instant $t + 1$.

Ces remarques seront généralisées dans la section suivante.

II.7. Eléments de Preuve

Nous rappelons que $Seq_{[k]}^{(\Omega)}$ désigne la séquence obtenue par l'ordonnancement d'un système Ω sur k cœurs. Dans ce paragraphe, nous démontrons que les sous-tâches sont ordonnancées avec TCR dans leurs fenêtres d'exécution initiales (c-à-d celles du système S). Pour cela, dans chaque mode, nous comparons la séquence TCR avec la séquence qui serait construite si on ordonnançait, sur le même nombre de cœurs, le système dont les paramètres temporels ont servi au calcul des fenêtres d'exécution des sous-tâches. C'est ainsi que la séquence TCR est comparée en mode "contraint" à $Seq_{[m+1]}^{(S')}$, en mode "recupération" à $Seq_{[m]}^{(S(I))}$ et en mode "normal" à $Seq_{[m]}^{(S)}$. Ces séquences sont en effet supposées valides et équitables (voir équations (1.2), (5.3) et (5.4)). Par ailleurs, du fait de l'inversion des priorités, dans le mode "recupération", nous étudierons le cas où il y a des sous-tâches retardées et le cas où il n'y en a pas.

Dans un premier temps, nous mettons en place quelques propriétés qui sont valides dans tous les cas. Ensuite, nous étudierons les deux cas annoncés (c-à-d avec ou sans sous-tâches retardées).

Nous notons par la suite $H(t), t \in \mathbb{N}$ la propriété suivante :
$H(t)$: A l'instant t ($t \geq t_d$), toutes les sous-tâches déjà ordonnancées dans $Seq_{[m]}^{S(I)}$ ont déjà été ordonnancées dans $Seq_{[(m+1)\longrightarrow_{t_p} m]}^{(S'\longmapsto S(I))}$:
$\forall \tau_i^j, Exec(\tau_i^j, Seq_m^{S(I)}) < t \ (t \geq t_d) \Longrightarrow Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_p} m]}^{(S'\longmapsto S(I))}) < t.$

Soulignons que dire qu'il n'y a pas de sous-tâche retardée revient à dire que $H(t_d)$ est vérifiée.

Dans la suite, pour simplifier les écritures, puisque les systèmes $S' \longmapsto S(I)$ et $S(I)$ s'exécutent sur m cœurs à partir de l'instant t_d, le nombre de cœur n'est plus précisé dans les notations et donc est implicite. C'est ainsi que nous notons par $Pretes(\Omega, t)$ et $NonElues(\Omega, t)$ respectivement, la liste des sous-tâches prêtes et la liste des sous-tâches non élues d'un système Ω à l'instant t.

La propriété ci-dessous établit la monotonie de la propriété $H(t)$:

$$\forall t \geq t_d : H(t) \Rightarrow H(t'), \text{ avec } t' \geq t.$$

Supposons $H(t)$ vérifiée. Nous montrons que $H(t+1)$ l'est, ce qui établira la monotonie de H. Soit donc τ_i^j tel que $Exec(\tau_i^j, Seq_m^{S(I)}) < t+1$. τ_i^j est ordonnancée dans $Seq_m^{S(I)}$ soit à l'instant t, soit avant.

— Si $Exec(\tau_i^j, Seq_m^{S(I)}) < t$ alors par $H(t)$ on a $Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_d} m]}^{(S'\longmapsto S(I))}) < t$ et donc $H(t+1)$ est vérifiée.

— Si $Exec(\tau_i^j, Seq_m^{S(I)}) = t$, alors dans $Seq_m^{S(I)}$ trois cas sont envisageables : soit τ_i^j est réveillée à l'instant t, soit τ_i^j était déjà réveillée mais n'était pas prête à l'instant $t-1$, soit enfin τ_i^j était prête à l'instant $t-1$ mais n'a pas été élue. Nous étudions ces trois cas :

✓ Cas 1 : τ_i^j est réveillée dans $S(I)$ à l'instant t.
τ_i^j est aussi réveillée dans $S' \longmapsto S(I)$ car les dates de réveil sont les mêmes à partir de l'instant t_d. Si τ_i^j est ordonnancée à l'instant t dans $Seq_m^{S(I)}$, c'est que τ_i^{j-1} a été ordonnancée avant l'instant t, donc on a $Exec(\tau_i^{j-1}, Seq_m^{S(I)}) < t$. D'après $H(t)$, il en découle que $Exec(\tau_i^{j-1}, Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}) < t$ et donc τ_i^j est prête à l'instant t dans $S' \longmapsto S(I)$ (i.e $\tau_i^j \in Pretes(S' \longmapsto S(I), t)$).

✓ Cas 2 : τ_i^j est réveillée à l'instant $t-1$ dans $S(I)$, mais n'est pas prête.
Dans ce cas, τ_i^{j-1} est ordonnancée dans $Seq_m^{S(I)}$ à l'instant $t-1$
($Exec(\tau_i^{j-1}, Seq_m^{S(I)}) = t-1$) et d'après $H(t-1)$ on a
$Exec(\tau_i^{j-1}, Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}) \leq t-1$. Donc, soit τ_i^j a déjà été ordonnancée dans $Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}$ à l'instant t (car si τ_i^{j-1} a été ordonnancée avant l'instant $t-1$ alors, τ_i^j était prête à $t-1$), soit τ_i^j est prête à l'instant t dans $S' \longmapsto S(I)$
(i.e $\tau_i^j \in Pretes(S' \longmapsto S(I), t)$).

✓ Cas 3 : τ_i^j est une sous-tâche prête non élue à l'instant $t-1$ dans $S(I)$.
Dans ce cas, à l'instant $t-1$, τ_i^j est prête dans $S(I)$ et τ_i^{j-1} a déjà été ordonnancée dans $Seq_m^{S(I)}$. On a donc $Exec(\tau_i^{j-1}, Seq_m^{S(I)}) < t-1$ et d'après $H(t-1)$, $Exec(\tau_i^{j-1}, Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}) < t-1$. Donc, à l'instant t dans $S' \longmapsto S(I)$, τ_i^j est soit prête (i.e $\tau_i^j \in Pretes(S' \longmapsto S(I), t)$), soit déjà ordonnancée.

Il ressort des trois cas ci-dessus qu'à l'instant t dans $S' \longmapsto S(I)$, τ_i^j est soit prête (i.e $\tau_i^j \in Pretes(S' \longmapsto S(I), t)$), soit déjà ordonnancée. Si à l'instant t, τ_i^j a déjà été ordonnancée dans $Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}$, alors $H(t+1)$ est vrai.

Étudions le cas $\tau_i^j \in Pretes(S' \longmapsto S(I), t)$.
Si dans $S' \longmapsto S(I)$ il existe m sous-tâches prêtes $\tau_{u_1}^{v_1}, ..., \tau_{u_m}^{v_m}$ plus prioritaires que τ_i^j, ces sous-tâches n'ont pas été ordonnancées plus tôt dans $Seq_m^{S(I)}$, car sinon, d'après $H(t)$ elle auraient déjà été ordonnancées plus tôt dans $Seq_{[(m+1)\longrightarrow t_d m]}^{(S' \longmapsto S(I))}$ également. Si les sous-tâches $\tau_{u_1}^{v_1}, ..., \tau_{u_m}^{v_m}$ sont prêtes à l'instant t dans $S(I)$, elles sont également plus prioritaires que τ_i^j et donc τ_i^j ne peut être ordonnancée à l'instant t dans $Seq_m^{S(I)}$. Si les sous-tâches $\tau_{u_1}^{v_1}, ..., \tau_{u_m}^{v_m}$ ne sont pas prêtes à l'instant t dans $S(I)$, c'est que leurs précédentes n'ont pas déjà été ordonnancées. Dans ce cas, pour chaque sous-tâche non prête parmi $\tau_{u_1}^{v_1}, ..., \tau_{u_m}^{v_m}$, la précédente est prête (car $Seq_m^{S(I)}$ est valide et équitable et il y a un chevauchement maximal d'une unité de temps entre des fenêtres d'exécution successives) et est plus

prioritaire que τ_i^j. Donc, il y a à nouveau m sous-tâches prêtes plus prioritaires que τ_i^j dans $S(I)$ et donc τ_i^j ne peut être ordonnancée à l'instant t dans $Seq_m^{S(I)}$.
Donc, comme $Exec(\tau_i^j, Seq_m^{S(I)}) = t$, on en déduit qu'à l'instant t, il n'existe pas m sous-tâches prêtes plus prioritaires que τ_i^j dans $S' \longmapsto S(I)$ et donc τ_i^j est ordonnancée à l'instant t.

Au final, nous avons établi que $Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_d} m]}^{(S'\longmapsto S(I))}) \leq t$ et donc $H(t+1)$ est vérifiée □

Nous pouvons reformuler la monotonie de $H(t)$ par la propriété $P(t)$ suivante :

Propriété 1. *$P(t)$:*
Si $H(t)$ est vraie alors, à partir de l'instant t, toute sous-tâche τ_i^j n'est pas ordonnancée dans $Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S'\longmapsto S(I))}$ plus tard que dans $Seq_{[m]}^{S(I)}$:
***Si** $H(t)$, **alors** $\forall \tau_i^j, (Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) = t', (t' \geq t)) \Longrightarrow Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S'\longmapsto S(I))}) \leq t'$* ■

Corollaire
Soit $t_0 \geq t_d$ le plus petit instant tel que $H(t_0)$ soit vérifié. Alors, à partir de t_0, toutes les échéances sont vérifiées dans $S' \longmapsto S(I)$■

En effet, $\forall \tau_i^j$ tel que $d_i^j \geq t_0$, $S(I)$ est valide et donc
$Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) < d_i^j \Longrightarrow Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S'\longmapsto S(I))}) < d_i^j$ (car les échéances sont les mêmes dans $S' \longmapsto S(I)$ et $H(t_0)$ est vraie).

II.7.a. Cas 1 : pas sous-tâche retardée (avec t_p et x quelconques)

Pour étudier la validité de $Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S'\longmapsto S(I))}$, nous nous plaçons d'abord dans le cas où il n'y a pas de sous-tâche retardée (c-à-d $H(t_d)$ est vrai) et nous prouvons le théorème suivant.

Théorème 2. *Si l'instant de survenue de la panne et le délai de détection sont quelconques, s'il n'y a pas de sous-tâches retardées, alors toute sous-tâche est ordonnancée dans sa fenêtre d'exécution :*
Si** $\forall \tau_i^j, Exec(\tau_i^j, Seq_m^{S(I)}) < t_d \Longrightarrow Exec(\tau_i^j, Sched_{(m+1)}^{S'}) < t_d$ **alors
$\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_p}(m)]}^{(S)}) < d_i^j$ ■

Démonstration. (du Théorème 2)
Une sous-tâche τ_i^j n'est ordonnancée que si elle est réveillée et donc
$\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq_{[(m+1)\longmapsto_{t_d}(m)]}^{(S)})$.
Il reste donc à prouver que les sous-tâches respectent leurs pseudo-échéances.
Soit donc τ_i^j une sous-tâche quelconque de S.

— Si τ_i^j est ordonnancée en mode "contraint" (c-à-d $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_p} m]}) < t_d$), alors $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_p}(m)]}) = Exec(\tau_i^j, Seq^{(S')}_{[m+1]}) < d_i^j$ car $Seq^{(S')}_{[m+1]}$ est valide et équitable.

— Si τ_i^j est ordonnancée en mode "récupération" alors, par hypothèse on a $H(t_d)$ et donc, d'après le corollaire ci-dessus, toutes les échéances sont respectées, d'où le résultat.

— Si τ_i^j est ordonnancée en mode "normal"(c-à-d $Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_p} m]}) \geq Next_H$) alors, on a
$Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_p}(m)]}) = Exec(\tau_i^j, Seq^{(S)}_{[m]}) < d_i^j$, car $Seq^{(S)}_{[m]}$ est valide et équitable.

En conclusion, lorsqu'il n'y a pas de sous-tâches retardées, dans tous les modes d'exécution, les sous-tâches sont ordonnancées dans leurs fenêtres respectives et donc le théorème est vérifié.

□

II.7.b. Cas 2 : existence de sous-tâches retardées (avec $t_p = 0$ et $x = 1$ quelconque)

Plaçons-nous maintenant dans le cas où il y a des sous-tâches retardées. La preuve étant très technique, dans un premier temps, nous considérons que la panne survient à l'instant d'une hyperpériode ($t_p = 0$), qu'elle est détectée l'instant d'après (i.e $x = 1$) et qu'il y a une unique sous-tâche retardée. Nous présentons ensuite quelques éléments de généralisation dans le paragraphe suivant. Nous prouvons donc le théorème suivant :

Théorème 3. ***Si*** $t_p = 0$, $x = 1$ *et* $\exists! \tau_a^0$, $(Exec(\tau_a^0, Seq^{(S(I))}_{[m]}) = 0) \wedge (Exec(\tau_a^0, Seq^{(S')}_{[(m+1)]}) > 0)$ ***alors*** $\forall \tau_i^j, r_i^j \leq Exec(\tau_i^j, Seq^{(S)}_{[(m+1)\longmapsto_{t_d}(m)]}) < d_i^j$ ■

Démonstration. (du Théorème 3)
Comme nous l'avons mentionné dans la preuve du théorème précédent, une sous-tâche τ_i^j n'est ordonnancée que si elle est réveillée. Il reste donc à prouver que les sous-tâches respectent leurs pseudo-échéances. Pour les modes "contraint" et "normal", la preuve est identique à celle du théorème précédent. Nous nous intéressons donc au mode récupération.

Pour commencer, rappelons qu'à l'instant $t_p = 0$, toutes les tâches sont activées via leurs premières sous-tâches τ_i^0. Nous qualifions de $N^o k$, toutes les sous-tâches τ_i^j du système telles que $j = k$. Par exemple, les premières sous-tâches τ_i^0 des tâches sont des $N^o 0$.
S'il y a une seule sous-tâche retardée τ_a^0, alors à l'instant 0, il y a $m - 1$ sous-tâches qui sont ordonnancées à la fois dans $Seq^{(S(I))}_{[m]}$ et dans $Seq^{(S')}_{[(m+1)]}$. E_0 représente l'ensemble de telles sous-tâches. Puisque dans $Seq^{(S')}_{[(m+1)]}$ sont ordonnancées $m + 1$ sous-tâches (en effet, il ne peut pas y

avoir d'unité de temps creux car la sous-tâche retardée n'est pas encore ordonnancée), il y deux sous-tâches τ_b^0 et τ_c^0 qui sont ordonnancées à l'instant 0 dans $Seq_{[(m+1)]}^{(S')}$ mais pas dans $Seq_{[m]}^{(S(I))}$. Ce sont des sous-tâches anticipées. C'est ce qui est illustré dans la Figure 6.12. D'où :
$\exists \tau_b^0$ et τ_c^0 telles que $Exec(\tau_b^0, Seq_{[(m+1)]}^{(S')}) = Exec(\tau_c^0, Seq_{[(m+1)]}^{(S')}) = 0$,
$(t1 = Exec(\tau_b^0, Seq_{[m]}^{(S(I))}) > 0) \wedge (t2 = Exec(\tau_c^0, Seq_{[m]}^{(S(I))}) > 0)$ avec $t_1 \leq t_2$.

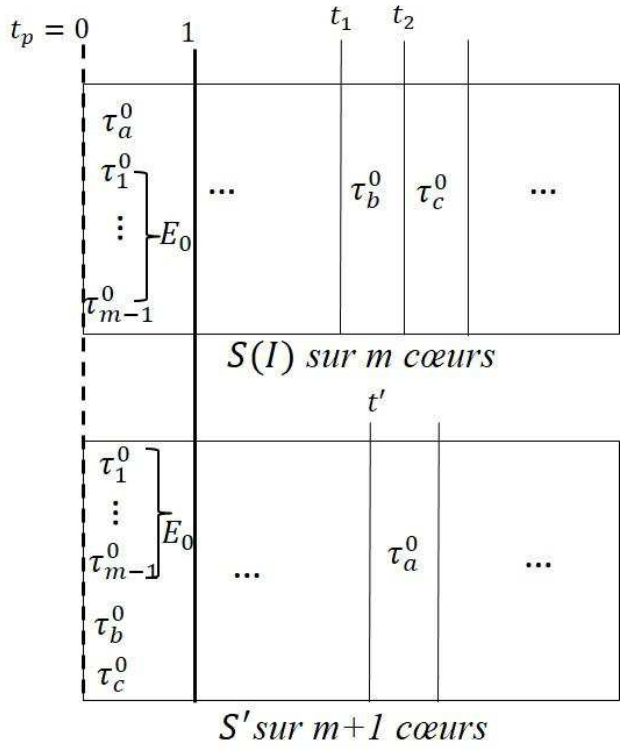

FIGURE 6.12 – Sous-tâches ordonnancées à l'instant $t_p = 0$ dans $S(I)$ et S'

Pour démontrer le théorème, nous utiliserons le lemme suivant.

Lemme 3. *$L(t)$:*
Soit $t1 = Exec(\tau_b^0, Seq_{[m]}^{(S(I))})$, s'il existe $t \times m$ sous-tâches de pseudo-échéances $\leq t$ (avec $t \leq t1$), alors la charge $U(t)$ induite par les tâches qui ont des exécutions dans $[0, t]$ est telle que $U(t) \geq m$
■

La preuve de ce lemme est technique, c'est pourquoi elle est donnée en annexe.

Nous prouvons ensuite le Théorème 3 par induction.

— A l'instant $t_d = 1$
Il y a reconfiguration et donc, les sous-tâches ont les mêmes fenêtres d'exécution et donc le même ordre de priorité dans $S(I)$ et dans $S' \longmapsto S(I)$. On note S_1 la liste des sous-tâches prêtes à l'instant 1 à la fois dans $S' \longmapsto S(I)$ et dans $S(I)$. On a les relations suivantes :
$Pretes(S' \longmapsto S(I), 1) = S_1 \bigcup \{\tau_a^0\}$ et $Pretes(S(I), 1) = S_1 \bigcup \{\tau_b^0, \tau_c^0\}$.
Nous montrons que dans $S' \longmapsto S(I)$:

✓ La sous-tâche retardée τ_a^0 est ordonnancée : $Exec(\tau_a^0, Seq_{[m]}^{(S' \longmapsto S(I))}) = 1$.
En effet, considérons les m sous-tâches prêtes les plus prioritaires de $S' \longmapsto S(I)$.
Si τ_a^0 en fait partie, alors elle est ordonnancée et on a $Exec(\tau_a^0, Seq_{[(m+1) \longmapsto t_d(m)]}^{(S' \longmapsto S(I))}) = 1$.
Sinon, les m sous-tâches prêtes les plus prioritaires de $S' \longmapsto S(I)$ appartiennent à S_1. Ces sous-tâches sont réveillées à l'instant 1 (et donc sont des N^o1). En effet, si l'une d'elle était activée à l'instant 0, elle aurait été prête et plus prioritaire que τ_a^0 à l'instant 0 dans $S(I)$ et donc aurait été ordonnancée dans $S(I)$ à la place de τ_a^0 et par conséquent, ne serait plus présente dans S_1. Donc, il s'agit bien des sous-tâches N^o1.
Si donc on suppose que les m sous-tâches les plus prioritaires sont des sous-tâches de S_1 et donc des N^o1, leurs pseudo-échéances d_i^1 sont inférieures ou égales à celle de τ_a^0. Mais dans ce cas, les pseudo-échéances de leurs précédentes τ_i^0 sont inférieures ou égales à celle de τ_a^0. Donc, à l'instant 0, on a m sous-tâches τ_i^0 plus prioritaires que τ_a^0 dans $S(I)$ et donc τ_a^0 n'est pas ordonnancé dans $S(I)$, ce qui est contradictoire.
En définitive, à l'instant 1, τ_a^0 fait partie des m sous-tâches les plus prioritaires de $S' \longmapsto S(I)$ et donc est ordonnancée.

✓ La sous-tâche retardée τ_a^0 respecte sa pseudo-échéance.
En effet, puisque $Exec(\tau_a^0, Seq_{[(m+1)]}^{(S')}) > 0$, τ_a^0 est ordonnancé dans S' au plus tôt à l'instant 1 et avant $d_a'^0$, car $Seq_{[(m+1)]}^{(S')}$ est valide et équitable. D'où, $d_a'^0 > 1$. Or, $d_a^0 \geq d_a'^0$ et donc $Exec(\tau_a^0, Seq_{[(m+1) \longmapsto t_d(m)]}^{(S' \longmapsto S(I))}) = 1 < d_a^0$.

— Si $t_1 = 1$: la sous-tâche anticipée τ_b^0 est ordonnancée dans $Seq_{[m]}^{(S(I))}$ à l'instant 1.
Les m sous-tâches les plus prioritaires de $S(I)$ comprennent $m-1$ sous-tâches de S_1 et la sous-tâche τ_b^0 ; celles de $S' \longmapsto S(I)$ comprennent les mêmes $m-1$ sous-tâches de S_1 et la sous-tâche τ_a^0. Donc, comme le montre la Figure 6.13 (a) dans $Seq_{[(m+1) \longmapsto t_d(m)]}^{(S' \longmapsto S(I))})$, τ_a^0 remplace τ_b^0 dans l'ordonnancement. En plus de τ_a^0 qui respecte sa pseudo-échéance, les autres sous-tâches sont ordonnancées à la même date que dans $Seq_{[m]}^{(S(I))}$ et donc avant leur pseudo-échéance. Toutes les sous-tâches ordonnancées dans $Seq_{[m]}^{(S(I))}$ sont aussi déjà ordonnancées dans $Seq_{[(m+1) \longmapsto t_d(m)]}^{(S' \longmapsto S(I))})$. La propriété $P(1)$ s'applique et donc $\forall \tau_i^j, Exec(\tau_i^j, Seq_{[(m+1) \longrightarrow t_d(m)]}^{(S' \longmapsto S(I))}) \leq Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) < d_i^j$. (voir Propriété 1)
Le théorème est vérifié.

Si par contre $t_1 > 1$, étant donné que dans $Seq_{[m]}^{(S(I))}$ la sous-tâche τ_b^0 n'a pas été ordonnancée à l'instant 1, les m sous-tâches ordonnancées sont toutes des sous-tâches de S_1. Or dans $Seq_{[(m+1)\longmapsto_{t_d}(m)]}^{(S'\longmapsto S(I))}$, il y a eu $m-1$ sous-tâches de S_1 ordonnancées avec τ_a^0. Donc, comme le montre la Figure 6.13 (b), il y a une sous-tâche $\tau_{u_1}^{v_1}$ ordonnancée à l'instant 1 dans $Seq_{[m]}^{(S(I))}$ qui n'est pas ordonnancée dans $Seq_{[(m+1)\longmapsto_{t_d}(m)]}^{(S'\longmapsto S(I))}$. C'est une sous-tache reportée, en l'occurrence la m^e sous-tâche de S_1 dans l'ordre de priorité. Ceci se traduit par :
$\exists \tau_{u_1}^{v_1}, (Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S(I))}) = 1) \wedge (Exec(\tau_{u_1}^{v_1}, Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S'\longmapsto S(I))}) > 1)$.

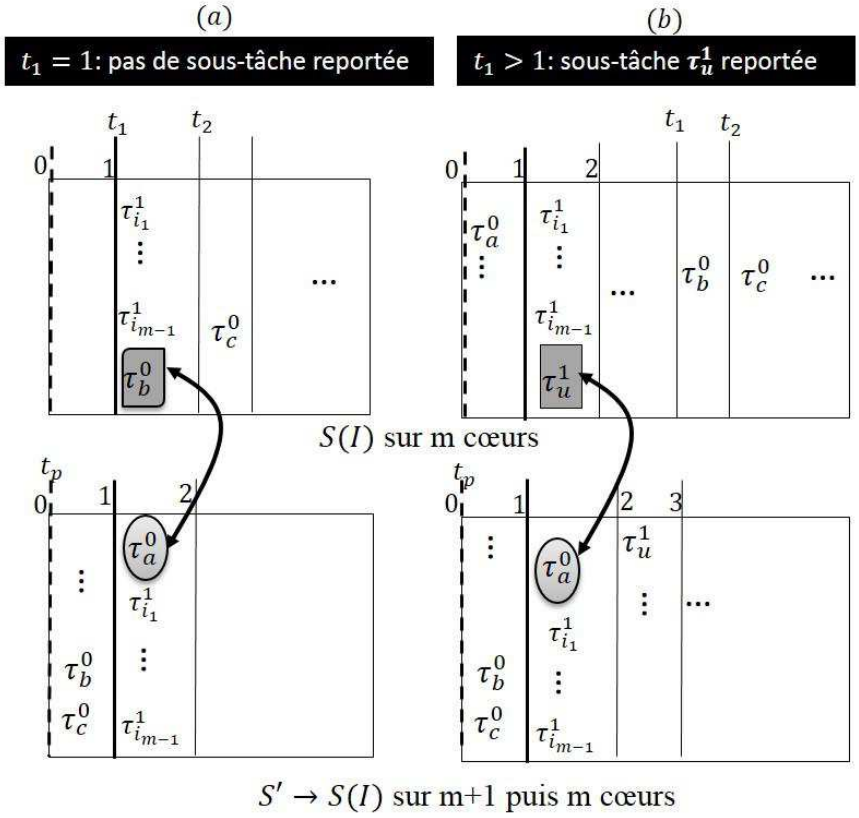

FIGURE 6.13 – Scénarios d'ordonnancement de la sous-tâche retardée à $t_d = 1$

— Si $t_1 = 2$: la sous-tâche anticipée τ_b^0 est ordonnancée dans $Seq_{[m]}^{(S(I))}$ à l'instant 2.
On note S_2 la liste des sous-tâches prêtes à la fois dans $S' \longmapsto S(I)$ et dans $S(I)$. On a :
$Pretes(S' \longmapsto S(I), 2) = S_2 \bigcup \{\tau_u^1\}$.
$Pretes(S(I), 2) = S_2 \bigcup \{\tau_b^0, \tau_c^0\}$.
On montre que :

✓ La sous-tâche $\tau_{u_1}^{v_1}$ reportée à l'instant 1 est ordonnancée : $Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S'\longmapsto S(I))}) = 2$.
En effet, à l'instant $t = 1$ on a dans $S(I)$, $\tau_{u_1}^{v_1}$ et τ_b^0 prêtes. $\tau_{u_1}^{v_1}$ est ordonnancée (car $Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S(I))}) = 1$) et τ_b^0 ne l'est pas (car $t_1 = 2$). Donc, la sous-tâche $\tau_{u_1}^{v_1}$ est plus prioritaire que la sous-tâche τ_b^0.
A l'instant 2, les m sous-tâches prêtes les plus prioritaires de $S(I)$ comprennent τ_b^0 (et τ_c^0, si $t_1 = t_2 = 2$), ainsi que $(m-1)$ sous-tâches de S_2. Ces sous-tâches de S_2 ont la même priorité dans $S' \longmapsto S(I)$ et donc sont dans les m sous-tâches les plus prioritaires. Comme $\tau_{u_1}^{v_1}$ est plus prioritaire que τ_b^0, $\tau_{u_1}^{v_1}$ fait partie des m sous-tâches prêtes les plus prioritaires de $S' \longmapsto S(I)$ et donc $Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S'\longmapsto S(I))}) = 2$.

✓ La sous-tâche reportée $\tau_{u_1}^{v_1}$ respecte son échéance : $Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S'\longmapsto S(I))}) < d_{u_1}^{v_1}$.
En effet, puisque $Exec(\tau_u^1, Seq_{[m]}^{(S(I))}) = 1$ et $Seq_{[m]}^{(S(I))}$ est valide et équitable, on a $d_{u_1}^{v_1} \geq 2$.
Si on avait $d_{u_1}^{v_1} = 2$, cela voudrait dire qu'à l'instant 1, les m sous-tâches ordonnancées dans $Seq_{[m]}^{(S(I))}$ ont une pseudo-échéance de 2 et donc sont des sous-tâches N^o0 (car d'après l'équation (5.2) on a $u_i < 1\ \forall\tau_i$). Or les m sous-tâches ordonnancées à l'instant 0 sont aussi des N^o0 et donc ont une pseudo-échéance de 2. Ce qui fait dans l'intervalle de temps $[0, 1]$ un total de $2 \times m$ sous-tâches ordonnancées appartenant à des tâches différentes qui ont une pseudo-échéance égale à 2. D'après le lemme L(1), cela induit une charge $U(t) \geq m$. Puisque à l'instant 1 τ_b et τ_c ne sont pas déjà ordonnancée dans $Seq_{[m]}^{(S(I))}$, on a : $U_{S(I)} \geq U(t) + u_b + u_c > m$ et donc $S(I)$ n'est pas ordonnançable sur m cœurs, ce qui est absurde. Conclusion : $d_{u_1}^{v_1} > 2$ et donc $Exec(\tau_{u_1}^{v_1}, Seq_{[m]}^{(S'\longmapsto S(I))}) = 2 < d_{u_1}^{v_1}$.

En définitive, si $t_1 = 2$ les m sous-tâches ordonnancées dans $Seq_{[(m+1)\longmapsto t_d(m)]}^{(S'\longmapsto S(I))}$ comprennent $(m-1)$ sous-tâches de S_2 ordonnancées comme dans $Seq_{[m]}^{(S(I))}$ (donc avant leurs pseudo-échéances) et $\tau_{u_1}^{v_1}$ ordonnancée (avant sa pseudo-échéance) à la place de τ_b^0. Toutes les sous-tâches ordonnancées dans $Seq_{[m]}^{(S(I))}$ sont aussi déjà ordonnancées dans $Seq_{[(m+1)\longmapsto t_d(m)]}^{(S'\longmapsto S(I))}$). La propriété $P(2)$ s'applique et on a :
$\forall\tau_i^j, Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow t_d(m)]}^{(S'\longmapsto S(I))}) \leq Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) < d_i^j$.
Le théorème est vérifié.
Si par contre $t1 > 2$, $\tau_{u_1}^{v_1}$ est ordonnancée à la place d'une autre sous-tâche $\tau_{u_2}^{v_2}$ et l'analyse se poursuit aux instants suivants.

— De l'instant $t = 3$ à l'instant $t_1 - 1$:
Il n' y a pas d'unités de temps creux dans $Seq_{[m]}^{(S(I))}$ car τ_b^0 n'est pas encore ordonnancée. On itère le même raisonnement. Si $\tau_{u_{t-1}}^{v_{t-1}}$ désigne la sous-tâche reportée à l'instant t-1, on montre que dans $Seq_{[(m+1)\longrightarrow t_d(m)]}^{(S'\longmapsto S(I))}$:

- ✓ $\tau_{u_{t-1}}^{v_{t-1}}$ est parmi les m sous-tâches les plus prioritaires de $S' \longmapsto S(I)$ et donc $Exec(\tau_{u_{t-1}}^{v_{t-1}}, Seq_{[m]}^{(S' \longmapsto S(I))}) = t$.
- ✓ $d_{u_{t-1}}^{v_{t-1}} > t$ et donc $\tau_{u_{t-1}}^{v_{t-1}}$ respecte son échéance (grâce au Lemme 1).
- ✓ $\tau_{u_{t-1}}^{v_{t-1}}$ est ordonnancée à la place d'une sous-tâche $\tau_{u_t}^{v_t}$ qui est reportée (car τ_b^0 n'est toujours pas ordonnancée dans $Seq_{[m]}^{(S(I))}$).
- ✓ les $m-1$ sous-taches ordonnancées avec $\tau_{u_{t-1}}^{v_{t-1}}$ respectent leurs échéances car elles sont ordonnancées à la même date que dans $Seq_{[m]}^{(S(I))}$.

— A l'instant t_1 où τ_b^0 est ordonnancée dans $Seq_{[m]}^{(S(I))}$, le théorème est vérifié.
En effet, il existe une sous-tâche reportée τ_u^v qui ordonnancée dans $Seq_{[m]}^{(S(I))}$ à l'instant $t_1 - 1$, mais qui n'est pas encore ordonnancée dans $Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S' \longmapsto S(I))}$. On montre par le raisonnement précédent que :

- ✓ τ_u^v est parmi les m sous-tâches les plus prioritaires de $S' \longmapsto S(I)$ et donc $Exec(\tau_u^v, Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S' \longmapsto S(I))}) = t_1$.
- ✓ τ_u^v respecte son échéance car $d_u^v > t_1$. En effet, si on avait $d_u^v = t_1$, il y aurait $t_1 \times m$ sous-tâches de pseudo-échéances inférieures ou égales à t_1 et d'après le lemme $L(t_1)$, $U(t_1) \geq m$ et donc : $U_{S(I)} \geq U(t_1) + u_b + u_c > m$ et donc $S(I)$ ne serait pas ordonnançable sur m cœurs, ce qui est absurde.
- ✓ τ_u^v est ordonnancée avec $m-1$ autres sous-tâches et à la place de τ_b^0. Il n'y a pas de report de sous-tâche.
- ✓ $m-1$ sous-taches ordonnancées avec $\tau_{u_{t-1}}^{v_{t-1}}$ respectent leurs échéances car elles sont ordonnancées à la même date que dans $Seq_{[m]}^{(S(I))}$.
- ✓ Toutes les sous-tâches ordonnancées dans $Seq_{[m]}^{(S(I))}$ sont aussi déjà ordonnancées dans $Seq_{[(m+1)\longmapsto_{t_d}(m)]}^{(S' \longrightarrow S(I))}$). La propriété $P(t_1)$ s'applique et on a :
$\forall \tau_i^j, Exec(\tau_i^j, Seq_{[(m+1)\longrightarrow_{t_d}(m)]}^{(S' \longmapsto S(I))}) \leq Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) < d_i^j$.
Le théorème est vérifié.

□

II.7.c. Principe de généralisation aux autres cas

Dans ce paragraphe, nous indiquons comment généraliser la preuve de la technique TCR en présence de plusieurs sous-tâches retardées. Dans un premier temps, nous maintenons $t_p = 0[H]$ et $x = 1$. L'idée générale de la preuve est similaire à celle du Théorème 3.
Tout d'abord, on remarque qu'à l'instant $t_p = 0[H]$ s'il y a un nombre k de sous-tâches retardées ordonnancées dans $Seq_{[m]}^{(S(I))}$ et pas dans $Seq_{[(m+1)]}^{(S')}$, il y a $m-k$ sous-tâches ordonnancées dans

les deux séquences. Puisque $m+1$ sous-tâches au total sont ordonnancées dans $Seq^{(S')}_{[(m+1)]}$, il y a $k+1$ ($=(m+1)-(m-k)$) sous-tâches ordonnancées dans $Seq^{(S')}_{[(m+1)]}$ mais pas dans $Seq^{(S(I))}_{[m]}$. Donc à k sous-tâches retardées $\tau^{w_1}_{r_1}...\tau^{w_k}_{r_k}$ correspondent $k+1$ sous-tâches anticipées, ce qui se traduit par :
$\exists \tau^{z_1}_{a_1}...\tau^{z_{k+1}}_{a_{k+1}}$, $(Exec(\tau^{z_j}_{a_i}, Seq^{(S')}_{[(m+1)]}) < t_d) \wedge (t^{z_j}_{a_i} = Exec(\tau^{z_j}_{a_i}, Seq^{(S(I))}_{[m]}) \geq t_d)$,
avec $t^{z_1}_{a_1} \leq t^{z_2}_{a_2} \leq ... \leq t^{z_{k+1}}_{a_{k+1}}$. Posons $t_{max} = max(t^{z_j}_{a_i})$.
Dans $Seq^{(S(I))}_{[m]}$, l'ordonnancement des sous-tâches anticipées est prévu plus tard que l'instant 0 et peut-être à des instants différents. Notons k_i le nombre de sous-tâches anticipées ordonnancées dans $Seq^{(S(I))}_{[m]}$ à l'instant $t=i$. $\sum k_i = k+1$.
On montre que dans la séquence $Seq^{(S' \longmapsto S(I))}_{[(m+1) \longmapsto_{t_d}(m)]}$:

— A l'instant $t_d = 1$, toutes les sous-tâches retardées sont ordonnancées avant leurs pseudo-échéances.
En effet, ces sous-tâches sont réveillées à $t_p = 0$ mais n'ont pas ordonnancées dans $Seq^{(S' \longmapsto S(I))}_{[(m+1) \longmapsto_{t_d}(m)]}$. Elles sont donc prêtes l'instant $t_d = 1$ et plus prioritaires que les autres sous-tâches de $S' \longmapsto S(I)$ non ordonnancées à l'instant 0, car elles sont déjà ordonnancées dans $Seq^{(S(I))}_{[m]}$. Elles sont aussi plus prioritaires que les sous-tâches prêtes réveillées à l'instant 1 car si ce n'était pas le cas, les précédentes de ces sous-tâches auraient été plus prioritaires dans $S(I)$ à l'instant 0 et on aurait pas eu $Exec(\tau^{w_j}_{r_i}, Seq^{(S(I))}_{[m]}) < t_d$.
Donc les sous-tâches retardées sont les sous-tâches prêtes les plus prioritaires de $S' \longmapsto S(I)$ et puisque $k < m$, elles sont ordonnancées et on a
$Exec(\tau^{w_j}_{r_i}, Seq^{(S' \longmapsto S(I))}_{[(m+1) \longrightarrow_{t_p}(m)]}) = t_d$.

Les pseudo-échéances sont respectées. En effet, dans la séquence $Seq^{(S')}_{[(m+1)]}$, les sous-tâches retardées sont ordonnancées après l'instant $t_p = 0$, soit à un instant $t \geq 1$. Cette séquence étant valide et équitable, les pseudo-échéances des sous-tâches retardées sont donc supérieures à 1.

— Soit k_1 le nombre de sous-tâches anticipées ordonnancées dans $Seq^{(S(I))}_{[m]}$ à l'instant $t = 1$. Si $k_1 \geq k$, les k sous-tâches retardées sont ordonnancées dans $Seq^{(S' \longmapsto S(I))}_{[(m+1) \longrightarrow_{t_p}(m)]}$ à la place de k sous-tâches anticipées. Toutes les sous-tâches ordonnancées dans $Seq^{(S(I))}_{[m]}$ sont aussi ordonnancées dans $Seq^{(S' \longmapsto S(I))}_{[(m+1) \longrightarrow_{t_p}(m)]}$. La propriété $P(1)$ s'applique et permet de conclure que toutes les sous-tâches respectent leurs pseudo-échéances. La démonstration s'arrête à ce niveau.

— Si $k_1 < k$, les k sous-tâches retardées sont ordonnancées à la place de k_1 sous-tâches anticipées et de $k - k_1$ autres sous-tâches qui sont reportées. Comme on l'a montré dans le Théorème 3, une sous-tâche reportée à un instant t est ordonnancée à l'instant $t+1$ en respectant sa pseudo-échéance. On peut assister ainsi à des reports en cascade de sous-tâches de l'instant $t = 2$ à l'instant $t_{max} - 1$.

A l'instant t_{max}, toutes les $k+1$ sous-tâches anticipées ont été ordonnancées dans $Seq_{[m]}^{(S(I))}$. A leurs places dans $Seq_{[(m+1)\longrightarrow t_p(m)]}^{(S' \longmapsto S(I))}$, les k sous-tâches retardées et les sous-tâches reportées ont été ordonnancées. La propriété $P(t_{max})$ s'applique et permet de conclure que toutes les sous-tâches respectent leurs pseudo-échéances.

La preuve de la technique TCR en présence de sous-tâches retardées pour un instant de panne et un délai de détection quelconque (t_p et x quelconques) est en cours d'élaboration. Pour généraliser le raisonnement précédent à ces cas de figure, il nous reste à établir les résultats observés expérimentalement suivants :

— si $x > 1$, (même avec $t_p = 0$) le nombre de sous-tâches anticipées est supérieur au nombre de sous-tâches retardées ;
— les sous-tâches retardées sont toutes ordonnancées à l'instant t_d. Pour cela, il faut montrer qu'elles sont plus prioritaires que les autres sous-tâches non élues dans $S' \longmapsto S(I)$ à l'instant $t_d - 1$. En effet, à la reconfiguration, on change d'ordre de priorité et donc il y a possibilité d'inversion de priorité entre une sous-tâche retardée et une autre sous-tâche non élue. De plus, si $x > 1$, il faut montrer que le nombre de sous-tâches retardées est plus petit que le nombre de coeur ($k < m$).
— une sous-tâche reportée à un instant t est ordonnancée plus tard avant sa pseudo-échéance.

III. La Technique du Flux Apériodique

Les techniques précédentes sont conçues pour récupérer les sous-tâches perdues sans faute temporelle, ce qui impose des restrictions aux systèmes ordonnançables (conditions (2.2) et (6.1) pour les techniques *TSS* et *TCR* et conditions (6.3) et (6.4) en plus pour *TCR*). Si la technique *TCR* n'est pas applicable alors que *TSS* l'est, le système à ordonnancer est surchargé et nécessite davantage de redondance matérielle. Notre objectif est de proposer une technique qui peut s'appliquer à tout système ordonnançable (c-à-d respectant uniquement la condition (2.2)) avec seulement un cœur redondant (c-à-d $\mu = 1$). Pour ce faire, nous autorisons un nombre borné de fautes temporelles : seules les instances des tâches impactées par la panne peuvent s'exécuter après leurs échéances. Inspirée des travaux de Choquet-Geniet et Malo [176], la présente technique utilise un flux apériodique pour ré-ordonnancer les sous-tâches perdues dans les unités de temps creux. Une unité de temps creux est un instant pendant lequel un cœur reste libre parce qu'aucune tâche ne lui a été affectée. Comme on peut le voir sur la Figure 6.14, l'exécution du système démarre sur $m+1$ cœurs dans le mode "normal". Lorsque la panne est détectée (à l'instant t_d), le système bascule dans en mode "récupération" où les tâches gardent leurs paramètres d'origine, mais un flux apériodique constitué des sous-tâches

perdues est ajouté. Lorsque toutes les sous-tâches perdues du flux ont été réordonnancées, le système retourne en mode normal.

Nos hypothèses sont les suivantes :

1. La condition (2.2) est satisfaite et donc le système S est ordonnançable sur m cœurs.
2. Il y a des unités de temps creux dans une hyperpériode et donc

$$IT = (m - U) \times H > 0 \tag{6.4}$$

Le choix d'un nombre minimal de cœurs $m = \lfloor U \rfloor + 1$ assure la satisfaction de ces deux hypothèses. Pour gérer le flux de sous-tâches à reprendre, nous devons avoir une idée, pour un intervalle de temps donné, du nombre d'unités de temps creux et de leur localisation. Ce problème est abordé dans le paragraphe qui suit.

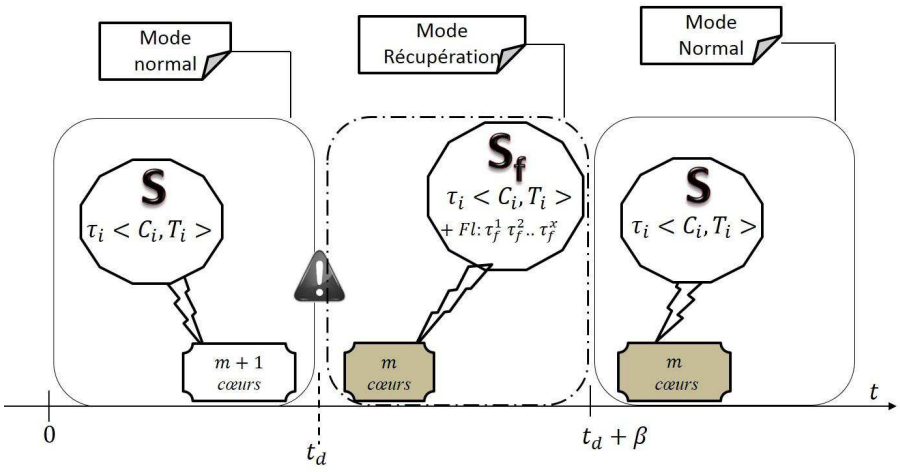

FIGURE 6.14 – Ordonnancement PD2 avec panne et reprise par la technique TFA

III.1. Répartition des temps creux

Si nous considérons la répartition par défaut dans laquelle les unités de temps creux surviennent dans l'ordonnancement aux instants où le nombre de cœurs fonctionnels est supérieur au nombre de sous-tâches prêtes, nous n'avons aucune information et aucun contrôle sur leur emplacement dans la séquence d'ordonnancement, ce qui représente un problème pour cette technique. En effet, avec une telle répartition, la panne peut être détectée au moment où toutes les unités de temps creux de l'hyperpériode courante se sont produits et il faut attendre l'hyperpériode suivante pour que la récupération débute. Inversement, les unités de temps creux peuvent être toutes regroupées à la fin de l'hyperpériode alors que la panne a été détectée

longtemps à l'avance. Dans ce dernier cas, la récupération débutera plus tardivement. Notre objectif est cependant de ré-ordonnancer les sous-tâches perdues le plus tôt possible après la détection de la panne, ce qui nécessite de connaitre et de gérer la répartition des unités de temps creux. Dans [176], les auteurs proposent une méthode de répartition équitable des unités de temps creux basée sur l'ajout d'une tâche $\tau_c < IT, H >$ au système. Cette tâche, appelée *tâche creuse*, a pour WCET le nombre d'unités de temps creux sur une hyperpériode et pour période l'hyperpériode du système. Son facteur d'utilisation est $u_c = \frac{IT}{H}$. La tâche creuse τ_c est ordonnancée par PD2 avec les autres tâches du système et une unité de temps creux est identifiée à chaque fois qu'une sous-tâche de τ_c est affectée à un cœur. Les unités de temps creux sont ainsi identifiées par les sous-tâches de τ_c et leurs emplacements peuvent être gérés par l'algorithme d'ordonnancement. De plus, le nombre minimal d'unités de temps creux entre un instant t et un instant t' peut être calculé par la formule suivante [176] :

$$W_{min}(t, t') = \lfloor U_0 \times t' \rfloor - \lceil U_0 \times t \rceil . \tag{6.5}$$

Cette formule sera utile pour déterminer le délai maximal de récupération des sous-tâches perdues. Cette répartition équitable présente aussi l'avantage de garantir qu'il n'y a pas plusieurs unités de temps creux au même instant et donc, telle que l'exige la règle du non parallélisme, des sous-tâches d'une même tâche ne risquent pas d'être ordonnancées au même instant.

III.2. Gestion du parallélisme d'exécution

Comme nous venons de le mentionner, en ordonnancement multiprocesseur ou multicoeur, le parallélisme d'exécution est interdit. Cela signifie qu'il n'est pas possible pour une tâche de s'exécuter sur des cœurs différents au même instant. Ce phénomène peut cependant apparaître avec la technique actuelle parce qu'après la détection de la panne, les sous-tâches perdues dans le flux sont en attente d'exécution en même temps que les autres sous-tâches prêtes du système. C'est pourquoi, comme le montre la Figure 6.15, lorsqu'une unité de temps creux est repérée sur un cœur, une sous-tâche du flux peut être ordonnancée sur ce cœur au même instant qu'une sous-tâche prête de la même tâche est ordonnancée sur un autre cœur. Dans la suite, nous dirons que ces deux sous-tâches sont en *concurrence d'exécution*. De plus, nous distinguons le non-parallélisme *partiel* du non-parallélisme *total*. Dans le non-parallélisme partiel, deux sous-tâches sont en concurrence d'exécution uniquement si elles appartiennent à la même instance de tâche. Ainsi, deux sous-tâches appartenant à des instances différentes d'une même tâche peuvent être ordonnancées sur des cœurs différents au même instant. Dans le non-parallélisme total par contre, deux sous-tâches sont en concurrence d'exécution si elles appartiennent à la même tâche. Donc, deux sous-tâches de la même tâche ne peuvent pas être ordonnancées au même instant sur des cœurs différents, qu'elles appartiennent ou pas à des instances différentes. Comme on peut l'observer sur l'exemple du système Θ (Figure 6.18), le non parallélisme total peut engendrer un effet domino : à chaque fois qu'une unité de temps creux est identifiée, il y a concurrence d'exécution entre les sous-tâches du flux et les sous-tâches prêtes du système. Si un

tel effet se produit, il est impossible de réordonnancer les sous-tâches perdues. Pour éviter cela, nous choisissons la règle de non-parallélisme partiel dans la présente technique. Ce choix n'est pas sans fondement car les langages de programmation actuels offrent la possibilité d'exécuter en parallèle deux processus (c-à-d deux instances) d'une même tâche sur un processeur multicoeur.

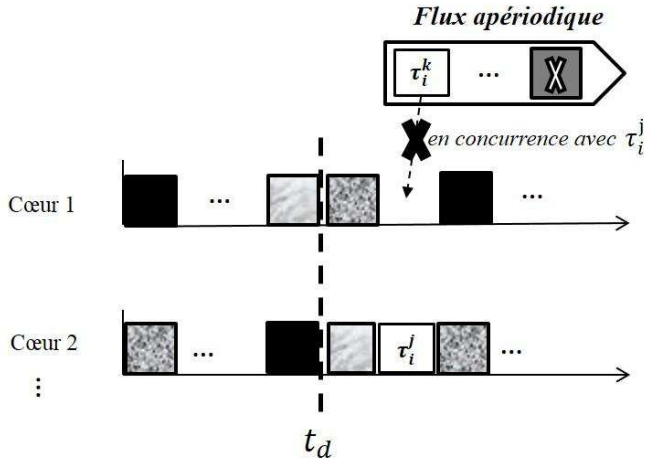

FIGURE 6.15 – Illustration de la concurrence d'exécution entre sous-tâches

Basée sur la répartition équitable des unités de temps creux, l'idée principale de la technique *TFA* est la suivante : lorsqu'une sous-tâche creuse est élue (c-à-d qu'une unité de temps creux est identifiée), on ordonnance à sa place la sous-tâche la plus prioritaire du flux qui n'est pas en concurrence d'exécution avec les autres sous-tâches élues du système. Pour des raisons de cohérence dans la séquence d'ordonnancement, nous convenons que durant la phase de récupération, lorsque deux sous-tâches de la même instance sont prêtes, la sous-tâche réveillée le plus tôt (qui donc a le plus petit index et se trouve dans le flux) est ordonnancée et l'autre est placée dans le flux. Le protocole de reconfiguration P_{TFA} suivant est défini.

Protocole P_{TFA}

Dans le mode "récupération", à tout instant $t \geq t_d$:

1. **Cohérence** : pour toute sous-tâche prête τ_i^j parmi les m les plus prioritaires du système, s'il existe une sous-tâche τ_i^k du flux appartenant à la même tâche (donc telle que $k < j$), alors τ_i^k est élue et τ_i^j est ajoutée au flux ;
2. **Ré-ordonnancement** : si une sous-tâche creuse τ_c^q est parmi les m sous-tâches les plus prioritaires du système et s'il existe une sous-tâche τ_f^h qui est la plus prioritaire du flux parmi les sous-tâches qui ne sont pas en concurrence d'exécution avec l'une des $(m-1)$

autres sous-tâches prioritaires, alors τ_f^h est ordonnancée à la place de τ_c^q. Si toutes sous-tâches du flux sont en concurrence d'exécution avec l'une des $(m-1)$ autres sous-tâches prioritaires, la sous-tâche τ_c^q est ordonnancée, ce qui correspond à une unité de temps creux effective.

III.3. Algorithme d'ordonnancement

L'Algorithme 5 effectue l'ordonnancement PD2 d'un système par la technique *TFA*. Les primitives utilisées sont celles de la section précédente. La primitive $reconfigurer()$ est redéfinie. Elle applique le protocole de reconfiguration P_{TFA} pour déterminer à chaque instant la liste des sous-tâches à ordonnancer en fonction des sous-tâches prêtes à cet instant et des sous-tâches du flux.

III.4. Exemple

Considérons à nouveau le système $\psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$ avec $x = 2$ et $t_d = 3$ avec panne sur le coeur $C3$.
Le nombre total d'unités de temps-creux par hyperpériode est $IT = 10$. Pour une répartition équitable de ces unités de temps creux, la tâche creuse $\tau_c < 10, 12 >$ est ajoutée au système. Comme le montre la Figure 6.16, les sous-tâches τ_2^0 et τ_4^1 perdues aux instants 1 et 2 sont ré-ordonnancées respectivement aux instants $t_1 = 3$ et $t_2 = 4$ en remplacement des sous-tâches τ_c^3 et τ_c^4. Il ne s'est produit aucune concurrence d'exécution et la phase de récupération a duré 2 unités de temps. La séquence générée est valide mais pas équitable car la sous-tâche τ_2^0 ne respecte pas sa pseudo-échéance mais toutes les autres sous-tâches respectent les leurs (voir les fenêtres d'exécution dans le Tableau 1.1).

Pour mettre en évidence le phénomène de concurrence d'exécution avec le non-parallélisme partiel, considérons le système ψ avec une panne sur $C3$ détectée cette fois à l'instant $t_d = 2$ après $x = 2$ unités de temps. Les sous-tâches perdues τ_3^0 et τ_2^0 sont ajoutées au flux. La phase de récupération se déroule de la façon suivante :

- A l'instant $t_d = 2$, les sous-tâches prêtes sont τ_c^2, τ_3^1 et τ_4^1 (par ordre de priorité). Le contenu du flux est τ_3^0 et τ_2^0. Il reste 3 cœurs fonctionnels et une unité de temps creux est identifiée (sous-tâche τ_c^2). La sous-tâche τ_3^0 de la tâche τ_3 est la plus prioritaire du flux. Cependant, elle ne peut pas être ordonnancée à la place de τ_c^2 car elle est en concurrence d'exécution avec τ_3^1. C'est donc la sous-tâche τ_2^0 qui sera ordonnancée à la place de τ_c^2 car elle n'est pas en situation de concurrence d'exécution. Par ailleurs, pour la cohérence de la séquence, τ_3^0 est ordonnancée et τ_3^1 intègre le flux.
- A l'instant $t = 3$, une unité de temps creux est identifiée (sous-tâche τ_c^3). La sous-tâche τ_3^1 est la seule du flux et n'est pas en situation de concurrence d'exécution. Elle est donc ordonnancée à la place de τ_c^3.

Algorithm 5: Ordonnancement PD2 avec panne et reprise par la technique TFA

Inputs : $S : \tau_i < C_i, T_i >$: système à ordonnancer ;
$C = (C_1, C_2, .., C_{m+1})$: liste des cœurs du processeur ;
x : délai de détection de la panne.

Output: $Seq = Seq^{(S)}_{[(m+1)\longmapsto t_d(m)]}$: Séquence d'ordonnancement de S sur une hyperpériode.

$ST \leftarrow sousTaches(S)$;
t $\leftarrow 0$;
$t_d \leftarrow 0$;
$init(Seq)$;

// Affectation des sous-tâches avant la panne

while $detection() = -1$ **do** *// i.e tant qu'aucun cœur n'est défaillant*
    P $\leftarrow pretes(ST, t)$;
    $ranger(P)$;
    k $\leftarrow 1$;
    j $\leftarrow 1$;
    while $(k \leq m+1) \wedge (j \leq card(P))$ **do**
        $Seq(C(k), t) \leftarrow P(j)$;
        j $\leftarrow$ j+1 ;
        k $\leftarrow$ k+1 ;
    t $\leftarrow$ t+1 ; $t_d \leftarrow$ t ;

$num \leftarrow detection()$;

// Panne détectée: réconfiguration et poursuite de l'ordonnancement

$Flux \leftarrow \emptyset$;
for $i = t_d - x$ *to* $t_d - 1$ **do** *// constitution de la liste des sous-tâches perdues*
    $Flux \leftarrow Flux \bigcup Seq(C(num), i)$;

$Next_H \leftarrow \lfloor \frac{t_d}{H} \rfloor \times H + H$;

while $t < Next_H$ **do**
    P $\leftarrow pretes(STI, t)$;
    $Elues \leftarrow reconfigurer(P, Flux)$;
    $ranger(Elues)$;
    k $\leftarrow 1$;
    j $\leftarrow 1$;
    while $(k \leq m_x + 1) \wedge (j \leq card(Elues))$ **do**
        if $k \neq num$ **then**
            $Seq(C(k), t) \leftarrow P(j)$;
            j $\leftarrow$ j+1 ;
        k $\leftarrow$ k+1 ;
    t $\leftarrow$ t+1 ;

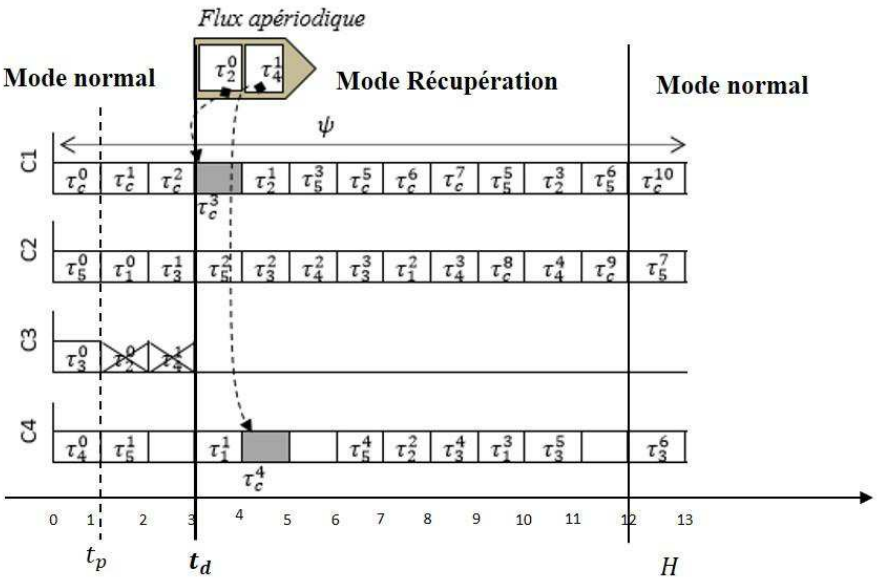

FIGURE 6.16 – Exemple d'ordonnancement par *TFA* sans concurrence d'exécution

La séquence d'ordonnancement construite est présentée dans la Figure 6.17. Cette séquence est valide et équitable car toutes les sous-tâches, y compris τ_3^0 et τ_2^0, respectent leurs pseudo-échéances.
Pour terminer, nous mettons en évidence l'effet domino engendré par le non-parallélisme lors de l'ordonnancement du système suivant :
$\Theta : \tau_0 < 5, 6 >, \tau_1 < 5, 6 >, \tau_2 < 3, 4 >, \tau_3 < 5, 12 >$. On a $U_\Theta = 2.83$ (donc $m = 3$), $H = 12$ et $IT = 2$.
La tâche $\tau_c < 2, 12 >$ est ajoutée pour une répartition équitable des temps creux. Nous considérons une panne sur le cœur C3 à l'instant $t_p = 0$ avec $x = 2$. Les sous-tâches τ_2^0 et τ_2^1 sont perdues. τ_2^0 est rattrapée à l'instant $t = 11$ à la place de la sous-tâche creuse τ_c^2. Cependant, comme le montre la Figure 6.18, la sous-tâche τ_2^1 ne peut pas être rattrapée, car à chaque instant où une sous-tâche creuse est ordonnancée, une sous-tâche de τ_2 l'est aussi et donc il y a concurrence d'exécution. C'est ce qui se passe aux instants $t = 17$ et $t = 18$.

III.5. Expérimentation et Preuve

Tout comme dans les autres techniques, les résultats expérimentaux de la technique *TFA* sont concluants. Sur les 550 systèmes testés avec un délai de détection $x = 2$ unités de temps et des instants de panne aléatoires, on a enregistré un taux de validité de plus de 80%. Dans les séquences d'ordonnancement valides, l'équité n'est pas toujours observée (c-à-d que toutes les sous-tâches ne sont pas ordonnancés avant leurs pseudo-échéances), ce qui ne représente pas

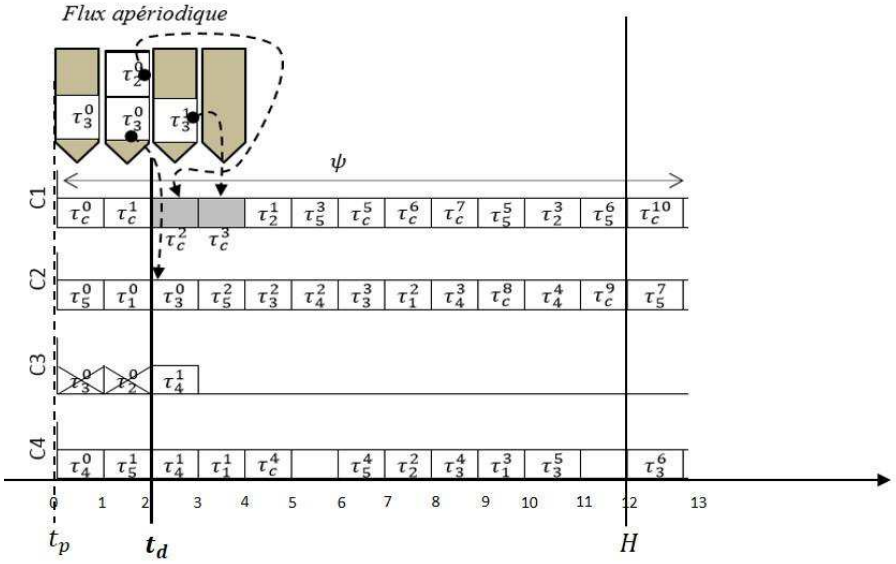

FIGURE 6.17 – Exemple d'ordonnancement par la Technique *TFA* avec concurrence d'exécution

un problème en soi, car l'objectif final reste le respect des échéances des tâches, c'est-à-dire la validité. On a aussi constaté que toutes les instances qui n'ont pas perdu de sous-tâches respectent leurs échéances, qu'elles appartiennent ou pas à des tâches impactées par la panne, ce qui est conforme à nos spécifications. Les dépassements d'échéances s'observent uniquement sur les instances des tâches pour lesquelles les sous-tâches perdues ont été en situation de concurrence d'exécution. Cependant, la présence des concurrences d'exécution n'entraine pas forcément des dépassements d'échéances (voir l'exemple du système Ψ ci-dessus). Nous avons aussi pu constater que la récupération des sous-tâches perdues se fait dans un temps borné. Ces résultats expérimentaux valident cette technique.

En utilisant le raisonnement du chapitre précédent dans lequel nous avons montré que sans la récupération des sous-tâches perdues, les autres sous-tâches s'exécutent dans leurs fenêtres d'exécution, nous prouvons que les instances non affectées par la panne respectent leurs échéances.
Pour ce qui est des instances affectées, des dépassements d'échéances ont été autorisés. Il serait donc absurde de tenter de démontrer le respect des échéances. Cependant, nous pouvons borner le délai de récupération des sous-tâches perdues et donc déterminer l'instant maximal après lequel il est garanti que toutes ces sous-tâches ont été réordonnancées, et le système a donc été basculé à nouveau en mode normal.

Soit β le délai maximal de récupération. L'instant de récupération le plus tardif est donc $t_d + \beta$. La valeur de β est déterminée par encadrement du nombre minimal de temps creux

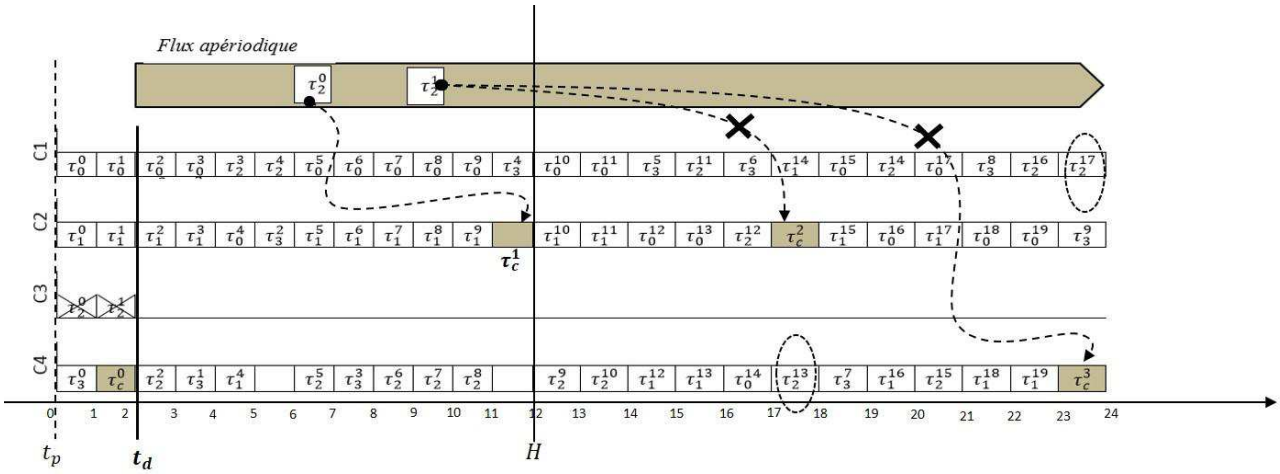

FIGURE 6.18 – Non parallélisme total : mise en évidence de l'effet domino

identifiés entre les instants t_d et $t_d + \beta$.
Pour cela, nous nous servons de l'encadrement de $\lfloor y \rfloor$ et $\lceil y \rceil$ pour tout nombre réel y.
En effet, nous avons $y - 1 < \lfloor y \rfloor \leq y$.
En posant $y = u_c \times t'$ nous avons $u_c \times t' - 1 < \lfloor u_c \times t' \rfloor \leq u_c \times t'$ (a).
Nous savons aussi que $y < \lceil y \rceil \leq y + 1 \Leftrightarrow -y - 1 \leq -\lceil y \rceil < -y$ et donc,
avec $y = u_c \times t$ nous avons $-u_c \times t - 1 \leq -\lceil u_c \times t \rceil < -u_c \times t$ (b).
En additionnant les inégalités (a) et (b), nous obtenons l'inégalité :
$u_c \times (t' - t) - 2 < \lfloor u_c \times t' \rfloor - \lceil u_c \times t \rceil \leq u_c \times (t' - t)$.
Ce qui correspond en utilisant (6.4) à :

$$u_c \times (t' - t) - 2 < W_{min}(t, t') \leq u_c \times (t' - t) \tag{6.6}$$

Considérons d'abord un scénario *optimiste* dans lequel aucune concurrence d'exécution ne se produit. Dans un tel scénario, l'exécution complète des sous-tâches perdues est garantie s'il y a au moins x unités de temps creux identifiées entre l'instant t_d et l'instant $t_d + \beta$. Cela se traduit par :

$$W_{min}(t_d, t_d + \beta) \geq x \tag{6.7}$$

D'après le premier membre de l'inégalité (5.6) nous avons :
$u_c \times (t_d + \beta - t_d) - 2 < W_{min}(t_d, t_d + \beta)$. L'equation (5.4) est donc vérifiée si
$u_c \times (t_d + \beta - t_d) - 2 \geq x \Leftrightarrow u_c \times \beta \geq x + 2 \Leftrightarrow \beta \geq \dfrac{x+2}{u_c}$
D'où on peut considérer comme borne maximale de récupération (*optimiste*).

$$\beta = \left\lceil \frac{x+2}{u_c} \right\rceil \text{ u.t.} \tag{6.8}$$

Revenons à présent au scénario *pessimiste* avec concurrence d'exécution. Puisque la technique *TFA* est basée sur le non-parallélisme partiel, les concurrences d'exécution entre des sous-tâches d'une même instance prennent fin lorsqu'une nouvelle instance de la tâche est activée, c'est-à-dire au plus tard à la prochaine période de cette tâche. Étant donné que les sous-tâches perdues peuvent appartenir à des tâches différentes, toutes les concurrences d'exécution possibles disparaitront à la prochaine période de la tâche de plus grande période. La borne maximale de récupération dans ce scénario est calculée par la formule

$$\beta^{'} = \beta + max(T_i) \text{ u.t} \tag{6.9}$$

Dans l'hypothèse où les x_i sous-tâches perdues d'une tâche τ_i appartiennent à des instances différentes, seules les sous-tâches de la dernière instance activée avant la détection de la panne peuvent être en situation de concurrence d'exécution (à cause du non-parallélisme partiel). La borne maximale (*pessimiste*) ci-dessus reste valide dans ce cas.

Les résultats expérimentaux ont confirmé qu'en l'absence de concurrence d'exécution, la borne optimiste β est toujours respectée. Toutefois, cette borne peut être dépassée en présence de compétition d'exécution sans nécessairement causer la non validité de la séquence d'ordonnancement. La borne $\beta^{'}$ est toujours respectée.
Dans l'exemple du système Ψ nous avons $\beta = \left\lceil \frac{2+2}{10/12} \right\rceil = 5$ et $\beta^{'} = 5 + 12 = 17$. Les bornes de récupération sont respectées car dans les deux scénarii la durée de récupération est de 2 unités de temps.

IV. Comparaison entre les techniques

Afin d'opérer un choix entre les trois techniques présentées dans les sections précédentes, une étude comparative est nécessaire. Dans un premier temps, cette étude est menée sur l'exemple du système Ψ. Ensuite, elle est généralisée.

IV.1. Étude d'un exemple

Pour comparer les techniques, plusieurs indicateurs peuvent être retenus : le nombre de cœurs additionnels μ, la charge de traitement induite, la durée de la période de récupération et le résultat d'ordonnancement. Le Tableau 6.3 présente les valeurs de ces indicateurs obtenues dans les exemples précédents pour l'ordonnancement du système Ψ par les trois techniques avec $t_d = 3$ et $x = 2$.

A la lecture de ce tableau, on observe que l'ordonnancement du système Ψ avec la technique TSS nécessite plus de cœurs qu'avec les deux autres techniques et induit une charge à traiter

TABLE 6.3 – Comparaison des trois techniques avec le système ψ

	Charge à traiter	μ	S/tâches perdues	récupération	Résultat
TSS	4.0	2	τ_5^1, τ_5^2	8 u.t	valide et équitable
TCR	$3.92 - 2.41 - 2.16$	1	τ_3^1, τ_5^2	8 u.t	valide et équitable
TFA	2.16	1	τ_2^0, τ_4^1	2 u.t	valide non équitable

plus importante. Les techniques TCR et TFA utilisent un seul cœur de plus que nécessaire, mais TFA induit une charge à traiter moindre. La séquence d'ordonnancement du système Ψ produite avec les trois techniques est valide, mais l'équité n'est pas garantie avec TFA. Cependant le système récupère plus vite les sous-tâches perdues avec TFA qu'avec TCR et plus vite avec TCR qu'avec TSS. Toutefois, il faut souligner qu'une comparaison des techniques basée sur le délai de récupération n'est pas objective. En effet, dans les techniques TSS et TCR, ce délai dépend des paramètres temporels des tâches impactées, plus spécifiquement du WCET et de la période. Or, bien qu'on ait considéré le même instant de détection de panne et le même délai de détection dans TSS et TCR, les tâches impactées ne sont pas les mêmes. En général, une tâche ayant une petite période récupèrera plus vite qu'une tâche de plus grande période dans ces deux techniques. Dans la technique TFA par contre, la durée de récupération dépend d'une part du nombre d'unités de temps creux identifiées après la détection de la panne (et donc des paramètres de la tâche creuse utilisée dans la répartition équitable), et d'autre part, du nombre de concurrences d'exécution. Toutefois, il est indéniable que la répartition équitable des unités de temps creux augmente les chances de disposer très tôt d'unités de temps creux après la détection de la panne et donc de récupérer rapidement.

IV.2. Caractéristiques des différentes techniques

Le Tableau 6.4 résume les principales caractéristiques de chaque technique. Comme nous pouvons le constater, la technique TSS est sûre, simple d'utilisation et peut s'appliquer à tout système qui satisfait la condition (5.1). Le système s'exécute dans le même mode avant et après la détection de la panne et donc, les sous-tâches gardent les mêmes fenêtres d'exécution au cours de l'ordonnancement, ce qui évite le problème d'inversion de priorité. Cependant, le système initial est surchargé, ce qui nécessite plus de cœurs additionnels. *TCR* est une technique plus flexible qui, tout comme *TSS*, assure la validité et l'équité de la séquence d'ordonnancement avec un seul cœur additionnel et moins de charge à traiter. Cependant, elle s'applique à une plage plus réduite de systèmes ordonnançables parce qu'elle impose des conditions supplémentaires. TFA est une technique flexible qui peut s'appliquer à tout système ordonnançable. Sa limite est que la validité de la séquence d'ordonnancement n'est pas garantie, en particulier en présence de concurrences d'exécution. Toutefois, le nombre de fautes temporelles est borné, de même que la durée de récupération.

Il ressort de cette comparaison que chaque technique a des avantages et des inconvénients et donc, le choix d'une technique au détriment des autres est laissé au concepteur qui le fera en fonction du type du système à ordonnancer, des caractéristiques de la plateforme matérielle d'exécution qu'il possède et de ses objectifs opérationnels.

TABLE 6.4 – Caractéristiques de chaque technique de tolérance

	Sous-tâches de Substitution	**Contraindre-Relâcher**	**Flux Apériodique**
Charge supplémentaire	élevée et statique	décroissante	Aucune
Cœurs additionnels	peut dépasser un	un seul	un seul
Paramètres temporels	modifiés	modifiés	préservés
Priorités entre les tâches	préservées	modifiées	préservées
Modes d'exécution	un seul	trois	deux
Comportement	rigide	flexible	flexible
Applicabilité	tout système respectant (5.1)	pas tous les systèmes ordonnançables	tout système ordonnançable
Validité	garantie à tout instant	garantie à tout instant	garantie avant l'instant de la panne et après l'instant de fin de récupération

IV.3. Utilisation conjointe des techniques

Nous avons vu dans le chapitre précédent que la technique TRML est utilisable si on peut accepter que quelques instances de tâches ne s'exécutent pas intégralement. Par ailleurs, dans les sections précédentes de ce chapitre, nous avons vu que la technique TCR s'utilise quand il est impératif que toutes les instances s'exécutent intégralement avant leurs échéances alors que TFA s'utilise s'il faut que les instances terminent leurs exécutions, mais qu'un dépassement ponctuel d'échéances peut être acceptable. Un système peut cependant comporter des tâches de chacune de ces trois natures. Dans ce cas, il peut être intéressant de combiner les trois méthodes, le traitement appliqué à chaque sous-tâche perdue dépendant de la nature de la tâche. On définit ainsi une nouvelle technique d'ordonnancement avec panne que nous appellerons *Technique Combinée*. Pour cela, on considère trois natures de tâches : *arretable (A)*, *retardable (R)* et *stricte (S)*. Lors de la détection de la panne, si une tâche de la nature A est impactée, l'exécution de l'instance courante peut être stoppée sans altérer considérablement la qualité du résultat. Par contre si une tâche de l'une des natures R et S est impactée, son instance

courante doit absolument terminer son exécution. A la différence de la nature S où l'exécution doit impérativement se terminer avant l'échéance, l'instance courante d'une tâche de nature R peut poursuivre son exécution même après son échéance. Le principe de la technique combinée est donc le suivant :

Lorsqu'une panne est détectée :
- si la tâche impactée est une tâche arretable, ses sous-tâches perdues ne sont pas récupérées (ordonnancement sans reprise avec TRML) ;
- si la tâche impactée est une tâche retardable, les sous-tâches perdues sont récupérées dans les unités de temps creux (ordonnancement avec TFA : le non respect de l'échéance d'une tâche retardable n'étant pas préjudiciable) ;
- si la tâche impactée est une tâche stricte, les sous-tâches perdues sont récupérées avant la prochaine échéance de la tâche (ordonnancement avec reprise par TCR).

Au début de l'ordonnancement, seules les tâches strictes ont des échéances contraintes et une tâche creuse (arretable) est ajoutée pour répartir équitablement les unités de temps creux. Les autres tâches (retardables et arretables) gardent des échéances sur requêtes. Si U_R et U_A désignent les charges respectives induites par les tâches retardables et arrêtables, $U_{S_{max}}$ la charge maximale induite par les tâches strictes (c'est-à-dire la charge obtenue en ajoutant x sous-tâches à la tâche stricte de plus petite période), la tâche creuse a pour WCET $IT = (m - (U_{S_{max}} + U_R + U_A)) \times H$ et pour période, l'hyperpériode H du système.

A titre d'illustration, prenons à nouveau le système
$\Psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$. $U = 2.16$ (et donc $m = 3$).
Le partitionnement de l'ensemble des tâches est donné dans le Tableau 7.4. Le système à ordonnancer est donc le suivant :
$\psi' : \tau'_1 < 1, 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau'_4 < 5, 9, 12 >, \tau_5 < 7, 12 >$. Son facteur de charge est $CH_{\psi'} = \sum_{i=1}^{n}(C_i/D'_i) = 3.80 < 4 = m + 1$.

TABLE 6.5 – Partitionnement de l'ensemble des tâches du système Ψ

	Arretables	Retardables	Strictes
	τ_2, τ_3	τ_5	τ_1, τ_4
charge (U)	$\frac{10}{12}$	$\frac{7}{12}$	$\frac{9}{12}$

On considère dans un premier temps une panne affectant le cœur $C2$ à l'instant 1. Le délai de détection est de $x = 2$ unités de temps. $U_{S_{max}} = U_S + \frac{x}{min(T_1,T_4)} = \frac{9}{12} + \frac{2}{3} = \frac{17}{12}$. La tâche creuse a pour WCET $(3 - (\frac{10}{12} + \frac{7}{12} + \frac{17}{12})) \times 12 = 2$ et donc pour paramètres $\tau_c < 2, 12 >$. La Figure 6.19 montre la séquence d'ordonnancement obtenue par l'application de la technique combinée. Les tâches τ_2 et τ_c sont impactées. Puisqu'il s'agit de tâches arretables, les sous-tâches perdues τ_c^1 (qui n'est rien d'autre qu'une unité de temps creux) et τ_2^0 ne sont pas reprises.

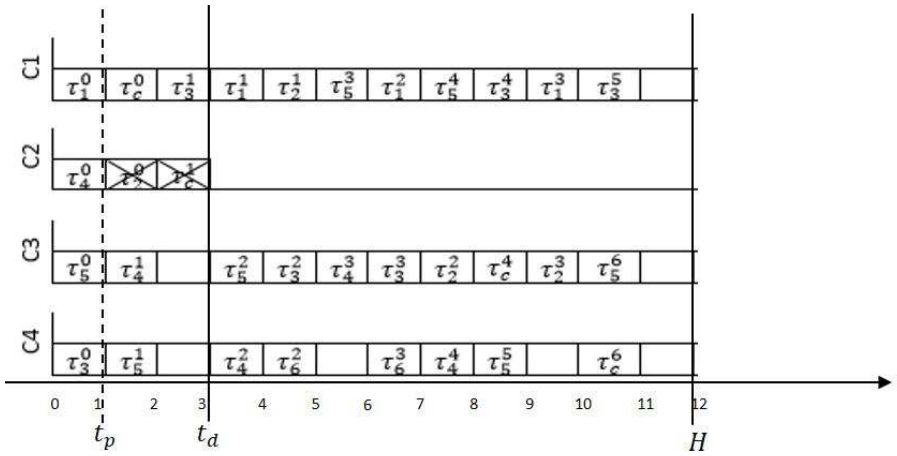

FIGURE 6.19 – Ordonnancement de Ψ par la technique combinée : tâches arretables impactées

Considérons à présent une panne sur le cœur $C3$ à l'instant 0. Les tâches impactées sont τ_4 et τ_5. Puisque τ_4 est de nature stricte, la sous-tâche perdue τ_4^1 est remplacée par la sous-tâche de substitution $s_4^{1,1}$, ce avant la prochaine échéance. Quant à la tâche retardable τ_5, sa sous-tâche τ_5^0 est reprise dans l'unité de temps creux τ_c^1. La séquence d'ordonnancement est présentée dans la Figure 6.20.

Comme on peut le constater les séquences d'ordonnancement des Figures 6.19 et 6.20 sont valides et équitables. L'expérimentation de la technique combinée avec le prototype FTA nécessite, pour chaque système généré, de définir un partitionnement des tâches par nature. Or comme nous l'avons vu, ce partitionnement est une activité de conception et donc se fait avant l'exécution du système.

Conclusion

Lorsque du temps supplémentaire doit être alloué pour réordonnancer les sous-tâches perdues, l'ordonnancement PD2 d'un système avec éventualité de défaillance d'un cœur peut s'effectuer grâce à l'une des trois techniques présentées dans ce chapitre. Ces techniques sont basées sur la redondance matérielle limitée, la modification des paramètres temporels du système, les changements de mode d'exécution et l'exploitation des temps creux. La première technique TSS augmente le WCET des tâches pour prévoir des sous-tâches additionnelles qui pourront remplacer les sous-tâches perdues. Elle peut s'appliquer à tout système ordonnançable respectant la condition (5.1) et garantit la validité et l'équité de la séquence d'ordonnancement. C'est une solution sûre qui peut être choisie s'il y a suffisamment de cœurs disponibles sur la plateforme d'exécution. La seconde technique TCR est appropriée lorsque le nombre de cœurs disponibles sur la plateforme d'exécution est limité. Elle réduit le délai critique des tâches pour prévoir une

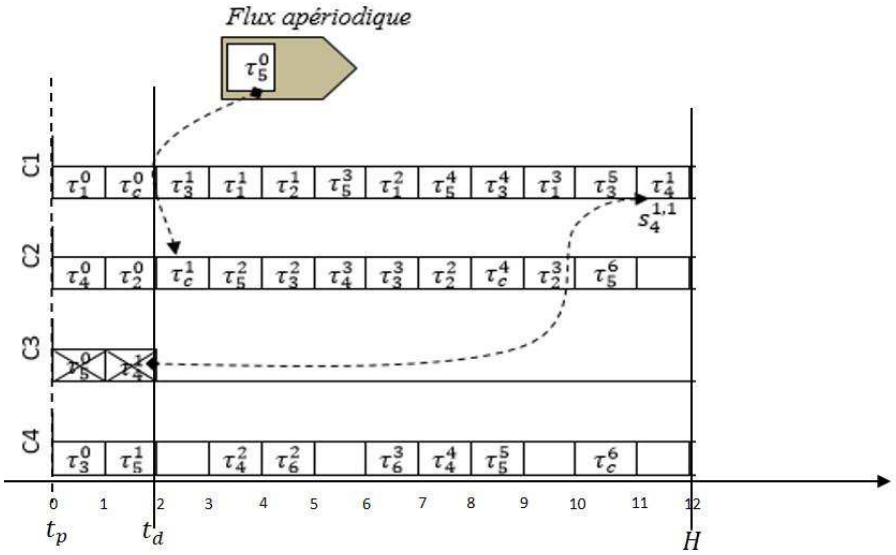

FIGURE 6.20 – Ordonnancement de Ψ par la technique combinée : tâches retardables et strictes impactées

marge de temps qui pourra être utilisée pour créer des fenêtres d'exécution supplémentaires à affecter au réordonnancement des sous-tâches perdues. C'est une technique flexible qui garantit la validité et l'équité de la séquence d'ordonnancement avec une charge supplémentaire réduite et un seul cœur additionnel. Elle peut être préférée si le système satisfait aux conditions d'application. La dernière technique TFA procède dans un premier temps à une répartition équitable des unités de temps creux qu'elle utilise ensuite pour réordonnancer les sous-tâches perdues, dans le respect de la règle du non-parallélisme partiel. C'est une technique flexible qui s'applique à tout système ordonnançable. Les concepteurs peuvent la préférer aux autres techniques si le respect des échéances des tâches n'est pas une contrainte rigide, c'est-à-dire, si un nombre borné de fautes temporelles est acceptable. Enfin, les trois techniques peuvent être conjuguées afin d'adopter pour chaque tâche le comportement le plus adapté à sa nature.

Le prochain chapitre traitera de la défaillance de plusieurs cœurs. Dans ce cas, même en utilisant de la redondance matérielle, au bout d'un certain nombre de pannes, le nombre de cœurs fonctionnels restants ne garantit plus l'ordonnançabilité du système et la tolérance nécessite de nouvelles approches.

Chapitre

7 Possibilités d'Extension

Sommaire

Résumé

Ce chapitre aborde les possibilités d'extension des techniques de tolérance présentées dans les chapitres précédents aux cas de plusieurs pannes. Il s'inspire des approches classiques de tolérance qui sont préalablement introduites. On considère à cet effet qu'au cours de l'ordonnancement des tâches du système, plusieurs cœurs du processeur tombent simultanément ou successivement en panne.

Introduction

Dans les chapitres précédents, nous avons proposé des techniques de tolérance d'un système temps-réel à une panne détectée après un délai de x unités de temps ($x \geq 1$). Il a alors été supposé qu'au cours de l'ordonnancement du système par un algorithme équitable, un seul cœur du processeur tombe en panne. Dans la suite, on parlera dans ce cas *d'ordonnancement mono-panne*. Lorsque l'ordonnancement du système s'effectue sans reprise des sous-tâches perdues, la Technique de la Redondance Matérielle Limitée (TRML) est utilisée. En cas d'ordonnancement avec reprise, la Technique des Sous-tâches de Substitution (TSS), la Technique Contraindre puis Relâcher (TCR) et la Technique du Flux Apériodique (TFA) peuvent être employées de manière individuelle ou combinée.

Dans ce chapitre, nous traitons le cas où plusieurs cœurs du processeur tombent en panne : on parle alors *d'ordonnancement multi-panne*. Les pannes peuvent survenir simultanément ou successivement : à l'instant t_{p_1}, λ_1 cœurs tombent en panne ; à l'instant t_{p_2}, λ_2 cœurs tombent en panne et ainsi de suite, avec $\lambda_k \geq 1$. A un moment donné, le nombre de cœurs restants peut finir par être inférieur à la charge du système et de ce fait, la condition nécessaire et suffisante d'ordonnançabilité (équation (1.2)) ne sera plus vérifiée. Dès lors, une réduction de la charge du système s'impose. Nous étudions ici comment dans un tel contexte les techniques mono-pannes peuvent être étendues à l'ordonnancement multi-panne. Dans la première section, nous présentons deux approches classiques d'ordonnancement multi-pannes : l'approche par changement de mode de criticité et l'approche par ordonnancement basée sur les valeurs ("*value based scheduling*"). Une présentation détaillée de ces approches est fournie dans [131] pour la première et [62] pour la deuxième. En nous inspirant de ces approches classiques, nous évoquons dans la Section II les possibilités d'extension de nos techniques mono-pannes à l'ordonnancement multi-panne.

I. Approches classiques d'ordonnancement multi-pannes

I.1. Approche par changement de mode de criticité

Nous avons vu au Chapitre 3 qu'un système temps-réel peut être conçu avec plusieurs niveaux de criticité : on parle de système à criticité multiple (MCS : Mixed Criticality System). Rappelons que la criticité désigne le niveau d'assurance contre les pannes nécessaire pour un composant d'un système. En général, dans les MCS, certains paramètres des tâches à l'instar du WCET sont des fonctions du niveau de criticité du système : la tâche possède plusieurs WCET, le WCET choisi dépend du niveau de criticité. Différents modes d'exécution correspondant aux différents niveaux de criticité du système sont définis. Au cours de son exécution, des changements de mode de criticité permettent d'adapter le système à l'évolution de son environnement. La définition des niveaux de criticité a un double avantage. Premièrement, elle permet de mettre en place un mécanisme rapide et efficace de détection de pannes. En effet, l'éventualité d'une panne peut être soulevée dès lors qu'on constate qu'un travail épuise sa pire durée d'exécution sans terminer son exécution. Ceci suppose qu'on dispose hors ligne d'informations indiquant la fin d'exécution de chaque tâche. De tels mécanismes de détection sont courants dans les systèmes à sécurité critique (*safety-critical systems*) qui requièrent un certain niveau de tolérance aux pannes. Deuxièmement, en cas d'insuffisance de ressource (nombre de cœurs fonctionnels insuffisants par exemple), le système peut passer à un mode de criticité supérieur où la charge est réduite du fait de la suppression de certaines tâches ou travaux. Il s'en suit une dégradation progressive du système jusqu'au niveau de criticité maximal dans lequel toute suppression de tâches ou de travaux s'avère fatale.

Dans la littérature, peu de travaux traitant directement de la tolérance aux pannes des MCS ont été publiés. On peut néanmoins citer les travaux de Huang et al. [170] qui mettent l'accent sur l'adaptation des services et la tolérance aux pannes des MCS. On peut aussi citer les travaux de Lin et al. [177] et ceux de Pathan [178] qui abordent la tolérance aux pannes des MCS par l'exécution des copies primaires et des copies de secours des tâches. Tout comme Thekkilakattil et al [179], tous ces travaux traitent des pannes transitoires en architecture mono-processeur. Dans la section suivante, nous verrons comment utiliser le changement de mode de criticité pour la tolérance aux pannes permanentes en architecture multicoeur.

I.2. Approche par ordonnancement basée sur les valeurs

Dans l'optique de réaliser une dégradation progressive en présence d'une surcharge d'un système temps-réel dynamique, l'ordonnancement basé sur les valeurs a été proposé par plusieurs travaux [180, 106, 181, 182, 183, 184]. Il s'agit d'un paradigme d'ordonnancement qui ordonnance les tâches de telle sorte que la valeur globale (encore appelée utilité globale) du système est maximisée [180]. Ce paradigme se fonde sur l'hypothèse que chaque tâche du système lui procure une certaine « valeur » (ou récompense) si elle respecte son échéance et dans le cas

contraire (échéance manquée), elle encoure une pénalité. La « valeur » représente l'importance de la tâche alors que son délai critique représente son urgence en terme de délai d'exécution. Cependant, le processus d'attribution des valeurs à chaque tâche dépend de l'application et est une opération complexe [185]. Des exemples de fonction de valeur sont proposées dans [186] et [182]. Les heuristiques d'ordonnancement basées sur les valeurs attribuent la priorité aux tâches en combinant la valeur de la tâche avec d'autres paramètres statiques de la tâche tels que son délai critique et son WCET. Plusieurs algorithmes qui combinent valeur de la tâche et délai critique sont donnés dans [187, 184]. Lorsqu'une tâche est prête, l'ordonnanceur décide s'il doit ou non l'accepter dans la séquence. Une tâche est acceptée si son ajout à la séquence ne compromet pas la capacité des tâches déjà acceptées à respecter leurs échéances. Dans le cas contraire, elle est rejetée. Dans les algorithmes de [187, 184], l'ordonnanceur ajoute premièrement une tâche nouvellement réveillée à la séquence en fonction de son délai critique sans aucune considération pour sa valeur. Dans le cas où cette nouvelle tâche n'est pas acceptée, l'ordonnanceur essaye de supprimer de la séquence toutes les tâches de valeur inférieure déjà acceptées. Dans le cas où la séquence n'est toujours pas valide, la nouvelle tâche est rejetée et les anciennes tâches acceptées retrouvent leurs anciennes positions dans la séquence. Dans [180], les auteurs proposent un ordonnanceur basé sur la valeur de fiabilité pour les systèmes temps-réel multiprocesseurs dynamiques dont l'objectif principal est de maximiser l'indice de performance du système. L'indice de performance capture les paramètres de récompense ou de pénalité (selon que la tâche respecte ou non son échéance) et arbitre entre ordonnançabilité et fiabilité. Les tâches sont supposées apériodiques, indépendantes et non-préemptibles. L'ordonnancement de multiples versions d'une tâche sur différents processeurs assure la tolérance. Les auteurs ont travaillé en recherchant le niveau de redondance approprié pour chaque tâche (en utilisant des heuristiques) qui maximise l'indice de performance du système. Cependant, aucune étude de cas d'applications temps-réel n'est fournie. Dans [106], l'auteur a proposé, pour les systèmes multiprocesseurs, des algorithmes d'ordonnancement basés sur les valeurs tolérant aux pannes. Les tâches sont supposées indépendantes, apériodiques et non préemptives. Le schéma de priorité utilise une combinaison du temps d'exécution, de la valeur et du délai critique. Pour la tolérance aux pannes, la technique primary-backup est utilisée. La performance est comparée avec l'algorithme EDF et il en ressort que, pour les tâches d'égale valeur, EDF est meilleur, alors que pour les tâches de valeurs différentes, le schéma de priorité proposé est meilleur en terme de somme de valeurs, mais avec un taux de rejet légèrement plus élevé que EDF. Comme nous pouvons le constater, ces travaux reposent sur un modèle de tâches différent du notre. Toutefois, dans le deuxième paragraphe de la prochaine section, nous nous inspirons de l'ordonnancement basé sur les valeurs pour définir une approche de tolérance aux pannes des systèmes reposant sur notre modèle de tâches.

II. Possibilités d'extension des techniques mono-pannes

Dans cette section, nous étudions comment étendre les techniques TRML, TSS, TCR et TFA à la tolérance à plusieurs pannes en nous inspirant des approches classiques présentées à la section précédente. Nous nous situons dans le contexte où le nombre de cœurs fonctionnels ne garantit plus la vérification de la condition nécessaire et suffisante d'ordonnançabilité du système. L'objectif global est donc d'alléger le système tout en préservant le plus grand nombre de fonctionnalités et/ou en maximisant la charge restante. Les approches proposées n'ayant pas fait l'objet de travaux approfondis, nous donnons leurs idées générales et les illustrons sur un exemple. Mais avant, il faut savoir si toutes les techniques mono-pannes sont extensibles à l'ordonnancement multi-panne.

II.1. Étude des techniques mono-pannes

L'objectif étant de réduire la charge du système, la technique TSS s'avère inappropriée du fait qu'elle prévoit d'avance des sous-tâches additionnelles (et donc une augmentation de la charge du système) qui sont ordonnancées même après la phase de récupération. De même, en débutant l'ordonnancement avec des échéances contraintes, la technique TCR augmente la charge du système initial. De plus, elle soulève le problème d'inversion de priorité entre les sous-tâches, ce qui complique la preuve. Contrairement aux techniques TSS et TCR, la technique TFA conserve la charge initiale du système, les sous-tâches perdues étant récupérées dans les unités de temps creux. Bien que la technique TFA autorise un nombre limité de dépassements d'échéances, la charge qu'elle induit est minimale par rapport aux autres technique et donc son extension à l'ordonnancement multi-panne est plus efficace. C'est pourquoi, dans un scénario avec reprise des sous-tâches perdues, les approches d'ordonnancement proposées dans cette section reposent sur la technique TFA. Toutefois, pour ordonnancer les sous-tâches du flux apériodique, nous exploiterons aussi bien les unités de temps creux identifiées par les sous-tâches creuses que celles qui surviennent lorsque le nombre de sous-tâches prêtes est inférieur au nombre de cœurs fonctionnels. Nous qualifions ces dernières d'unités de temps creux par épuisement.

II.2. Extension par changement de mode de criticité

II.2.a. Présentation de l'approche

Dans cette approche, plusieurs niveaux de criticité sont définis en fonction du nombre de cœurs fonctionnels. Par exemple, on définit les niveaux de criticité $L_{m+1}, L_m, L_{m-1}, .., L_{m_0}$ correspondant respectivement à : $m+1$ cœurs, m cœurs, .., m_0 cœurs fonctionnels. m est le nombre minimum de cœurs nécessaire pour garantir l'ordonnançabilité du système et m_0 ($m_0 < m$) le nombre de cœurs requis pour son fonctionnement le plus dégradé acceptable. Le système est alors conçu avec plusieurs modes de criticité $M(L_m), M(L_{m-1}), .., M(L_{m_0})$ et donc certains

paramètres des tâches sont définis en fonction du niveau de criticité de manière à réduire progressivement la charge globale du système lors d'un changement de mode de criticité. A cet effet, deux possibilités sont offertes : réduire la durée d'exécution des tâches et/ou réduire le nombre d'instance. Dans le premier cas, le WCET C_i d'une tâche τ_i est une fonction décroissante du niveau de criticité L de telle sorte $C_i(L_m) > C_i(L_{m-1}) > ... > C_i(L_{m_0})$. Dans le deuxième cas, c'est la période T_i qui est une fonction croissante du niveau de criticité et on a $T_i(L_m) < T_i(L_{m-1}) < ... < T_i(L_{m_0})$. Rappelons que déterminer les valeurs de ces paramètres dépend exclusivement des concepteurs qui seuls peuvent savoir la durée d'exécution ou la période qu'on peut attribuer à une tâche sans altérer considérablement la qualité de service du système. Avec une telle conception, la charge U du système est une fonction décroissante du niveau de criticité telle que $U(L_m) < m$, $U(L_{m-1}) < m-1$,...,$U(L_{m_0}) < m_0$. Dans chaque mode de criticité $M(L_\delta)$, cette charge est définie de manière à ce qu'il y ait des unités de temps creux disponibles pour réordonnancer les sous-tâches perdues (i.e $IT(L_\delta) = H \times (\delta - U(L_\delta)) > 0$). La tâche creuse a donc pour WCET $C_c(L_\delta) = IT(L_\delta)$.

Le système débute son exécution dans le mode $M(L_{m+1})$ et l'ordonnancement se fait soit suivant la technique TRML (scénario sans reprise), soit suivant la technique TFA (scénario avec reprise), ce qui signifie que l'on utilise les paramètres initiaux des tâches. Lorsque la première défaillance d'un cœur est enregistrée, les paramètres temporels des tâches restent inchangés. Le système bascule dans le mode $M(L_m)$ qui est identique à $M(L_{m+1})$. Durant la suite de l'exécution, l'occurrence d'une nouvelle défaillance entrainera le passage du mode de criticité courant au mode de criticité $M(L_{m+1-\omega})$, où ω est le nombre de cœurs défaillants. Il appartient aux concepteurs de décider si les travaux inachevés du mode précédent sont autorisés ou pas à poursuivre leur exécution dans le nouveau mode. Nous proposons le protocole suivant lors du passage d'un mode de criticité $M(L_\delta)$ à un mode $M(L_{\delta-1})$. $C_i(L_\delta)$ et $C_i(L_{\delta-1})$ désignent respectivement les WCET d'une tâche τ_i dans les deux modes. La Figure 7.1 présente le niveau d'exécution de l'instance courante d'une tâche à l'instant t_M du changement de mode. Les sous-tâches *restantes* sont celles qui restent à ordonnancer pour atteindre un niveau d'exécution de $C_i(L_{\delta-1})$ unités de temps requis dans le mode $M(L_{\delta-1})$. Les sous-tâches en *surplus* sont celles qu'il faut ordonnancer en plus des sous-tâches restantes pour atteindre le niveau d'exécution de $C_i(L_\delta)$ unités de temps requis dans le mode $M(L_\delta)$. Le protocole de passage du mode $M(L_\delta)$ au mode $M(L_{\delta-1})$ est le suivant :

- Pour l'instance courante de la tâche creuse τ_c : ordonnancer les sous-tâches restantes dans le nouveau mode $M(L_{\delta-1})$ et abandonner les sous-tâches en surplus. Les prochaines instances de τ_c auront pour WCET $C_c(L_{\delta-1})$;
- Pour les instances courantes des autres tâches : ordonnancer les sous-tâches restantes dans le nouveau mode $M(L_{\delta-1})$. Si l'ordonnancement est sans reprise des sous-tâches perdues, abandonner les sous-tâches en surplus. Sinon, mettre les sous-tâches restantes dans le flux afin de les ordonnancer dans les prochaines unités de temps creux. La prochaine instance d'une tâche τ_i aura pour WCET $C_i(L_{\delta-1})$.

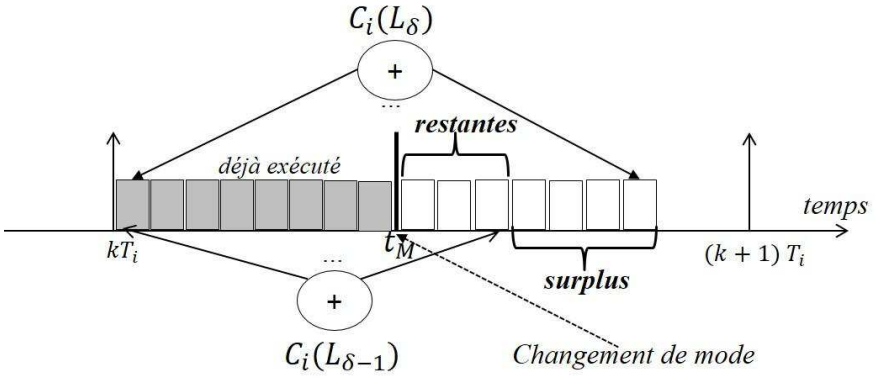

FIGURE 7.1 – Niveau d'exécution d'une tâche lors du changement de mode de criticité

II.2.b. Exemple d'ordonnancement

Considérons à nouveau le système
$\Psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$.
Rappelons que $U = 2.16$ (et donc $m = 3$ cœurs), l'hyperpériode est $H = ppcm(3, 4, 6, 12) = 12$ et le nombre d'unités de temps creux $IT = 10$. En application de la technique de la redondance matérielle limitée, l'ordonnancement débutera sur 4 cœurs. Quatre niveaux de criticité sont considérés : le niveau L_4 qui correspond à plus de 4 cœurs fonctionnels et qui est équivalent au niveau L_3 correspondant à 3 cœurs fonctionnels, le niveau L_2 qui correspond à 2 cœurs fonctionnels et le niveau L_1 qui correspond à 1 cœur fonctionnel. Le système est donc conçu suivant les trois modes de criticité $M(L_3) = M(L_4)$, $M(L_2)$ et $M(L_1)$. Les travaux inachevés du mode précédent sont autorisés à poursuivre leur exécution dans le nouveau mode jusqu'à l'activation de nouvelles instances des tâches. Les paramètres temporels des tâches sont donnés dans le tableau 7.1. On peut noter que les WCET sont des fonctions décroissantes du niveau de criticité.

TABLE 7.1 – Paramètres des tâches du système Ψ dans différents modes

τ_i	r_i	$C_i(L_3)$	$C_i(L_2)$	$C_i(L_1)$	D_i	T_i
τ_1	0	1	1	0	3	3
τ_2	0	2	1	1	6	6
τ_3	0	2	1	0	4	4
τ_4	0	5	5	5	12	12
τ_5	0	7	5	3	12	12
$U(L) = \frac{C_i(L_m)}{T_i}$		2.16	1.58	0.83	-	-
τ_c	0	10	5	2	12	12

On considère deux pannes successives survenues aux instants $t_{p_1} = 1$ sur le cœur $C2$ et $t_{p_2} = 5$

sur le cœur $C3$. Le délai de détection de la panne est $x = 2$ unités de temps. Le changement de mode de criticité a lieu à l'instant 7 où la deuxième défaillance est détectée. Les états des sous-tâches à l'instant du changement de mode sont présentés dans le tableau 7.2.

TABLE 7.2 – Etat des sous-tâches à l'instant 7 où le système passe de $M(L_3)$ à $M(L_2)$

τ_i	τ_i^j	r_i^j	d_i^j	N° instance	Etat
τ_1	τ_1^0	0	3	1	ordonnancée
	τ_1^1	3	6	2	ordonnancée
	τ_1^2	6	9	3	ordonnancée
	τ_1^3	9	12	4	non activée
τ_2	τ_2^0	0	3	1	ordonnancée
	τ_2^1	3	6	1	ordonnancée
	τ_2^2	6	9	2	restante
	τ_2^3	9	12	2	surplus
τ_3	τ_3^0	0	2	1	ordonnancée
	τ_3^1	2	4	1	ordonnancée
	τ_3^2	4	6	2	ordonnancée
	τ_3^3	6	8	2	ordonnancée
	τ_3^4	8	10	3	non activée
	τ_3^5	10	12	3	non activée
τ_4	τ_4^0	0	3	1	ordonnancée
	τ_4^1	2	5	1	ordonnancée
	τ_4^2	4	8	1	ordonnancée
	τ_4^3	7	10	1	prête
	τ_4^4	9	12	1	non activée
τ_5	τ_5^0	0	2	1	ordonnancée
	τ_5^1	1	4	1	ordonnancée
	τ_5^2	3	6	1	ordonnancée
	τ_5^3	5	7	1	ordonnancée
	τ_5^4	6	9	1	perdue
	τ_5^5	8	11	1	surplus
	τ_5^6	10	12	1	surplus

A l'instant $t_M = 7$, la troisième instance de la tâche τ_1 est achevée et la quatrième instance n'a pas encore été activée. Puisque $C_1(L_2) = C_1(L_3)$, il n'y a pas de sous-tâches restantes, ni de surplus. Pour ce qui est de la tâche τ_2, sa deuxième instance est en cours et la sous-tâche τ_2^2 est prête. Puisque le WCET de cette tâche passe de $C_2(L_3) = 2$ à $C_2(L_2) = 1$, τ_2^2 est une sous-tâche restante dans cette instance et τ_2^3 une sous-tâche en surplus qui est ajoutée au flux. L'instance courante (deuxième instance) de la tâche τ_3 est achevée et la prochaine n'est pas encore activée. Il n' y a donc pas de sous-tâches restantes ni de sous-tâche en surplus. Puisque

le WCET de cette tâche passe de $C_3(L_3) = 2$ à $C_3(L_2) = 1$, dans la troisième instance, seule τ_3^4 sera ordonnancée et τ_3^5 sera abandonnée. La première instance de la tâche τ_4 est en cours. Puisque $C_4(L_2) = C_4(L_3)$, il n'y a pas de sous-tâches restantes, ni de surplus. Le nombre de sous-tâches à ordonnancer dans les instances suivantes reste le même. Pour la tâche τ_5 dont la première instance est en cours, le WCET passe de $C_5(L_3) = 7$ à $C_5(L_2) = 5$. Puisque cinq de ses sous-tâches ont déjà été ordonnancées, il n'y a pas de sous-tâches restantes. τ_5^5 et τ_5^6 sont des sous-tâches en surplus qui sont ajoutées dans le flux. Enfin, le WCET de la tâche creuse τ_c passe de 10 à 5 et six de ses sous-tâches ont déjà été ordonnancées, il n'y a pas de sous-tâches restantes. D'après le protocole décrit ci-dessus, les quatre sous-tâches en surplus τ_c^6, τ_c^7, τ_c^8 et τ_c^9 sont tout simplement abandonnées. Il faudra donc compter sur les temps creux par épuisement ou sur les prochaines instances de la tâche τ_c pour ordonnancer les sous-tâches du flux.
La Figure 7.2 montre la séquence d'ordonnancement du système dans le scénario sans reprise. Comme on peut le constater, les échéances des tâches sont respectées et donc l'ordonnancement est valide.

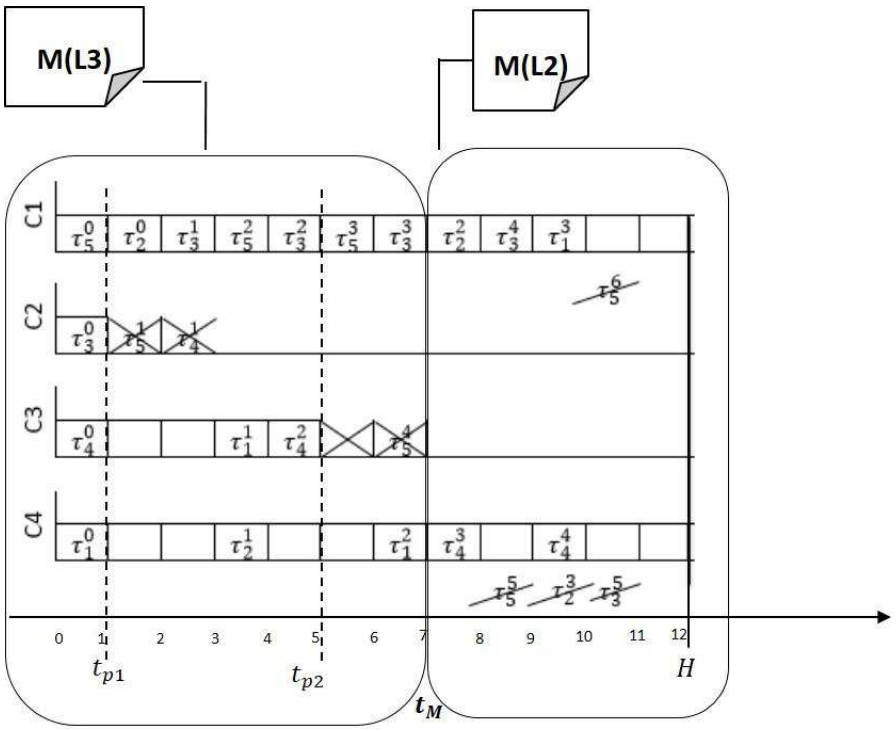

FIGURE 7.2 – Exemple d'ordonnancement multi-panne sans reprise par changement de mode de criticité.

La séquence d'ordonnancement dans le scénario avec reprise est présentée dans la Figure 7.3. Les sous-tâches perdues τ_1^0 et τ_3^1 sont récupérées dans les unités de temps creux τ_c^3 et τ_c^4 tandis que τ_4^2 et τ_3^3 sont récupérées à l'instant 10 dans les unités de temps creux par épuisement. Les sous-tâches en surplus τ_5^5, τ_2^3 qui sont dans le flux sont ordonnancées dans les unités de temps

creux par épuisement restantes alors que τ_5^6 est ordonnancée à l'hyperpériode suivante à la place de τ_c^{10}. Cette séquence n'est pas valide car τ_5 ne respecte pas son échéance, ce qui n'est pas grave car avec la technique TFA, des dépassements d'échéances sont autorisés dans certaines limites.

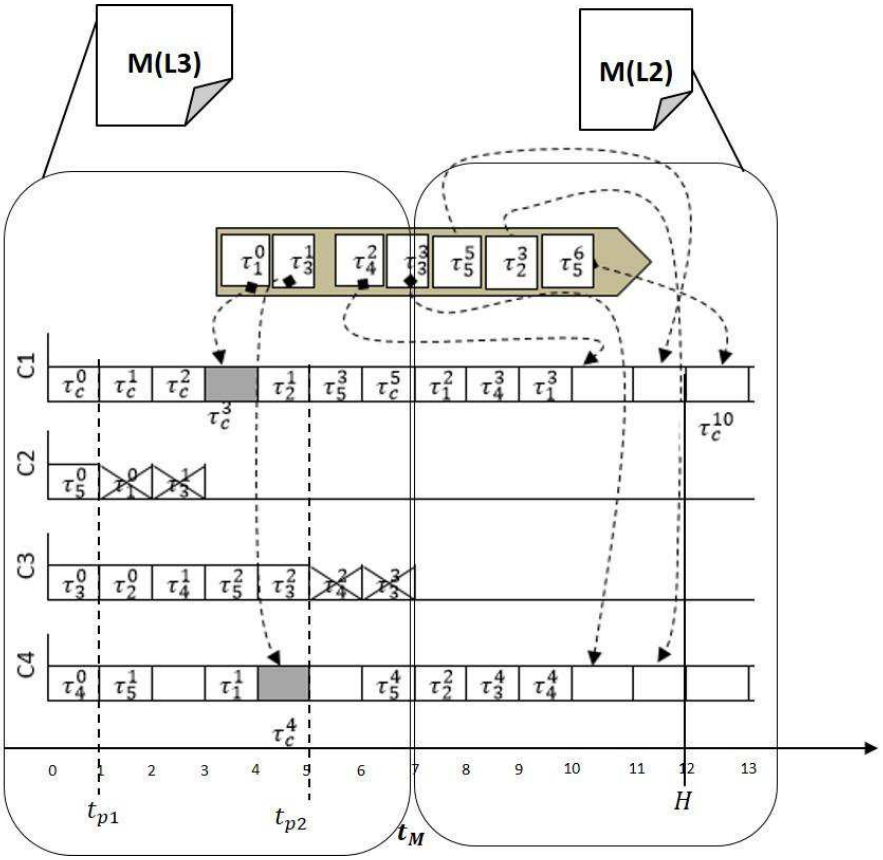

FIGURE 7.3 – Exemple d'ordonnancement multi-panne avec reprise par changement de mode de criticité.

II.3. Ordonnancement basé sur l'importance

II.3.a. Présentation de l'approche

Cette approche est inspirée de l'ordonnancement basé sur les valeurs. La valeur ici représente non plus une récompense ou une pénalité quant au respect du délai critique, mais l'importance de la tâche dans le système. Pour cela, nous considérons trois niveaux d'importance : le niveau *Secondaire* (S), le niveau *Préférable* (P) et le niveau *Vital* (V). Les tâches secondaires peuvent ne pas être exécutées sans préjudice pour la qualité de service. Si les tâches préférables ne sont pas exécutées, cela conduira à des résultats dégradés mais acceptables. Par contre, la

non-exécution des tâches vitales produit des résultats inacceptables. On procède alors à un partitionnement de l'ensemble des tâches du système par niveau d'importance et l'ordonnancement s'opère en fonction de l'importance de la tâche impactée par la panne. Notons que les niveaux d'importance sont définis par les concepteurs du système et donc l'importance est considérée par l'ordonnanceur comme un paramètre supplémentaire de la tâche. En l'absence de panne, l'ordonnancement des tâches s'opère sans tenir compte de l'importance des tâches.

On note U_{SPV} la charge induite par les tâches secondaires, préférables et vitales ; U_{PV} la charge induite par les tâches préférables et vitales ; U_V la charge induite uniquement par les tâches vitales. m_{SPV}, m_{PV} et m_V désignent les nombres minima de cœurs nécessaires pour supporter les charges U_{SPV}, U_{PV} et U_V respectivement ($m_{SPV} \geq m_{PV} \geq m_V$). Soit m_f le nombre de cœurs fonctionnels du processeur.

L'idée principale de notre approche est la suivante : l'ordonnancement débute avec $m_{SPV}+1$ cœurs ; à chaque fois qu'une panne est détectée :

- si $m_f \geq m_{SPV}$ l'ordonnancement se poursuit sans aucun abandon de tâche ;
- si $m_{PV} \leq m_f < m_{SPV}$ l'ordonnancement se poursuit avec les tâches préférables et les tâches vitales. Certaines tâches secondaires sont abandonnées ;
- si $m_V \leq m_f < m_{PV}$ l'ordonnancement se poursuit avec les tâches vitales. Les tâches secondaires et certaines tâches préférables sont abandonnées ;
- si $m_f < m_V$ le système ne peut plus rendre un service minimal et donc il est défectueux.

II.3.b. Sélection des tâches à abandonner

Soit U_{actuel} la charge du système courant, $U(\tau)$ le facteur d'utilisation de la tâche τ et $imp(\tau)$ la primitive qui retourne son niveau d'importance. Si $U_{actuel} > m_f$, on cherche à abandonner des tâches de façon à ce que U_{actuel} soit inférieur ou égal à m_f. Lorsqu'il faut abandonner des tâches d'un niveau d'importance I, deux objectifs peuvent être visées : maximiser la charge résiduelle et/ou minimiser le nombre de tâches abandonnées. On retrouve là le problème du sac à dos qui est NP-complet, d'où le recours à des heuristiques. Pour le premier objectif, deux heuristiques sont proposées : l'heuristique des tâches les plus légères (H1) et l'heuristique de la tâche marginale (H2). Pour le deuxième objectif, l'heuristique du nombre de tâches restantes (H3) est proposée.

Heuristique des tâches les plus légères

L'idée de cette heuristique est de maximiser le charge globale résiduelle en supprimant prioritairement les tâches les plus légères du niveau d'importance considéré jusqu'à l'obtention d'une charge adaptée au nombre de cœurs fonctionnels. L'Algorithme 6 présente les étapes de cette heuristique.

Algorithm 6: Heuristique des tâches légères

repeat
  supprimer τ_l, la tâche la plus légère d'importance I;
  $U_{actuel} \leftarrow U_{actuel} - U(\tau_l)$;
  $marge \leftarrow U_{actuel} - m_f$;
until $(marge \leq 0) \vee (\{\tau / imp(\tau) = I\} = \emptyset)$;

Heuristique de la tâche marginale

Tout comme la précédente heuristique, celle-ci vise aussi à maximiser la charge globale résiduelle tout en supprimant le moins de tâches possible. L'idée est d'abandonner le nombre de tâches qu'il faut pour juguler le surplus de charge. A cet effet, on abandonne à chaque itération la tâche la plus lourde parmi celles dont le facteur d'utilisation est inférieur ou égal à la marge. D'où l'Algorithme 7.

Algorithm 7: Heuristique de la tâche marginale

$marge \leftarrow U_{actuel} - m_f$;
repeat
  $\tau_{vict} \leftarrow \tau_v$ telle que $U(\tau_v) = max(U(\tau))$ avec $imp(\tau) = I \wedge U(\tau) \leq marge)$;
  supprimer(τ_{vict}) ;
  $U_{actuel} \leftarrow U_{actuel} - U(\tau_{vict})$;
  $marge \leftarrow U_{actuel} - m_f$;
until $(marge \leq 0) \vee (\{\tau, U(\tau) \leq marge \wedge imp(\tau) = I\} = \emptyset)$;
if $(marge > 0) \wedge$ $(\{\tau / imp(\tau) = I\} \neq \emptyset)$ **then**
  $\tau_{vict} \leftarrow$ la tâche restante la plus légère ;
  supprimer(τ_{vict}) ;
  $U_{actuel} \leftarrow U_{actuel} - U(\tau_{vict})$;
else
  problème impossible

Heuristique du nombre de tâches restantes

L'idée de cette heuristique est de garder le plus de tâches et donc de préserver autant que possible les fonctionnalités du système. Pour cela on abandonne la tâche la plus légère parmi celles dont le facteur d'utilisation est supérieure ou égal à la marge. Les instructions à suivre sont celles de l'Algorithme 8.

Algorithm 8: Heuristique du nombre de tâches restantes

$marge \leftarrow U_{actuel} - m_f$;
$V \leftarrow \{\tau / imp(\tau) = I \wedge (U(\tau) \geq marge)\}$;
if $V \neq \emptyset$ **then**
    supprimer la tâche la plus légère de V ;
else
    repeat
        supprimer la tâche d'importance I la plus lourde;
        $marge \leftarrow U_{actuel} - m_f$;
    until $(marge < 0) \vee (\{\tau / imp(\tau) = I\} = \emptyset)$;

II.3.c. Comparaison des heuristiques

Le tableau 7.3 présente les résultats de l'application des heuristiques (H1), (H2) et (H3) sur quatre exemples. Les paramètres d'entrées sont constitués des facteurs d'utilisation $U(\tau_i)$ des tâches du niveau d'importance à abandonner, la charge du système courant U_{actuel} et le nombre de cœurs fonctionnels m_f. Pour chaque heuristique le tableau ressort la valeur de U_{actuel} après l'abandon de tâches et le nombre de tâches restantes. Il en ressort que dans tous les cas, l'heuristique (H3) serait meilleure en terme de nombre de tâches restantes, ce qui correspond bien à son objectif. Cependant, on ne peut rien déduire de ces heuristiques pour ce qui est de la valeur de la charge actuelle et donc, aucune heuristique n'est meilleure que les autres. Les expérimentations que nous mènerons dans les futurs travaux permettront d'enrichir cette étude comparative.

TABLE 7.3 – Application des heuristiques sur des exemples

	Exemple 1	**Exemple 2**
Paramètres	$U(\tau_i)$: 0.9 ; 0.4 ; 0.3 ; 0.2 ; 0.1 $U_{actuel} = 6.5$; $m_f = 6$	$U(\tau_i)$: 0.9 ; 0.6 ; 0.4 ; 0.2 ; 0.1 $U_{actuel} = 3.8$; $m_f = 3$
H1	$U_{actuel} = 5.9$; 2 tâches restantes	$U_{actuel} = 2.5$; 1 tâches restante
H2	$U_{actuel} = 5.8$; 3 tâches restantes	$U_{actuel} = 2.8$; 3 tâches restantes
H3	$U_{actuel} = 5.6$; 4 tâches restantes	$U_{actuel} = 2.7$; 4 tâches restantes
	Exemple 3	**Exemple 4**
Paramètres	$U(\tau_i)$: 0.7 ; 0.3 ; 0.3 ; 0.1 ; 0.1 ; 0.1 ;0.1 $U_{actuel} = 2.4$; $m_f = 2$	$U(\tau_i)$: 0.7 ; 0.5 ; 0.5 $U_{actuel} = 2.6$; $m_f = 2$
H1	$U_{actuel} = 2$; 3 tâches restantes	$U_{actuel} = 1.7$; 1 tâche restante
H2	$U_{actuel} = 1.8$; 5 tâches restantes	$U_{actuel} = 1.7$; 1 tâche restante
H3	$U_{actuel} = 1.7$; 6 tâches restantes	$U_{actuel} = 1.9$; 2 tâches restantes

II.3.d. Exemple d'ordonnancement

Nous considérons à nouveau de plus le système
$\Psi : \tau_1 < 1, 3 >, \tau_2 < 2, 6 >, \tau_3 < 2, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$. $U = 2.16$ (et donc $m = 3$).
Le partitionnement de l'ensemble des tâches est donné dans le Tableau 7.4.

TABLE 7.4 – Partitionnement de l'ensemble des tâches du système Ψ

	Secondaires	Préférables	vitales
	τ_2, τ_3	τ_5	τ_1, τ_4
charge (U)	$\frac{10}{12}$	$\frac{7}{12}$	$\frac{9}{12}$

$U_V = \frac{9}{12} = 0.75$ et donc $m_V = 1$; $U_{PV} = \frac{7}{12} + \frac{9}{12} = \frac{16}{12} = 1.33$ et donc $m_{PV} = 2$; $U_{SPV} = \frac{10}{12} + \frac{16}{12} = \frac{26}{12} = 2.16$; $m_{SPV} = 3$ et donc l'ordonnancement débute sur 4 cœurs. On considère à nouveau deux défaillances survenant sur le cœur $C2$ à l'instant 1 et sur le cœur $C3$ à l'instant 5. Le délai de détection est de 2 unités de temps. Lorsque la première défaillance est détectée, aucune tâche n'est abandonnée car le nombre de cœurs fonctionnels restant est supérieur à la charge. Cependant à partir de l'instant 5 où la deuxième défaillance est détectée, les tâches secondaires (c-à-d τ_2 et τ_3) peuvent être abandonnées. On a $U_{actuel} = 2.16$, $m_f = 2$ et donc une marge à juguler de 0.16. L'utilisation de l'heuristique du nombre de tâches restantes conduit à l'abandon de la tâche τ_2 pour une charge résiduelle $U_{actuel} = 2.16 - \frac{2}{6} = 1.82 < m_f$.

La Figure 7.4 montre la séquence d'ordonnancement obtenue dans le scénario sans reprise et la Figure 7.5 la séquence dans le scénario avec reprise. Les sous-tâches τ_2^2 et τ_2^3 ne sont pas ordonnancées.

Des expérimentations seront menées dans les prochains travaux pour conjecturer sur chaque heuristique et réaliser une étude comparative entre elles.

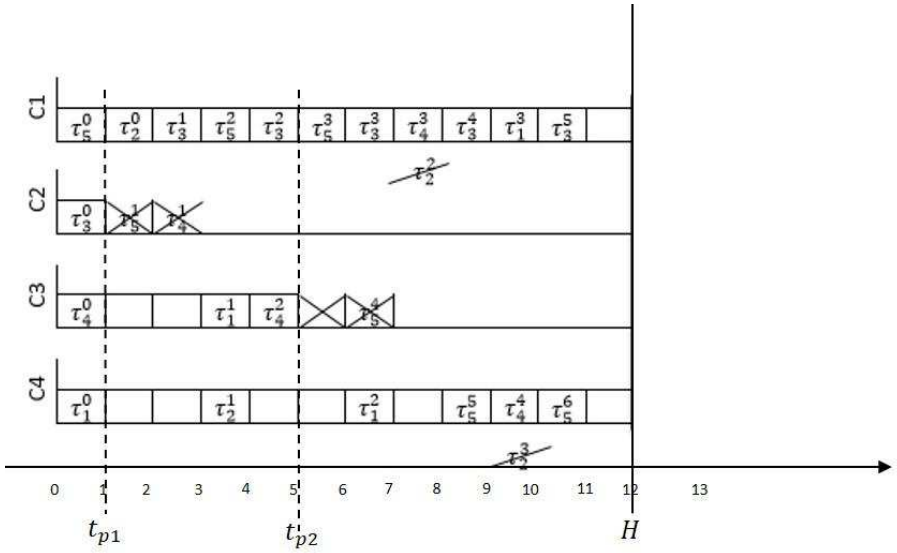

FIGURE 7.4 – Exemple d'ordonnancement multi-panne sans reprise par abandon des tâches secondaires.

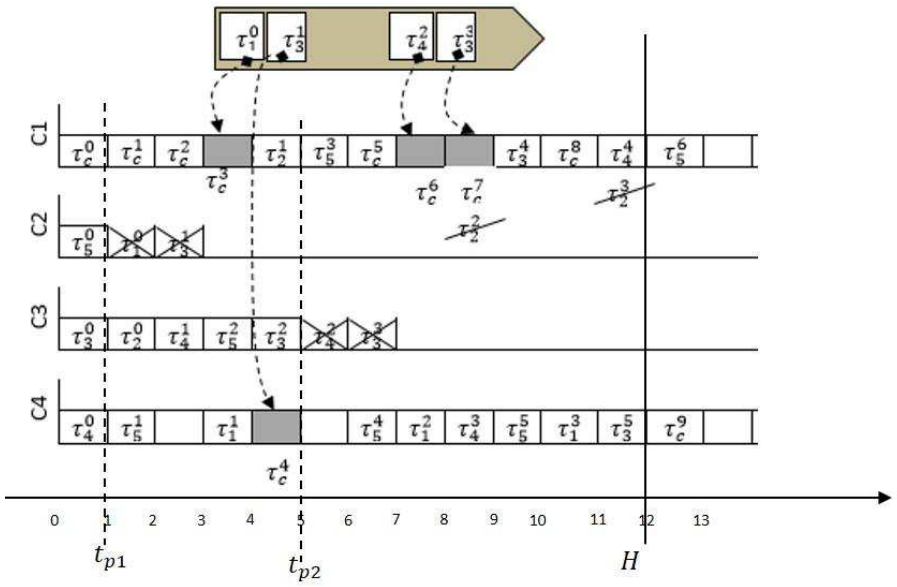

FIGURE 7.5 – Ordonnancement multi-panne avec reprise par abandon des tâches secondaires.

Conclusion

Dans ce chapitre, nous avons abordé la problématique de l'ordonnancement d'un système temps-réel en présence de défaillances multiples de plusieurs cœurs du processeur. Nous nous sommes appuyés sur les approches classiques que sont le changement de mode de criticité et l'ordonnancement basé sur les valeurs pour proposer deux approches d'ordonnancement multi-panne par un algorithme équitable. Dans la première approche, plusieurs niveaux de criticité sont définis en fonction du nombre de coeurs fonctionnels restants et le changement de mode de criticité permet d'adapter la charge du système au nombre de coeurs restants. La deuxième approche quant à elle procède à un partitionnement de l'ensemble des tâches du système en fonction du niveau d'importance. Elle nécessite cependant de définir une heuristique qui permet d'alléger le système à chaque occurrence de panne tout en maintenant une charge maximale et/ou en réduisant le nombre de tâches abandonnées. Il ressort de cette étude que la définition d'une technique de tolérance multi-panne sur la base de ces approches est avant tout une problématique liée à la conception du système. En effet, c'est aux concepteurs d'estimer le degré d'importance des tâches ou bien de définir les niveaux de criticité en fonction de la nature du système, du rôle de chaque tâche qui le constitue et de la plateforme matérielle d'exécution.

Chapitre

8 Conclusion et Perspectives

Sommaire

Résumé

Ce chapitre récapitule les travaux effectués dans cette thèse et présente les perspectives de recherche qui se dégagent.

I. Conclusion Générale

L'avènement des plateformes multicoeurs a permis aux constructeurs de réaliser une avancée remarquable dans les performances des processeurs et de satisfaire les demandes des industriels qui souhaitent des ordinateurs moins encombrants, plus fins et plus légers. Cette évolution technologique augmente la productivité dans des ordinateurs de plus petite taille capables d'exécuter simultanément plusieurs applications complexes et de réaliser davantage de tâches en moins de temps. Pour les applications temps-réel critiques où les fautes temporelles peuvent avoir des conséquences catastrophiques, l'utilisation d'une plateforme d'exécution multicoeur augmente les garanties en terme de respect des échéances des tâches. Plusieurs secteurs d'activité (aéronautique, automobile, télécommunication,..) expriment un fort intérêt pour les processeurs multicœurs, obligeant par là-même les concepteurs de systèmes à en évaluer les contraintes pour garantir une sûreté de fonctionnement. L'un des moyens de mise en œuvre de la sûreté de fonctionnement est la tolérance aux pannes, c'est-à-dire, la capacité du système à délivrer un service conforme aux spécifications en dépit d'une panne. Dans cette thèse, nous nous sommes intéressés à la tolérance des systèmes temps-réel multicoeurs à des pannes matérielles permanentes affectant durablement un ou plusieurs cœurs du processeur. Le système, constitué de tâches indépendantes et périodiques, à départs simultanés et à échéances sur requête est ordonnancé suivant une politique globale par l'algorithme équitable PD2. Cet algorithme procède à un découpage temporel des tâches en sous-tâches qui représentent chacune une exécution de la tâche pendant une unité de temps. L'objectif est de construire une séquence d'ordonnancement valide en dépit de la défaillance d'un ou plusieurs cœurs. La détection de la panne n'étant pas nécessairement instantanée, les sous-tâches affectées à un cœur défaillant entre l'instant de survenue de la panne et l'instant de sa détection sont perdues. En fonction de la nature des tâches dans le système, il faudra ou pas récupérer cette perte d'exécution. C'est pourquoi, deux scénarios ont été envisagés : l'ordonnancement sans reprise des sous-tâches perdues et l'ordonnancement avec reprise des sous-tâches perdues.

Tout d'abord, nous avons supposé qu'un seul cœur tombait en panne tout au long de l'ordonnancement. Dans le scénario sans reprise, nous avons montré que la Technique de la Redondance Matérielle Limitée (TRML) qui consiste à prévoir un seul cœur de plus que le minimum nécessaire garantit la validité de la séquence. Dans le scénario avec reprise, trois techniques de tolérance ont été proposées. La première technique TSS (Technique des Sous-tâches de Substitution) augmente le WCET de chaque tâche en fonction du délai de détection pour prévoir des sous-tâches supplémentaires qui seront utilisées en remplacement des sous-tâches perdues. Si cette technique est sûre et facile à mettre en œuvre, elle nécessite plus de redondance matérielle et engendre une surcharge du système. La deuxième technique (TCR : Technique Contraindre puis Relâcher) contraint l'échéance des tâches au début de l'ordonnancement puis les relâche après la panne. En procédant ainsi, elle crée une marge de temps (appelée marge de tolérance) entre le délai critique et la période de chaque tâche qui peut être utilisée pour récupérer les sous-tâches perdues. C'est une technique plus flexible que la précédente qui repose sur la redon-

dance matérielle limitée à un cœur. Cependant, elle impose plusieurs contraintes au système à ordonnancer et soulève un problème d'inversion de priorité qui complique la preuve. La troisième technique (TFA : Technique du Flux Apériodique) procède à une répartition équitable des unités de temps creux qu'elle utilise ensuite pour réordonnancer les sous-tâches perdues. Elle a l'avantage de s'appliquer à tout système ordonnançable, mais nécessite qu'on s'autorise un nombre limité de dépassements d'échéances. Cependant, la récupération du système s'opère en un temps borné. Enfin, le partitionnement de l'ensemble des tâches en trois catégories selon leurs natures (arretable, retardable et stricte) permet une utilisation conjointe des techniques TRML, TCR et TFA. Ainsi, lorsqu'une panne est détectée, si la tâche impactée est arretable, les sous-tâches perdues ne sont pas récupérées. Par contre, si elle est retardable, les sous-tâches perdues sont récupérées dans les unités de temps creux. Si elle est stricte, les sous-tâches perdues seront récupérées dans sa marge de tolérance.

Ensuite, nous avons considéré la défaillance de plusieurs cœurs du processeur. Nous avons alors étudié les possibilités d'extension des techniques ci-dessus à ce contexte. En effet, la défaillance de plusieurs cœurs peut avoir pour conséquence qu'à un instant donné, le nombre de cœurs fonctionnels ne garantit plus l'ordonnançabilité du système. Il faut alors réduire la charge du système pour l'adapter à la nouvelle configuration. Deux approches d'ordonnancement ont été proposées. La première suppose que le système est conçu avec plusieurs modes de criticité correspondant au nombre de cœurs fonctionnels. Les paramètres des tâches sont alors définis pour chaque mode de criticité. Lorsqu'une panne est détectée un changement de mode de criticité permet au système de s'adapter à la nouvelle configuration. La deuxième approche procède également sur un partitionnement des tâches selon leur importance pour le système. A la détection d'une nouvelle panne, certaines tâches ou travaux sont abandonnées en fonction de leur importance, de manière à maximiser la charge restante et/ou à minimiser le nombre de tâches abandonnées.

II. Perspectives

Les travaux menés dans le cadre de cette thèse ouvrent de nombreuses perspectives de recherche que nous comptons explorer. Pour ce qui est de l'ordonnancement mono-panne avec reprise présenté au Chapitre 6, deux perspectives se dégagent. En effet, nous avons présenté la technique TCR dont l'expérimentation avec le prototype FTA a produit des résultats concluants. La preuve a été élaborée dans le cas où la panne survient à n'importe quel instant et qu'il n'existe pas de sous-tâche retardée. Lorsqu'il existe des sous-tâches retardées, nous avons prouvé la validité de la technique si la panne survient à l'instant d'une hyperpériode (c-à-d $t_p = 0[H]$) et est détectée l'instant d'après (c-à-d $x = 1$). Ensuite, nous avons indiqué des éléments de généralisation de cette preuve pour le même instant de panne, mais avec un délai de détection plus large. Dans les travaux futurs, nous complèterons cette preuve pour le cas d'une panne survenant à un instant quelconque et pour un délai de détection arbitraire.

En deuxième perspective, nous mènerons une expérimentation de la technique combinée qui, en fonction de la nature de la tâche impactée, applique soit un ordonnancement sans reprise, soit un ordonnancement avec reprise par la technique TCR ou par la technique TFA. Ensuite, nous élaborerons la preuve de cette technique en nous inspirant de celles des techniques TRML, TCR et TFA.

Dans le cas de l'ordonnancement multi-panne (Chapitre 7), les approches proposées (changement de mode de criticité et abandon de tâche par ordre d'importance) n'ont pas encore fait l'objet d'une expérimentation. Nous comptons donc tester chaque approche sur une large gamme de systèmes, ce qui nous permettra de les caractériser et de les comparer. Pour l'approche basée sur l'importance en particulier, trois heuristiques ont été proposées pour discriminer entre les tâches à abandonner. Nous comptons tester ces trois heuristiques et mener une étude comparative qui pourra déboucher sur des recommandations.
Par ailleurs, nous envisageons une utilisation couplée des deux approches multi-pannes. En fonction de la marge qu'il y a entre la charge actuelle du système et le nombre de cœurs fonctionnels, l'idée est d'abandonner certaines tâches en fonction de leur importance au moment du changement de mode de criticité.

Enfin, dans la technique TFA, le ré-ordonnancement des sous-tâches perdues se fait uniquement dans les unités de temps creux identifiées à l'aide des sous-tâches creuses. Or, il existe aussi des unités de temps creux (appelées unités de temps creux par épuisement) qui surviennent lorsque le nombre de sous-tâches prêtes est inférieur au nombre de coeurs fonctionnels et qui sont dus à l'utilisation avant la panne du cœur supplémentaire. Dans les futurs travaux, nous étudierons la possibilité d'intégrer ces unités de temps creux par épuisement et de voir leur impact sur la borne de récupération. Cela nous permettra d'affiner nos techniques de tolérance mono-panne et multi-panne.

Annexes

Lemme 4. *$L(t_0)$:*
Soit $t1 = Exec(\tau_b^0, Seq_{[m]}^{(S(I))})$, s'il existe $t_0 \times m$ sous-tâches de pseudo-échéances inférieures ou égales à t_0 ($t_0 \leq t1$), alors la charge $U(t_0)$ induite par les tâches qui ont des exécutions dans $[0, t_0]$ est telle que $U(t_0) \geq m$ ■

La Figure 8.1 recense les sous-tâches ordonnancées dans la séquence $Seq_{[m]}^{(S(I))}$ dans l'intervalle de temps $[0, t_1]$, les notations étant explicitées en début de preuve.

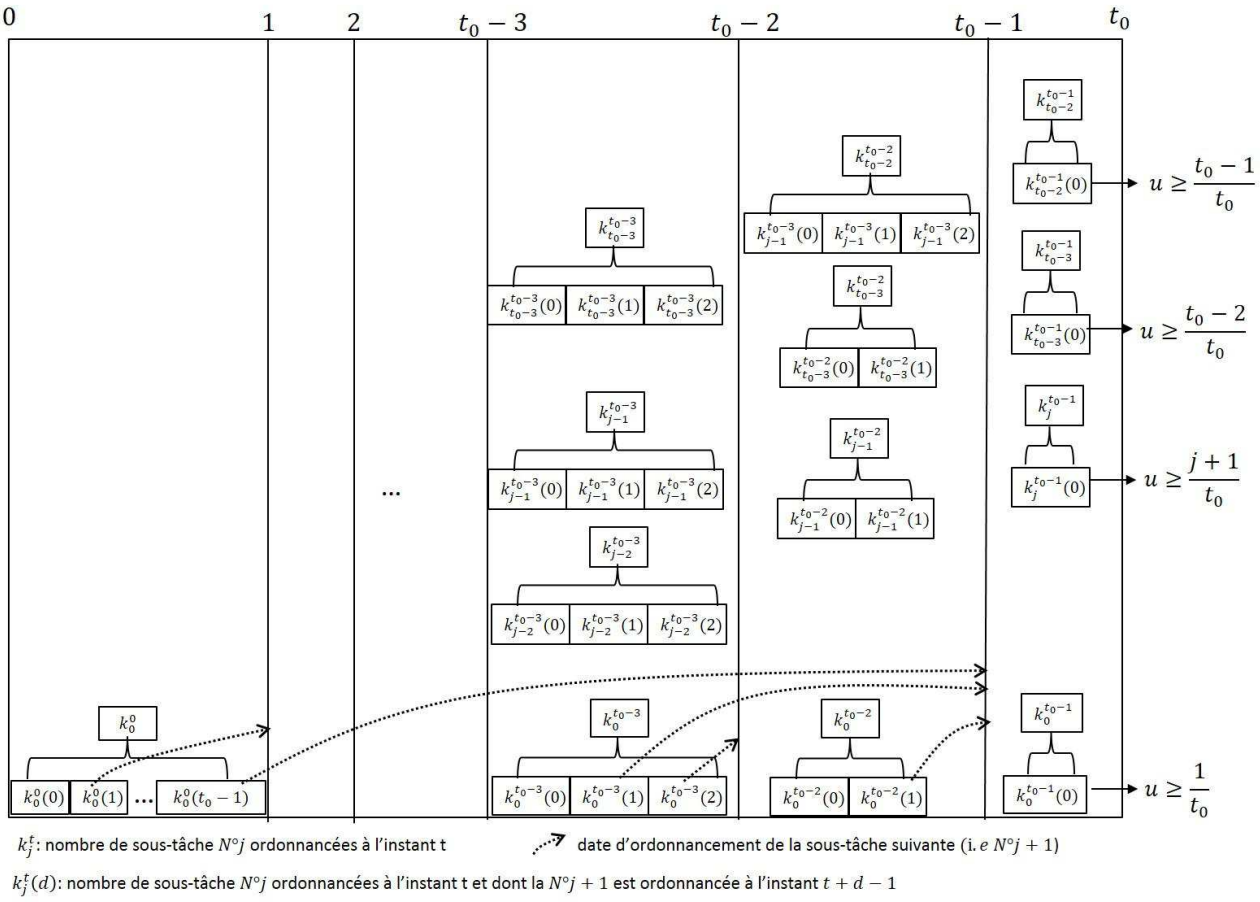

FIGURE 8.1 – Sous-tâches ordonnancées dans le système intermédiaire dans l'intervalle [0, t_1]

Démonstration. $(L(t_0))$
Pour commencer, nous introduisons les notations suivantes afin de partitionner l'ensemble des sous-tâches ordonnancées dans l'intervalle de temps $[0, t_0]$:

$N^o k$: sous-tâche τ_i^j telle que $j = k$. En d'autres termes, il s'agit des k^e sous-tâches des différentes tâches ;

Soit τ_i^j une sous-tâche d'une tâche τ_i. On rappelle que τ_i^{j-1} est la sous-tâche précédente de τ_i^j, τ_i^{j+1} la sous-tâche suivante et τ_i sa tâche mère.

Dans la sous-séquence de $Seq_{[m]}^{(S(I))}$ correspondant à l'intervalle de temps $[0, t_0]$, On définit les ensembles suivants :

$E_j(t) = \{\tau_i^j, 1 \leq i \leq n$ telles que $Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) = t\}$: ensemble des sous-tâches $N^o j$ ordonnancées à l'instant t.
On pose $k_j^t = card(E_j(t))$

$E_j(t,d) = \{\tau_i^j, 1 \leq i \leq n$ telles que $Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) = t \wedge Exec(\tau_i^{j+1}, Seq_{[m]}^{(S(I))}) = t + d\}$: ensemble des sous-tâches $N^o j$ ordonnancées à l'instant t et dont la suivante (qui donc est une $N^o j + 1$) est ordonnancée à l'instant $t + d$. Si $d = 0$, $Exec(\tau_i^{j+1}, Seq_{[m]}^{(S(I))}) \geq t_0$.
On pose $k_j^t(d) = card(E_j(t,d))$
Il en découle que $E_j(t) = \cup_{d=0}^{t_0-t-1} E_j(t,d)$ (et il s'agit d'une partition).

$E(t) = \{\tau_i^j, 1 \leq i \leq n$ et $j \in \mathbb{N}$ tel que $Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) = t\}$: ensemble de toutes les sous-tâches ordonnancées à l'instant t. On a $E(t) = \cup_{j=0}^{t} E_j(t)$

$E = \{\tau_i^j, Exec(\tau_i^j, Seq_{[m]}^{(S(I))}) < t_0\} = \cup_{t=0}^{t1-1} E(t)$: ensemble des sous-tâches ordonnancées avant l'instant t_0

Notons qu'il n'y a pas de temps creux sur $[0, t_0]$ puisque τ_b^0 n'est pas encore ordonnancée. D'où, il y a m sous-tâches ordonnancées à tout instant t et en tout $m \times t_0$ sous-tâches ordonnancées avant l'instant t_0 et donc $Card(E(t)) = m$, $Card(E) = m \times t_0$.
Parmi les sous-tâches ordonnancées à un instant t quelconque, on peut avoir des sous-tâches $N^o 0$, $N^o 1$, $N^o 2$,.., $N^o t$ et donc $\sum_{j=0}^{t} k_j^t = m$ (1).
Par ailleurs, on établit les relations suivantes :
$k_j^t = \sum_{d=0}^{t_0-t-1} k_j^t(d)$ (2)
$k_j^t = \sum_{p=j-1}^{t-1} k_{j-1}^p(t-p)$, $j > 0$ (3) (partitionnement en fonction de la sous-tâche qui précède).

Notre hypothèse est que si $\tau_i^j \in E$ alors $d_i^j \leq t_0$. Le facteur de charge u_i de la tâche mère de τ_i^j dans chaque cas de figure est telle que :

— Si $\tau_i^j \in E_j(t,0) \Longrightarrow d_i^j \leq t_0$ on a $\dfrac{j+1}{u_i} \leq \left\lceil \dfrac{j+1}{u_i} \right\rceil \leq t_0 \Longrightarrow u_i \geq \dfrac{j+1}{t_0}$.

— Si $\tau_i^j \in E_{t_0-1}(t_0-1,0) \Longrightarrow d_i^{t_0-1} \leq t_0$ on a $\dfrac{t_0}{u_i} \leq \left\lceil \dfrac{t_0}{u_i} \right\rceil \leq t_0 \Longrightarrow u_i \geq 1$.

Donc, comme $\forall i, u_i < 1$ on a $k_{t_0-1}^{t_0} = 0$.

D'où, si $\tau_i^j \in E_j(t,0)$ et $\tau_{i'}^{j'} \in E_{j'}(t',0)$, alors $j' \neq j \Longrightarrow i \neq i'$.

Recenser toutes les tâches intervenant dans le calcul de $E(t_0)$ revient donc à déterminer les tâches mères des sous-tâches dans $E_j(t,0)$, $j = 0..t_0-1$ et $t = 0..t_0-1$.

On note $Mere_j(t,0) = \{\tau_i / \exists \tau_i^j \in E_j(t,0)\}$. $(j,t) \neq (j',t') \Longrightarrow Mere_j(t,0) \cap Mere_{j'}(t',0) = \emptyset$ et on a $Card(Mere_j(t,0)) = k_j^t(0)$.

La charge totale du système U et la charge induite $U(t_0)$ sont telles que :

$U > U(t_0) \geq \sum_{\tau_i \in \cup Mere_j(t,0)} u_i \geq \sum_{t=0}^{t_0-2}(\sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0))$

$= \sum_{t=0}^{t_0-2}(\sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0)) + \sum_{j=0}^{t_0-2} \dfrac{j+1}{t_0} k_j^{t_0-1}$

$\Leftrightarrow U(t_0) \geq \sum_{j=0}^{t_0-2} \dfrac{k_j^{t_0-1}}{t_0} + \sum_{j=0}^{t_0-2} \dfrac{j}{t_0} k_j^{t_0-1} + \sum_{t=0}^{t_0-2}(\sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0))$

$= \dfrac{m}{t_0}(par(1)) + \sum_{j=1}^{t_0-2} \dfrac{j}{t_0}(\sum_{t=j-1}^{t_0-2} k_{j-1}^t(t_0-1-t))(par(3)) + \sum_{t=0}^{t_0-2}(\sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0).$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-2}(\sum_{j=1}^{Min(t+1,t_0-2)} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1)) + \sum_{t=0}^{t_0-2}(\sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0).$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-3} \sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1)) + \sum_{j=1}^{t_0-2} \dfrac{j}{t_0} k_{j-1}^{t_0-2}(1) + \sum_{t=0}^{t_0-2} \sum_{j=0}^{t} \dfrac{j+1}{t_0} k_j^t(0).$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1) + \sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(0)) + \sum_{j=0}^{t_0-2}(\dfrac{j}{t_0} k_{j-1}^{t_0-2}(1) + \dfrac{j+1}{t_0} k_j^{t_0-2}(0)).$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1) + \sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(0)) + \sum_{j=1}^{t_0-2}(\dfrac{j}{t_0}(k_{j-1}^{t_0-2}(1) + k_{j-1}^{t_0-2}(0))$

$+ \dfrac{t_0-1}{t_0} k_{t_0-2}^{t_0-2}(0)).$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1) + \sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(0)) + \sum_{j=1}^{t_0-1} \dfrac{j}{t_0} k_{j-1}^{t_0-2} + \dfrac{t_0-1}{t_0} k_{t_0-2}^{t_0-2}.$

$\Leftrightarrow U(t_0) \geq \dfrac{m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(t_0-t-1) + \sum_{j=1}^{t+1} \dfrac{j}{t_0} k_{j-1}^t(0)) + \sum_{j=1}^{t_0-1} \dfrac{j}{t_0} k_{j-1}^{t_{01}-2}.$

$\Leftrightarrow U(t_0) \geq \frac{m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \frac{j}{t_0} k_{j-1}^t(t_0-t-1) + \sum_{j=1}^{t+1} \frac{j}{t_0} k_{j-1}^t(0)) + \sum_{j=1}^{t_0-1} \frac{k_{j-1}^{t_0-2}}{t_0} + \sum_{j=2}^{t_0-1} \frac{j-1}{t_0} k_{j-1}^{t_0-2}.$

$\Leftrightarrow U(t_0) \geq \frac{2m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1} \frac{j}{t_0}(k_{j-1}^t(t_0-t-1) + k_{j-1}^t(0) + \sum_{j=2}^{t_0-1}(\sum_{t=j-2}^{t_0-1} k_{j-2}^t(t_0-2-t))(\frac{j-1}{t_0}).$

$\Leftrightarrow U(t_0) \geq \frac{2m}{t_0} + \sum_{t=0}^{t_0-3} \sum_{j=1}^{t+1}(k_{j-1}^t(t_0-t-1) + k_{j-1}^t(0)) \frac{j}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=2}^{t+2}(\frac{j-1}{t_0})(k_{j-2}^t(t_0-t-2)).$

$\Leftrightarrow U(t_0) \geq \frac{2m}{t_0} + \sum_{t=0}^{t_0-3}(\sum_{j=1}^{t+1}(k_{j-1}^t(t_0-t-2) + k_{j-1}^t(t_0-t-1) + k_{j-1}^t(0))(\frac{j}{t_0}).$

On pose $P_s(t_0) = \sum_{t=0}^{s}(\sum_{j=1}^{t+1} k_{j-1}^t(0) + \sum_{u=s-t+1}^{t_0-t-1} k_{j-1}^t(u))(\frac{j}{t_0})$

On en déduit alors que $U(t_0) \geq \frac{2m}{t_0} + P_{t_0-3}(t_0)$.

On considère la propriété suivante :

Propriété 2. $P_s(t_0) = \frac{m}{t_0} + P_{s-1}(t)$, $1 \leq s \leq t_0 - 3$ ■

La preuve de cette propriété est donnée après celle du présent lemme.
On déduit de cette propriété que :
$P_{t_0-3}(t_0) = \frac{m}{t_0} + P_{t_0-4}(t_0)$
$P_{t_0-4}(t_0) = \frac{m}{t_0} + P_{t_0-5}(t_0)$
...
$P_2(t_0) = \frac{m}{t_0} + P_1(t_0)$
$P_1(t_0) = \frac{m}{t_0} + P_0(t_0).$
Par simplification membre à membre, il ressort que
$P_{t_0-3}(t_0) = \frac{t_0-3}{t_0} m + P_0(t_0) = \frac{t_0-2}{t_0} m.$
D'où, $U(t_0) \geq \frac{2m}{t_0} + \frac{t_0-2}{t_0} m = m \Leftrightarrow U(t_0) \geq m.$

Étant donné qu' à l'instant t_0 la sous-tâche τ_c^0 n'est pas encore ordonnancée car $Exec(\tau_c^0, Seq_{[m]}^{(S(I))}) \geq t_0$, on a $U > U(t_0)$ et donc $U > m$. □

Démonstration. $P_s(t_0)$
$P_s(t_0) = \sum_{t=0}^{s}(\sum_{j=1}^{t+1} k_{j-1}^t(0) + \sum_{u=s-t+1}^{t_0-t-1} k_{j-1}^t(u))(\frac{j}{t_0})$
$= \sum_{t=0}^{s-1}(....) + \sum_{j=1}^{s+1}(k_{j-1}^s(0) + \sum_{u=1}^{t_0-s-1} k_{j-1}^s(u))(\frac{j}{t_0})$
$= \sum_{t=0}^{s-1}(....) + \sum_{j=1}^{s+1} \sum_{u=0}^{t_0-s-1} k_{j-1}^s(u))(\frac{j}{t_0})$

$$= \sum_{t=0}^{s-1}(....) + \sum_{j=1}^{s+1} k_{j-1}^{s}\frac{j}{t_0}$$
$$= \sum_{t=0}^{s-1}(....) + \sum_{j=1}^{s+1} \frac{k_{j-1}^{s}}{t_0} + \sum_{j=1}^{s+1} \frac{j-1}{t_0}k_{j-1}^{s}$$
$$= \frac{m}{t_0} + \sum_{t=0}^{s-1}(\sum_{j=1}^{t+1} k_{j-1}^{t}(0) + \sum_{u=s-t+1}^{t_0-t-1} k_{j-1}^{t}(u))\frac{j}{t_0} + \sum_{j=1}^{s} \frac{j}{t_0}(\sum_{t=j-1}^{s-1} k_{j-1}^{t}(s-t)$$
$$= \frac{m}{t_0} + \sum_{t=0}^{s-1}(\sum_{j=1}^{t+1}(k_{j-1}^{t}(0) + \sum_{u=s-t+1}^{t_0-t-1} k_{j-1}^{t}(u)))\frac{j}{t_0} + \sum_{t=0}^{s-1}\sum_{j=1}^{t+1} \frac{j}{t_0}k_{j-1}^{t}(s-t)$$
$$= \frac{m}{t_0} + \sum_{t=0}^{s-1}(\sum_{j=1}^{t+1}(k_{j-1}^{t}(0) + \sum_{u=s-t+1}^{t_0-t-1} k_{j-1}^{t}(u) + k_{j-1}^{t}(s-t))\frac{j}{t_0}$$
$$= \frac{m}{t_0} + \sum_{t=0}^{s-1}(\sum_{j=1}^{t+1}(k_{j-1}^{t}(0) + \sum_{u=s-t}^{t_0-t-1} k_{j-1}^{t}(u))\frac{j}{t_0}$$
$$= \frac{m}{t_0} + P_{s-1}(t_0)$$

□

Table des figures

Liste des tableaux

Bibliographie

[1] Y. Ahangari Nanehkaran and S. Ahmadi. The challenges of multi-core processor. *International Journal of Advancements in Research and Technology*, 2(1) :36–39, June 2013.

[2] F. Kong, W. Yi, and Q. Deng. Energy-efficient scheduling of real-time tasks on cluster-based multicores. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[3] Thomas Megel. *Placement, Ordonnancement et Mécanismes de Migration de Tâches Temps-réel pour des Architectures Distribuées Multicoeurs*. PhD thesis, Université de Toulouse, 2012.

[4] S. M'Sirdi, W. Godard, and M. Pantel. Intégration d'applications avioniques sur processeurs multicoeurs. In *Actes de l'Ecole d'été Temps Réel (ETR'2015)*, pages 159–162, 2015.

[5] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 307–316, New York, NY, USA, 2010. ACM.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[7] R. Davis. A review of fixed priority and edf scheduling for hard real-time uniprocessor systems. *SIGBED Rev.*, 11(1) :8–19, February 2014.

[8] F. Cottet, E. Grolleau, S. Gérard, J. Hugues, Y. Ouhammou, and S. Tucci-Piergiovanni. *Systèmes Temps-Réel Embarqués : Spécification, Conception, Implémentation et Validation Temporelle (2e édition)*. DUNOD, 2014.

[9] M. Chéramy, A.M. Déplanche, and P.E Hladik. Ordonnancement temps-réel : des politiques monoprocesseurs aux politiques multiprocesseurs. *Archives Ouvertes*, hal-00662741, January 2012. Published by Foundation of Computer Science (FCS), NY, USA.

[10] C. Liu, L. Layland, and W. James. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, January 1973.

[11] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[12] J.Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4) :237 – 250, 1982.

[13] J. Goossens and N. Navet. Ordonnancement temps-réel multiprocesseur. In *Systèmes Temps Réel — Ordonnancement, Réseaux et Qualité de Service*, chapter 2. Hermès, 2006.

[14] S. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 360–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] G. C. Buttazzo. Rate monotonic vs. edf : Judgment day. *Real-Time Syst.*, 29(1) :5–26, January 2005.

[16] E. Grolleau. Ordonnancement et ordonnançabilité monoprocesseur. In *Actes de l'Ecole d'été Temps Réel (ETR'2011)*, 2011.

[17] A. Choquet-Geniet and G. Largeteau-Skapin. Size analysis in multiprocessor real-time scheduling. *International Journal Critical Computer-Based Systems*, 2014, 2014.

[18] Wind river. vxworks,.

[19] M. Barabanov. *A Linux-based Real-Time Operating System*. PhD thesis, New Mexico Institute of Mining and Technology, 1997.

[20] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2) :1–74, June 2004.

[21] M. Stilkerich, C. Wawersich, A. Gal, W. Schröder-Preikschat, and M. Franz. Osek/vdx api for java. In *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems : Linguistic Support for Modern Operating Systems*, PLOS '06, New York, NY, USA, 2006. ACM.

[22] IEEE Std 1003.1. Posix specification.

[23] S. Pailler. *Analyse Hors Ligne d'Ordonnançabilité d'Applications Temps Réel comportant des Tâches Conditionnelles et Sporadiques*. PhD thesis, École Nationale Supérieure de Mécanique et d'Aérotechnique, 2006.

[24] K. Tindell. Deadline monotonic analysis. *Embedded Systems Programming*, pages 20–38, 2000.

[25] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : A flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada : The Engineering of Correct and Reliable Software for Real-time &Amp ; Distributed Systems Using Ada and Related Technologies*, SIGAda '04, pages 1–8, New York, NY, USA, 2004. ACM.

[26] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu. *Exact Schedulability Analysis for Static-Priority Global Multiprocessor Scheduling Using Model-Checking*, pages 263–272. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[27] E. Grolleau and A. Choquet-Geniet. Off-line computation of real-time schedules using petri nets. *Discrete Event Dynamic Systems, DEDS*, 2002(3) :311–333, 2002.

[28] Chauviere B., Geniet D., and Schott R. Contributions to the multiprocessor scheduling problem. In *Computational Intelligence (IASTED-CI)*, pages 55–60, 2007.

[29] J. H. Anderson. A new look at pfair priorities. Technical report, In Submission, 1999.

[30] T. Kuroda. Cmos design challenges to power wall. In *Microprocesses and Nanotechnology Conference, 2001 International*, pages 6–7, Oct 2001.

[31] S. Moreaud. *Mouvement de Données et Placement des Tâches pour les Communications Haute Performance sur machines Hiérarchiques.* PhD thesis, Université de Bordeaux I, 2011.

[32] S. D'souza, A. Bhat, and R. Rajkumar. Sleep scheduling for energy-savings in multi-core processors. In *Euromicro Conference on Real-Time Systems(ECRTS 2016)*, pages 226–236, jul 2016.

[33] Sylvain Genevès. *Améliorations de Performances des Systèmes Multicœur : Environnement d'Exécution Evénementiel Efficace et Etude Comparative de Modèles de Programmation.* PhD thesis, Université de Grenoble, 2006.

[34] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multicore real-time scheduling for generalized parallel task models. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 217–226, Washington, DC, USA, 2011. IEEE Computer Society.

[35] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 927–930, July 2009.

[36] M. H. Mottaghi and H. R. Zarandi. Dfts : A dynamic fault-tolerant scheduling for real-time tasks in multicore processors. *Microprocess. Microsyst.*, 38(1) :88–97, February 2014.

[37] J.E.D.E. Failure mechanisms and models for semiconductor devices.

[38] *Exécution Prédictible sur Processeurs Pluri-coeurs.* PhD thesis.

[39] S. Malhotra, P. Narkhede, K. Shah, S. Makaraju, and M. Shanmugasundaram. A review of fault tolerant scheduling in multicore systems. *International Journal of Scientific and Technology Research*, 4(5) :132–136, 2015.

[40] S. Gotoda, M. Ito, and N. Shibata. Task scheduling algorithm for multicore processor system for minimizing recovery time in case of single node fault. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 260–267, Washington, DC, USA, 2012. IEEE Computer Society.

[41] D. Maume. Des processeurs multicoeurs capables de s'autoréparer.

[42] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, 1996.

[43] A. M. Déplanche. Ordonnancement temps-réel multiprocesseur- panorama sur les politiques globales, 2011.

[44] J. El Hayek. *Le Problème de Bin-Packing en Deux-Dimensions, le Cas Non-Orienté : Résolution Approchée et Bornes Inférieures.* PhD thesis, Université de Technologie de Compiègne, 2006.

[45] R. E. Korf. A new algorithm for optimal bin packing. In *Eighteenth National Conference on Artificial Intelligence*, pages 731–736, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[46] R. E. Korf. An improved algorithm for optimal bin packing. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1252–1258, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[47] P. Shaw. *A Constraint for Bin Packing*, pages 648–662. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[48] C. C. Lee and D. T. Lee. A simple on-line bin-packing algorithm. *J. ACM*, 32(3) :562–572, July 1985.

[49] D. Oh and T. P. Bakker. Utilization bounds for n-processor rate monotonescheduling with static processor assignment. *Real-Time Syst.*, 15(2) :183–192, September 1998.

[50] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Oper. Res.*, 26(1) :127–140, February 1978.

[51] U. C. Devi and J. H. Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Syst.*, 38(2) :133–189, February 2008.

[52] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus rt : A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[53] B. Andrea, B. Björn, and J. H. Anderson. Is semi-partitioned scheduling practical ? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, ECRTS '11, pages 125–135, 2011.

[54] P. Holman and J. H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4) :543–564, December 2005.

[55] S. Ramamurthy. Scheduling periodic hard real-time tasks with arbitrary deadlines on multiprocessors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 59–, Washington, DC, USA, 2002. IEEE Computer Society.

[56] S. Malo, A. Choquet-Geniet, and M. Bikienga. Pfair scheduling of late released tasks with constrained deadlines. In *4e Colloque National sur la Recherche en Informatique et ses Applications*, pages 142 – 149, 2012.

[57] Mohamed-Lamine BOUKHANOUFA. *Adaptabilité et Reconfiguration des Systèmes Temps-Réel Embarqués*. PhD thesis, Université Paris Sud, 2012.

[58] Villemeur A. *Sûreté de Fonctionnement des Systèmes Industriels*. Eyrolles, 1997.

[59] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Sûreté de fonctionnement informatique : Concepts, défis, directions, 2004.

[60] J. Arlat, Y. Crouzet, Y. Deswarte, J.C. Fabre, J.C. Laprie, and D. Powell. *Tolérance aux Fautes*, chapter La dimension technologique des systèmes d'information, pages 240–270. Encyclopédie de l'informatique et des systèmes d'information, vuibert edition, 2006.

[61] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1) :11–33, January 2004.

[62] K. B. Rakesh. *Fault-Tolerant Real-Time Scheduling for Multiprocessor Systems*. Lambert Academic Publishing, 2011.

[63] Y. CHALLAL. Tolérance aux pannes dans les rcsf, November 2008.

[64] Axel Krings. Fault tolerant system design, 2007.

[65] A. Meixner, M. E. Bauer, and D. Sorin. Argus : Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[66] J. Weaver, C.and Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. *SIGARCH Comput. Archit. News*, 32(2) :264–, March 2004.

[67] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the ibm power6 processor. *IBM J. Res. Dev.*, 52(3) :275–284, May 2008.

[68] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 177–, Washington, DC, USA, 2004. IEEE Computer Society.

[69] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2007.

[70] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems*. PhD thesis, University of Pittsburgh, 1996.

[71] L. Mancini. Modular redundancy in a message passing system. *IEEE Trans. Softw. Eng.*, 12(1) :79–86, January 1986.

[72] C. M. Krishna. *Real-Time Systems*. McGraw-Hill Higher Education, 1st edition, 1996.

[73] Hussain Al-Asaad and Alireza Sarvi. Fault tolerance for multiprocessor systems via time redundant task scheduling. In *VLSI*, 2003.

[74] M. Maode and H. Babak. A fault-tolerant strategy for real-time task scheduling on multiprocessor system. In *Parallel Architectures, Algorithms, and Networks, 1996. Proceedings., Second International Symposium on*, pages 544–546, Jun 1996.

[75] P. Agrawal. Fault tolerance in multiprocessor systems without dedicated redundancy. *IEEE Trans. Comput.*, 37(3) :358–362, March 1988.

[76] A. L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Transactions on Software Engineering*, SE-12(11) :1089–1095, Nov 1986.

[77] C. M. Krishna and K. G. Shin. Tutorial : Hard real-time systems. chapter On Scheduling Tasks with a Quick Recovery from Failure, pages 531–538. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[78] K.G Skin and Y.H Lee. Fault tolerance in real-time systems. pages 28–34. IEEE Real-Time Systems Newletter, 1991.

[79] Y. Oh and S. H. Son. An algorithm for real-time fault-tolerant scheduling in multiprocessor systems. In *Real-Time Systems, 1992. Proceedings., Fourth Euromicro workshop on*, pages 190–195, June 1992.

[80] A. A. Bertossi and L. V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Syst.*, 7(3) :229–245, November 1994.

[81] B. Mirle and A. M. K. Cheng. Simulation of fault-tolerant scheduling on real-time multiprocessor systems using primary backup overloading simulation of fault-tolerant scheduling on real-time multiprocessor systems using primary backup overloading. 2006.

[82] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(3) :272–284, March 1997.

[83] Y. Oh and S. H. Son. Scheduling real-time tasks for dependability. *Journal of the Operational Research Society*, 48(6) :629–639, 1997.

[84] C. M. Krishna. Fault-tolerant scheduling in homogeneous real-time systems. *ACM Comput. Surv.*, 46(4) :48 :1–48 :34, March 2014.

[85] A. Burns, R.I. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of Eighth Euromicro Workshop on Real-Time Systems, L'Aquila, Italy*, pages 29–33, 1996.

[86] M. Pandya and M. Malek. Minimum achievable utilization for fault-tolerant processing of periodic tasks. *IEEE Trans. Comput.*, 47(10) :1102–1112, October 1998.

[87] F. Liberato, R. Melhem, and D. Mossé. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Trans. Comput.*, 49(9) :906–914, September 2000.

[88] V. Berten, J. Goossens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8–, April 2006.

[89] R. M. Pathan and J. Jonsson. Ftgs : Fault-tolerant fixed-priority scheduling on multiprocessors. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1164–1175, Nov 2011.

[90] H. Peng and F. Yang. Fault tolerant global scheduling for multiprocessor hard real-time systems. In *Proceedings of International Conference on Information Sciences, Machinery and Energy*, pages 1588–1596. Atlantic Press, 2015.

[91] F. Liberato, S. Lauzac, R. Melhem, and D. Mosse. Fault tolerant real-time global scheduling on multiprocessors. In *In Proc. of The 10 th IEEE Euromicro Workshop in Real-Time Systems*, 1999.

[92] S. Balaji, L. Jenkins, L. M. Patnaik, and P. S. Goel. Workload redistribution for fault-tolerance in a hard real-time distributed computing system. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 366–373, June 1989.

[93] X. Qin, H. Jiang, and D. R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. In *Proceedings of the 2002 International Conference on Parallel Processing*, ICPP '02, pages 360–, Washington, DC, USA, 2002. IEEE Computer Society.

[94] A. Christy and T. Gopalakrishnan. Fault tolerant real-time systems. In *International Conference on Managing Next Generation Software Application*, pages 177–180, 2008.

[95] G. Manimaran and C. S. R. Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11) :1137–1152, Nov 1998.

[96] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerant scheduling on a hard real-time multiprocessor system. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 775–782, Apr 1994.

[97] A. A. Bertossi, L. V. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 10(9) :934–945, September 1999.

[98] H. Zongfen and Q. Xiao. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *Journal of Huazhong University of Science and Technology*, pages 12–14, 1999.

[99] R. Al-Omari, Arun K. Somani, and G. Manimaran. A new fault-tolerant technique for improving the schedulability in multiprocessor real-time systems. In *Proceedings of the*

15th International Parallel &Amp ; Distributed Processing Symposium, IPDPS '01, pages 32–, Washington, DC, USA, 2001. IEEE Computer Society.

[100] R.M. KADRI. Tolérance de panne dans un circuit multi-cœur, June 5 2014. WO Patent App. PCT/US2012/067,085.

[101] K. M. Chandy. A survey of analytic models of rollback and recovery stratergies. *Computer*, 8(5) :40–47, May 1975.

[102] T.-H. Lin and K. G. Shin. Damage assessment for optimal rollback recovery. *IEEE Trans. Comput.*, 47(5) :603–613, May 1998.

[103] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, 46(8) :942–947, August 1997.

[104] K. Ahn, J. Kim, and S. Hong. Fault-tolerant real-time scheduling using passive replicas. In *Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, PRFTS '97, pages 98–, Washington, DC, USA, 1997. IEEE Computer Society.

[105] N. A Speirs and P. A. Barrett. Using passive replicates in delta-4 to provide dependable distributed computing. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 184–190, June 1989.

[106] R.S. Yadav. *High Performance Fault-Tolerant Real-Time Scheduling Algorithms*. PhD thesis, Electronics and Computer Engg Deptt, IIT Roorkee, India, 2003.

[107] A. Lotfi, A. Bayat, and S. Safari. Architectural vulnerability aware checkpoint placement in a multicore processor. In *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*, pages 118–120, June 2012.

[108] Risat Mahmud Pathan. *Scheduling Algorithms For Fault-Tolerant Real-Time Systems*. PhD thesis, Chalmers University of Technology Sweden, 2010.

[109] P. M. Hettiarachchi, N. Fisher, M. Ahmed, L. Y. Wang, S. Wang, and W. Shi. A design and analysis framework for thermal-resilient hard real-time systems. *ACM Trans. Embed. Comput. Syst.*, 13(5s) :146 :1–146 :25, July 2014.

[110] Younès Chandarli. *Gestion de l'Energie Renouvelable et Ordonnancement Temps-Réel dans les Systèmes Embarqués*. PhD thesis, Université Paris-Est, 2014.

[111] N. Subramanian and L. Chung. Architecture-driven embedded systems adaptation for supporting vocabulary evolution. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 144–153, 2000.

[112] N. Subramanian and L. Chung. Software architecture adaptability : An nfr approach. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 52–61, New York, NY, USA, 2001. ACM.

[113] M.M. Lehman and J.F. Ramil. Towards a theory of software evolution - and its practical impact (working paper). In *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov*, pages 2–11. Press, 2000.

[114] A. Burns. System mode changes-general and criticality-based.

[115] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. Technical report, Amherst, MA, USA, 1989.

[116] J. Real and A. Crespo. Mode change protocols for real-time systems : A survey and a new proposal. *Real-Time Syst.*, 26(2) :161–197, March 2004.

[117] P. Pedro and A. Burns. 10th euromicro workshop on real-time systems'98. In *10th Euromicro Workshop on Real-Time Systems'98*, pages 172–179, Dec 1988.

[118] T. Kloda. *Conditions d'Ordonnançabilité pour un Langage Dirigé par le Temps.* PhD thesis, l'Université de Toulouse, 2015.

[119] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, 1996.

[120] J.V. Real Sáez. *Protocolos de Cambio de Modo Para Sistemas de Tiempo Real.* PhD thesis, Universitat Politècnica de València, 2000.

[121] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, Dec 1989.

[122] Paulo Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems.* PhD thesis, University of York, 1999.

[123] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority preemptively scheduled systems. In *Real-Time Systems Symposium, 1992*, pages 100–109, Dec 1992.

[124] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4) :301–324, October 1990.

[125] B. Andersson. Uniprocessor edf scheduling with mode change. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 572–577, Berlin, Heidelberg, 2008. Springer-Verlag.

[126] V. Nélis. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems.* PhD thesis, U.L.B, 2011.

[127] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 151–160, July 2009.

[128] V. Nelis, B. Andersson, J. Marinho, and S. M. Petters. Global-edf scheduling of multi-mode real-time systems considering mode independent tasks. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 205–214, July 2011.

[129] P. M. Yomsi, V. Nélis, and J. Goossens. Scheduling multi-mode real-time systems upon uniform multiprocessor platforms, 2010.

[130] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

[131] A. Burns and I. D. Robert. Mixed criticality systems - a review, 2016.

[132] P. Graydon and I. Bate. *Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling*, pages 19–24. 2013.

[133] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. How realistic is the mixed-criticality real-time system model ? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 139–148, New York, NY, USA, 2015. ACM.

[134] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch. Mixed-criticality embedded systems – a balance ensuring partitioning and performance. In *Proceedings of the 2015 Euromicro Conference on Digital System Design*, DSD '15, pages 453–461, Washington, DC, USA, 2015. IEEE Computer Society.

[135] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 13–22, Washington, DC, USA, 2010. IEEE Computer Society.

[136] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 147–155, Washington, DC, USA, 2008. IEEE Computer Society.

[137] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8 :284–292, 1993.

[138] N. C. Audsley. On priority asignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1) :39–44, May 2001.

[139] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Syst.*, 46(3) :305–331, December 2010.

[140] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8) :1140–1152, Aug 2012.

[141] Z. Guo and S. Baruah. The concurrent consideration of uncertainty in wcets and processor speeds in mixed-criticality systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 247–256, New York, NY, USA, 2015. ACM.

[142] C. Gu, N. Guan, Q. Deng, and W. Yi. Improving ocbp-based scheduling for mixed-criticality sporadic task systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 247–256, Aug 2013.

[143] H. Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 183–192, Nov 2010.

[144] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 253–262, New York, NY, USA, 2011. ACM.

[145] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 93–102, July 2013.

[146] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler on single and multi-processor platforms. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 684–687, Aug 2015.

[147] J.H. Anderson, S.K. Baruah, and B.B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality :Roadmap to Evolving UAV Certification*, 2009.

[148] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1864–1871, Washington, DC, USA, 2010. IEEE Computer Society.

[149] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Real-Time Systems Symposium, 2015 IEEE*, pages 305–316, Dec 2015.

[150] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[151] Namhoon K., B.C. Ward, C. Micaiah, C.Y. Fu, J.H. Anderson, and F.D. Smith. *Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning.* Institute of Electrical and Electronics Engineers Inc., 4 2016.

[152] M. Bommert. Schedule-aware distributed of parallel load in a mixed criticality environment. In *Proceedings of JRWRTC, RTNS*, pages 21—-24, 2013.

[153] L. Sigrist, G. Giannopoulou, P. Huang, A. Gomez, and L. Thiele. Mixed-criticality runtime mechanisms and evaluation on multicores. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 194–206, April 2015.

[154] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Object/Component/Service-Oriented Real-Time*

Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on, pages 1–10, March 2011.

[155] D. Tamas-Selicean and P. Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 282–283, Dec 2011.

[156] O. R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, TRUSTCOM '11, pages 1051–1059, Washington, DC, USA, 2011. IEEE Computer Society.

[157] P. Rodriguez, L. George, Y. Abdeddaïm, and J. Goossens. Multi-criteria evaluation of partitioned edf-vd for mixed-criticality systems upon identical processors. In *Proc. 1st Workshop on Mixed Criticality Systems*, pages 49–54, 2013.

[158] K. Bletsas and S. M. Petters. Using nps-f for mixed-criticality multicore systems. *SIGBED Rev.*, 10(2) :36–36, July 2013.

[159] R. Gratia, T. Robert, and L. Pautet. Adaptation of run to mixed-criticality systems. In *8th Junior Researcher Workshop on Real-Time Computing, RTNS,*, 2014.

[160] R. Gratia, T. Robert, and L. Pautet. Scheduling of mixed-criticality systems with run. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2015.

[161] P. Ekberg, M. Stigge, N. Guan, and W. Yi. State-based mode switching with applications to mixed-criticality systems. In *Proceedings of WMC, RTSS*, pages 61–66, 2013.

[162] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Syst.*, 50(1) :48–86, January 2014.

[163] P. Ekberg and W. Yi. Schedulability analysis of a graph-based task model for mixed-criticality systems. *Real-Time Syst.*, 52(1) :1–37, January 2016.

[164] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed-criticality systems. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 34–43, Washington, DC, USA, 2011. IEEE Computer Society.

[165] H. Su, D. Zhu, and D. Mossé. Scheduling algorithms for elastic mixed-criticality tasks in multicore systems. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 352–357, Aug 2013.

[166] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 147–152, March 2013.

[167] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed-criticality system. In *Proc. WMC, RTSS*, pages 43–48, 2013.

[168] A. Burns and S. Baruah. Towards a more practical model for mixed-criticality systems. In *Proc. WMC, RTSS*, pages 1–6, 2013.

[169] S. Baruah and A. Burns. Implementing mixed-criticality systems in Ada. In A. Romanovsky and T. Vardanega, editors, *Proceedings of Reliable Software Technology - Ada Europe 2011*, volume LNCS 6652, pages 174–188. Springer, 2011.

[170] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 131 :1–131 :6, New York, NY, USA, 2014. ACM.

[171] A. Choquet-Geniet and G. Largeteau-Skapin. Resource minimization for multiprocessor real-time applications. In *Proceedings of the 6th international Workshop on Verification and Evaluation of Computer and Communication Systems*, pages 3–12, Paris, 2012.

[172] E. Chantery and Y. Pencole. Modélisation et intégration du diagnostic actif dans une architecture embarquée. *Journal Européen des Systèmes Automatisées*, MSR 2009 :789–803, Nov 2009.

[173] A. Bondavalli, S. Chiaradonna, and F. Di Giandomenico. Efficient fault tolerance : an approach to deal with transient faults in multiprocessor architectures. In *Proceedings of 1994 International Conference on Parallel and Distributed Systems*, pages 354–359, Dec 1994.

[174] F. P. Preparata, G. Metze, and R. T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16(6) :848–854, Dec 1967.

[175] J. Goossens and C. Macq. Limitation of the hyper-period in real-time periodic task set generation. In *In Proceedings of the RTS Embedded System (RTS'01*, pages 133–147, 2001.

[176] A. Choquet-Geniet and S. MALO. Scheduling an Aperiodic Flow within a Real-Time System using Fairness Properties. *REVUE AFRICAINE DE LA RECHERCHEEN INFORMATIQUEET MATHÉMATIQUES APPLIQUÉES*, Volume 18, 2014, aug 2016.

[177] J. Lin, A.M.K. Cheng, D. Steel, and M.Y.-C. Wu. Scheduling mixed-criticality real-time tasks with fault tolerance. In *Proc.2nd Workshop on Mixed Criticality Systems (WMC), RTSS,)*, pages 39–44, 2014.

[178] R. M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Syst.*, 50(4) :509–547, July 2014.

[179] A . Thekkilakattl, A. Burns, R. Dobrin, and S. Punnekkat. Mixed criticality systems : Beyond transient faults. In *Proc. 3rd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 18–23, 2015.

[180] S. Swaminathan and G. Manimaran. A reliability-aware value-based scheduler for dynamic multiprocessor real-time systems. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 39–, Washington, DC, USA, 2002. IEEE Computer Society.

[181] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 90–99, Dec 1995.

[182] S. A. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 270–277, 1999.

[183] L. R. Welch and S. Brandt. Toward a realization of the value of benefit in real-time systems. In *Proceedings of the 15th International Parallel &Amp ; Distributed Processing Symposium*, IPDPS '01, pages 93–, Washington, DC, USA, 2001. IEEE Computer Society.

[184] S. Swaminathan. *Value-Based Scheduling in Real-Time Systems*. PhD thesis, lowa State University, 2002.

[185] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *J. Syst. Archit.*, 46(4) :305–325, February 2000.

[186] F. Wang, K. Ramamritham, and J. A. Stankovic. Determining redundancy levels for fault tolerant real-time systems. *IEEE Trans. Comput.*, 44(2) :292–301, February 1995.

[187] J. R. Haritsa, M. J. Carey, and M. Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2) :117–152, April 1993.

RESUME

Cette thèse propose des techniques de construction d'une séquence d'ordonnancement valide par un algorithme équitable pour des systèmes temps-réel multicoeurs subissant des pannes processeurs permanentes. Selon la nature des tâches les concepteurs peuvent allouer ou pas du temps supplémentaire pour récupérer l'exécution perdue. Dans un premier temps, nous traitons le cas de la défaillance d'un seul cœur. Nous montrons alors que s'il n'y a pas d'allocation de temps supplémentaire, prévoir un cœur de plus que le minimum nécessaire permet de construire une séquence valide : c'est la Technique de la Redondance Matérielle Limitée. Toutefois, si une telle allocation s'impose, nous proposons trois techniques : la Technique des Sous-tâches de Substitution qui augmente le WCET des tâches afin de prévoir du temps additionnel en remplacement du temps perdu, la Technique Contraindre puis Relâcher qui crée une marge de temps entre le délai critique et la période d'une tâche pour rattraper l'exécution perdue et la Technique du Flux Apériodique (TFA) qui réordonnance l'exécution perdue dans les unités de temps creux équitablement réparties. Ensuite, l'utilisation conjointe de ces techniques est envisagée en fonction de la nature des tâches impactées. Enfin, le cas de la défaillance de plusieurs cœurs est étudié. Pour ajuster la charge du système au nombre de cœurs fonctionnels, deux approches sont proposées : le changement de mode de criticité qui modifie les paramètres temporels de certaines tâches et l'abandon de certaines tâches en fonction de leur importance dans le système.

Mot clés : *ordonnancement temps-réel, algorithmes équitables, tolérance aux pannes, changement de mode, système à criticité multiple.*

ABSTRACT

This thesis proposes several techniques to build a valid schedule with a Pfair algorithm for multicore real-time systems despite permanent processor failures. Depending on the nature of the tasks, additional time may be allocated or not to recover the lost execution. First, we consider a single core failure. We then show that if no additional time is allocated, the use of a single more core than the required minimum provides a valid schedule : it is the Limited Hardware Redundancy Technique. However, if full recovery is mandatory, we propose three techniques : the Substitute Subtasks Technique which increases the WCET to provide additionnal time which can be used to recover the lost time, the Constrain and Release Technique which creates a time margin between each task's deadline and the following period which can be used to recover the lost execution and the Aperiodic Flow Technique which reschedules the lost execution within the idle time units. Then, these techniques are mixed to adapt the scheduling behaviour to the nature of the impacted tasks. Finally, the case of the failure of several cores is studied. To adapt the system load to the number of remaining functionnal cores we use the criticality mode change which modifies the temporal parameters of some tasks or we discard some tasks according to their importance.

Keywords : *Real-Time Scheduling, Pfair Algorithms, Failure Tolerance, Mode changes, Mixed Criticality Systems.*