Le modèle BGW pour les systèmes temps réel surchargés

Mohamed Ould Sass

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Mohamed OULD SASS

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Automatique et Informatique Appliquée, section CNU 61
Unité de recherche : Institut de Recherche en Communication et Cybernétique de Nantes (IRCCyN)

Soutenue le 24 septembre 2015
Thèse n° : ED 503-XXX

Le modèle BGW pour les systèmes temps réel surchargés

Ordonnancement monoprocesseur

JURY

Rapporteurs :	**M^me^ Samia BOUZEFRANE**, Maître de Conférences HDR, CNAM Paris, rapporteure
	M. Daniel SIMON, Chargé de Recherches HDR, INRIA-LIRMM Montpellier, rapporteur
Examinateurs :	**M. Laurent GEORGE**, Professeur à l'ESIEE, LIGM-Université de Paris Est
	M. Olivier SENAME, Professeur des universités, Institut Polytechnique de Grenoble
	M^me^ Rosa ABBOU, Maître de Conférences, Université de Nantes
Directrice de thèse :	**M^me^ Maryline CHETTO**, Professeure des universités, Université de Nantes

Remerciements

Cette thèse a été préparée dans l'équipe Systèmes Temps Réel au sein de l'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN). Je tiens à remercier Monsieur Michel Malabre d'avoir accepté et facilité mon intégration dans ce laboratoire qu'il dirige.

Je veux exprimer aussi ma profonde et sincère reconnaissance à Madame Maryline Chetto, Professeure à l'Université de Nantes, qui m'a fait l'honneur de diriger ce travail de thèse. En m'accordant sa confiance, elle m'a permis d'acquérir des compétences en RD temps réel et de m'épanouir professionnellement durant ces trois dernières années. Je la remercie aussi pour son attention et les conseils qu'elle m'a donnés tout au long de cette préparation.

J'adresse mes remerciements aux membres du jury pour le temps et l'attention qu'ils ont consacré à la lecture et à la critique de ce travail.

Que Madame Samia Bouzefrane, Maître de Conférences HDR, CNAM Paris, trouve ici ma reconnaissance et mes remerciements d'avoir accepté d'être rapporteur de mon mémoire de thèse.

Ma reconnaissance et mes remerciements vont également à Monsieur Daniel Simon, Chargé de Recherches HDR, LIRMM Montpellier, qui a accepté, lui aussi, d'être rapporteur de mon mémoire de thèse.

Je souhaite exprimer ma gratitude à Monsieur Laurent George, Professeur ESIEE, LIGM, Université de Paris Est, pour avoir accepté de faire partie de mon jury.

L'expression de ma gratitude va également à Monsieur Olivier Sename, Professeur, Institut Polytechnique de Grenoble, pour avoir accepté de participer à ce jury de thèse.

Je remercie également Madame Rosa Abbou, Maître de Conférences, Université de Nantes, d'avoir accepté de prendre part à ce jury de thèse.

Mes remerciements vont à toute ma famille pour leurs encouragements, leur soutien et à tous ceux qui ont contribué, même de très loin, à l'aboutissement de cette thèse.

Table des matières

Liste des tableaux

Table des figures

Introduction générale

Contexte

Un nombre assez considérable de travaux ont porté depuis plusieurs décennies sur l'ordonnancement temps réel. Toutefois, la grande majorité de ceux-ci ont trait à des applications temps réel dures. Ces applications se caractérisent par la nécessité de respecter cent pour cent des contraintes temporelles qui s'expriment en termes d'échéances c'est à dire de fins d'exécution au plus tard des tâches.

Cependant, l'informatique temps réel concerne maintenant de très nombreux domaines d'application caractérisés par des contraintes moins strictes que celles trouvées dans les applications critiques. On parle alors de systèmes temps réel fermes car on accepte avec une tolérance bien spécifiée que certaines échéances soient violées. On peut alors mesurer la performance d'un ordonnanceur par rapport à un autre en comparant la Qualité de Service (QdS) effective mesurée par le ratio d'échéances satisfaites.

Dans un système qualifié de sous-chargé qui n'utilise pas toute sa capacité de traitement, on peut, par un ordonnancement optimal, atteindre la QdS maximale. Dans un système surchargé, il nous faut contrôler la dégradation de la QdS sans compromettre le bon fonctionnement du système et sans mettre en danger son intégrité.

Dans ce contexte, il devient primordial pour l'utilisateur final de disposer d'un modèle de tâche capable de spécifier son comportement minimaliste attendu dans la situation de pire cas de surcharge. L'application doit en effet, en toutes circonstances, pouvoir continuer à fonctionner dans un mode dégradé le plus judicieux possible et en conformité avec les spécifications applicatives.

La problématique étudiée dans cette thèse concerne donc des systèmes temps réel fermes soumis à des surcharges de traitement dont nous cherchons à optimiser la Qualité de Service.

Ce travail de thèse fait suite à des travaux antérieurs effectués au sein de l'équipe STR (Systèmes Temps Réel) de l'IRCCyN et qui portaient séparément sur la tolérance aux fautes temporelles et la gestion de surcharge.

Plan général de la thèse

La première partie est consacrée à l'état de l'art. Nous proposons dans le premier chapitre de la thèse, un état de l'art sur l'informatique temps réel embarquée et dans le deuxième chapitre un état de l'art sur la gestion des fautes temporelles (conduisant au non respect des échéances) dans les systèmes surchargés.

Notre première contribution est décrite dans **la seconde partie**. Celle-ci concerne la proposition d'un nouveau modèle de tâches appelé modèle BGW (Black, Grey, White). Il combine deux modèles existants connus sous les noms respectifs de *modèle du mécanisme à échéance* et *modèle skip-over*.

Dans la troisième partie, composée des chapitres 4, 5 et 6, nous proposons des stratégies d'ordonnancement pour des tâches BGW. Chacune de ces stratégies d'ordonnancement possède des particularités (simplicité d'implémentation,...) et remplit des objectifs bien déterminés (par exemple, compromis entre QdS et équité de service).
Concernant l'évaluation de performances de ces stratégies, en particulier au regard de la QdS et des overheads de mise en oeuvre, plusieurs études de simulations, implémentées en langage C, ont été réalisées. Nous les décrivons dans cette partie où les résultats y sont commentés. Chaque chapitre correspond à une famille d'ordonnanceurs plus ou moins sophistiqués.

Enfin, le manuscrit se termine par une **conclusion générale** où nous synthétisons nos résultats. Nous y décrivons plusieurs axes de poursuite de ce travail dont l'ordonnancement en environnement multiprocesseur et l'ordonnancement régulé.

Et comme perspectives de recherche que ce travail peut ouvrir, nous proposons d'autres approches pour la gestion de surcharge dans un système temps réel monoprocesseur. En se basant sur le modèle BGW, nous suggérons d'utiliser des ordonnancements par régulation automatique de la QdS.

I

État de l’art

1

Fondamentaux de l'informatique temps réel

1.1 Présentation du chapitre

Dans ce chapitre, nous allons présenter les notions de base relatives à toute étude de conception ou d'analyse d'un système temps réel. Nous donnerons d'abord des définitions et décrirons l'architecture d'un système temps réel. Puis, nous nous intéresserons à la caractérisation et la modélisation des tâches temps réel. L'ordonnanceur étant le coeur d'un système d'exploitation temps réel, nous présenterons les principaux algorithmes d'ordonnancement. Ensuite, nous décrirons les méthodes d'analyse d'ordonnançabilité généralement utilisées et associées aux ordonnanceurs les plus répandus, à savoir l'ordonnanceur à priorités fixes, Deadline Monotonic et l'ordonnanceur à priorités dynamiques, Earliest Deadline First.

1.2 Systèmes temps réel

1.2.1 Définitions importantes

Un *système temps réel* est un système en réaction avec un environnement externe ou un procédé par le biais d'une instrumentation. Celle-ci se compose d'un ensemble de capteurs et d'actionneurs. Les capteurs délivrent des mesures représentatives de l'état de l'environnement ou/et du procédé. Les actionneurs reçoivent des commandes calculées par le système informatique (système de contrôle) et modifient donc l'état de l'environnement ou du procédé. Un système temps réel constitue donc un système en boucle fermée, ce qui impose au système informatique de contrôle d'assujettir son fonctionnement aux dynamiques imposées par l'environnement contrôlé.
Les systèmes temps réel sont classés en fonction de la criticité de la contrainte de temps c'est à dire de la gravité de la situation qui résulte du manquement au respect de la contrainte temporelle. Dans la très grande majorité des cas, la contrainte temporelle prend la forme d'une date critique appelée *échéance* avant laquelle un programme (appelé *tâche*) doit avoir terminé son exécution. Il s'agit par exemple de l'échéance associée à la réception d'une commande sur un actionneur. Par la suite, nous appellerons *faute temporelle* le non respect d'une échéance.
Ainsi, on dit qu'un système temps réel est *dur* (en anglais, hard) lorsque le non-respect d'une échéance de l'une de ses tâches, peut engendrer des catastrophes humaines ou matérielles. Ce type de système se rencontre dans de nombreux domaines incluant le contrôle d'une centrale nucléaire, le pilotage d'un avion, le contrôle/commande d'un robot, etc.

Les applications temps réel ne relèvent pas toutes du contrôle/commande de processus. Ainsi le secteur du multimédia ou de la téléphonie mobile sont aussi grands utilisateurs de la technologie temps réel. Prenons l'exemple d'une lecture de film en streaming à partir du réseau internet. L'affichage de chaque image dans l'idéal devrait se faire à cadence régulière et sans aucun retard. Or, nous pouvons accepter que l'affichage d'une image se fasse de temps à autre avec la tolérance de quelques millisecondes de retard. On parle alors de système temps réel *souple* (en anglais, soft). En effet, le retard de la réponse d'une tâche par rapport à son échéance n'engendre pas des conséquences graves mais juste une dégradation de la *Qualité de Service* (QdS).

La troisième classe de systèmes est celle des *systèmes temps réel fermes* pour lesquels nous acceptons que de temps à autres, certaines tâches d'application ne s'exécutent pas entièrement et par conséquent ne produisent aucun résultat. La QdS sera dégradée sans que cela ne remette en question la poursuite de l'application. Il s'agira pour le concepteur de faire en sorte que l'on tente à tout instant de maximiser cette QdS représentée par le pourcentage d'échéances effectivement respectées. Dans le cas d'une application multimédia, on pourra ainsi tolérer que certaines images en provenance du réseau soient détruites et donc non affichées lorsque le processeur est tellement chargé qu'il n'a matériellement pas le temps de traiter la totalité des images qui lui sont soumises à traitement. Dans une application de contrôle/commande, on tolérera que de temps à autres, aucune commande ne soit délivrée, ce qui suppose de connaître a priori la tolérance de perte de sorte que le processus contrôlé reste dans un état stable.

Dans cette thèse, nous nous intéresserons plus particulièrement aux systèmes temps réel fermes.

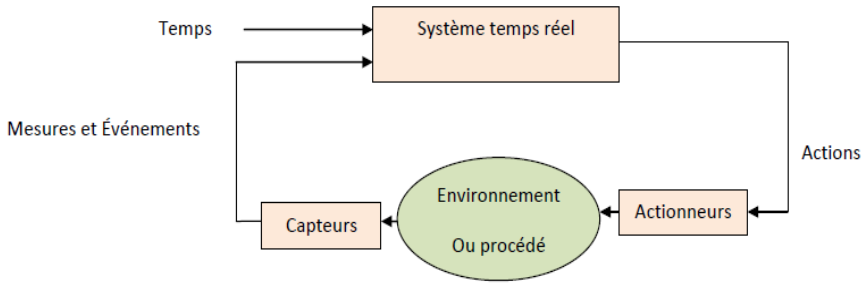


FIGURE 1.1 – Un système temps réel et son environnement.

1.2.2 Domaines d'applications temps réel

Aujourd'hui, l'informatique temps réel joue un rôle primordial dans notre société où les objets de toutes natures sont connectés pour s'échanger des informations. On parle ainsi de *systèmes cyber-physiques*, de l'*Internet des Objets*. Ces objets peuvent être des systèmes temps réel plus ou moins complexes devant acquérir des données, traiter des données et expédier des données dans des temps contraints. Ainsi toute notre société se trouve concernée par les systèmes temps réel qu'il s'agisse de produits grand public que de systèmes plus sophistiqués. C'est en particulier le cas des processus complexes de production, des systèmes de commande de vol, des automobiles, de l'acquisition et du suivi de données environnementales, du contrôle de réacteurs chimiques et des centrales nucléaires, des systèmes de télé-médecine, de l'automatisation industrielle, de certains appareils électroniques grand public, etc.

La grande majorité des systèmes temps réel sont des *systèmes embarqués*. Ces derniers se caractérisent par une liste de contraintes propres découlant de l'environnement physique dans lequel ils sont placés. Ces contraintes peuvent par exemple avoir une incidence sur la taille et le poids du système lorsque résidant sur

un petit matériel mobile. Elles peuvent aussi s'exprimer en termes de consommation énergétique car disposant d'une très petite batterie ce qui limite la puissance consommée. Ces contraintes environnementales exprimées en température, pression, vibrations peuvent aussi avoir un impact sur le choix des composants électroniques comme c'est le cas pour les applications militaires.
En résumé, les systèmes embarqués sont très différents les uns des autres selon le domaine d'application visé. Cela va des petits appareils portables (par exemple, les téléphones mobiles, caméras, navigateurs, appareils ECG, Holter, jouets intelligents, etc.) à de grands systèmes (par exemple, robots industriels, voitures, avions, etc.).

1.2.3 Architecture d'un système temps réel

L'*architecture matérielle* sur laquelle un systèmes temps réel exécute ses programmes informatiques, désigne l'ensemble des ressources physiques nécessaires d'une part au traitement des données et d'autre part aux interactions entre ce système informatique et l'environnement avec lequel celui-ci interagit. Ces ressources sont les processeurs, les unités d'entrées/sorties, les mémoires, les réseaux, etc.

En fonction du nombre de processeurs et de l'utilisation éventuelle ou non d'un réseau, on distingue trois grandes catégories d'architectures :

- *Architecture monoprocesseur* : tous les programmes (tâches) de l'application sont exécutés sur un unique processeur partagé.
- *Architecture multiprocesseur centralisée* : Les tâches s'exécutent sur plusieurs processeurs le plus souvent identiques qui se partagent une mémoire centrale commune, chaque processeur disposant en plus de sa mémoire propre.
- *Architecture multiprocesseur distribuée* : Les processeurs ne se partagent aucune mémoire et les processeurs s'échangent des informations via un bus ou un réseau. En général, les tâches sont assignées statiquement à chacun des processeurs. Éventuellement, elles ne migreront qu'en situation exceptionnelle telle que la panne d'un processeur. On parlera alors de reconfiguration dynamique des tâches suite à la modification de l'architecture matérielle. Pour une application de contrôle/commande de procédés industriels, une architecture distribuée pourra utiliser un réseau de terrain tel que CAN (Controller Area Network).

Ce travail de thèse portera sur une architecture matérielle monoprocesseur.

1.2.4 Logiciels temps réel

L'architecture logicielle d'un système informatique temps réel se compose essentiellement de deux composants :

- un exécutif temps réel et
- une application.

Pour être exploitées, les ressources d'une architecture matérielle nécessitent la mise en oeuvre d'un logiciel système appelé *Système d'exploitation temps réel* (en anglais, RTOS pour Real Time Opérating System) appelé aussi *exécutif temps réel*. Le coeur de ce logiciel contient les mécanismes indispensables à son fonctionnement et à la gestion des ressources matérielles. Il est appelé *noyau temps réel*.
L'exécutif a un rôle de centralisateur ou d'interface car il aiguille les événements (interruptions) reçus de l'environnement contrôlé vers le logiciel d'application qui se charge de traiter ces événements en conformité avec les spécifications. Parmi les systèmes d'exploitation les plus connus, on trouve des systèmes orientés à

utilisation commerciale comme par exemple, VRTX32, pSOS, OS9, VxWorks Chorus et des systèmes utilisés dans le domaine de la recherche comme MARS, Spring, ARTS, RK, TIMIX, MARUTI et HARTOS [21].

L'exécutif temps réel va donc non seulement gérer les ressources matérielles mais aussi gérer le logiciel d'application. Celui-ci se compose de programmes appelés *tâches*. Une tâche désigne le programme informatique qui réalise une fonction par exemple celle nécessaire au contrôle du procédé.
Les tâches font appel, lors de leur exécution, aux services proposés par l'exécutif temps réel et qui assurent principalement la synchronisation et la communication inter-tâches. Ces tâches nécessitent aussi des services liés à la gestion du temps leur permettant notamment de s'exécuter périodiquement. Elles nécessitent aussi de pouvoir démarrer leur exécution sur l'arrivée d'interruptions matérielles, d'où la présence dans l'exécutif de services de gestion d'événements.
Ainsi, à tout instant plusieurs tâches peuvent avoir reçu leur signal de réveil c'est à dire l'événement mentionnant à l'exécutif l'autorisation de les exécuter. Comme à un moment donné, plusieurs tâches se trouvent prêtes à s'exécuter, se pose un problème de concurrence pour l'accès au processeur.

L'*ordonnanceur* représente le composant central d'un système d'exploitation temps réel car il est en charge de sélectionner la tâche qui se verra autorisée à s'exécuter. La règle utilisée par l'ordonnanceur pour choisir la future *tâche active* (tâche en exécution) s'appelle *algorithme d'ordonnancement*.

Dans beaucoup de RTOS, on trouve des algorithmes d'ordonnancement cycliques et statiques, qui élaborent la description de l'activité du processeur (appelée *séquence d'ordonnancement*) de façon hors ligne. En d'autres termes, avant de lancer l'application, on définit la façon dont le temps processeur sera réparti sur les différentes tâches. Cependant, lorsqu' une plus grande souplesse est nécessaire, on doit faire appel à des techniques d'ordonnancement dites *en ligne*. Celles-ci construisent la séquence dynamiquement au fur et à mesure que les événements se produisent.

La grande partie des algorithmes d'ordonnancement en ligne utilisent la notion de *priorité*. Chaque tâche se voit affecter une priorité utilisée par l'ordonnanceur pour ordonner la liste des tâches prêtes à tout instant et ainsi sélectionner la plus prioritaire. On fait la distinction entre *priorités fixes* si assignées hors ligne et *priorités dynamiques* lorsqu'elles changent de valeurs au cours du temps.

En résumé, un RTOS conventionnel fournit, outre l'ordonnanceur, des services de base comme la communication inter-tâches, la synchronisation inter-tâches, la gestion de l'accès aux ressources partagées et la gestion des interruptions.

Dans cette thèse, nous utiliserons un modèle de tâches indépendantes.

1.3 Les tâches temps réel

1.3.1 Catégorisation des tâches

Dans un système temps réel, il existe essentiellement trois familles de tâches qui diffèrent selon la nature de l'événement qui les réveille : *tâches périodiques*, *tâches apériodiques* et *tâches sporadiques*.
Dans une application temps réel de contrôle/commande, on utilise des tâches périodiques pour exécuter de manière répétitive et cyclique les fonctions de traitement des données issues de capteurs et les fonctions de régulation et de suivi en boucle fermée. La période d'une tâche périodique découle directement de la vitesse d'évolution du procédé contrôlé c'est à dire de sa dynamique. Selon l'application, la période peut varier de quelques millisecondes à quelques minutes. Toute tâche périodique donne lieu à l'exécution d'une infinité de *jobs* appelés aussi *instances*. En général, on exige qu'une instance de tâche termine son exécution avant

que l'instance suivante de la même tâche ne soit réveillée.
On dira qu'une tâche est *sporadique* si elle est cyclique mais que sa période de répétition n'est pas constante. On connaît toutefois pour cette tâche l'intervalle de temps minimum séparant deux réveils successifs.
Les tâches apériodiques désignent des programmes qui démarrent leur exécution sur l'arrivée d'un événement non régulier et imprévisible et pour lequel il n'est pas possible d'associer un délai d'inter-arrivée. Typiquement, une tâche de traitement d'alarme est une tâche apériodique.

1.3.2 Spécifications d'une tâche périodique

Une tâche périodique sera notée τ_i. Liu et Layland sont à l'origine en 1973 des premiers travaux sur l'ordonnancement de tâches périodiques dans un système temps réel. τ_i est modélisée par quatre paramètres (r_i, C_i, T_i, D_i) définis comme suit :

- r_i : *Date de réveil de la première instance* de τ_i, qui correspond à la date de début d'exécution au plus tôt de τ_i.
- C_i : *Durée d'exécution* nécessaire au processeur pour exécuter toute instance de τ_i. En réalité, le temps d'exécution effectif d'une tâche varie entre une borne minimum $C_i \downarrow$ ou BCET (Best Case Execution Time) et une borne maximum $C_i \uparrow$ ou WCET (*Worst Case Execution Time*) c'est à dire $C_i \downarrow \leq C_i \leq C_i \uparrow$. Généralement, on étudie le comportement du système dans la situation pire cas c'est à dire avec $C_i = C_i \uparrow$.
- T_i : *Période de répétition* de τ_i qui est prête à être exécutée au début de chaque période.
- D_i : *Délai critique* ou *échéance relative* de τ_i. On a donc : $d_i = r_i + D_i$ où d_i désigne l'échéance absolue de la première instance de τ_i.

De ces paramètres temporels associés à chaque tâche, on en déduit des grandeurs caractéristiques de l'application temps réel :

- *le facteur d'utilisation du processeur* noté U_p. Cette grandeur comprise entre 0 et 1 indique le pourcentage de temps durant lequel le processeur se trouve occupé à exécuter des tâches de l'application. Ainsi, on dit que le processeur est *surchargé* si le taux d'utilisation est supérieur à 1. Autrement dit, une condition nécessaire de faisabilité dans le cas d'une application temps réel à contraintes dures est que le processeur soit *sous-chargé* c'est à dire que $U_p \leq 1$.

- *l'hyperpériode* notée H. Cette grandeur est égale au plus petit commun multiple des périodes des tâches de l'application, soit $H = PPCM(T_1, T_2, ...T_n)$.

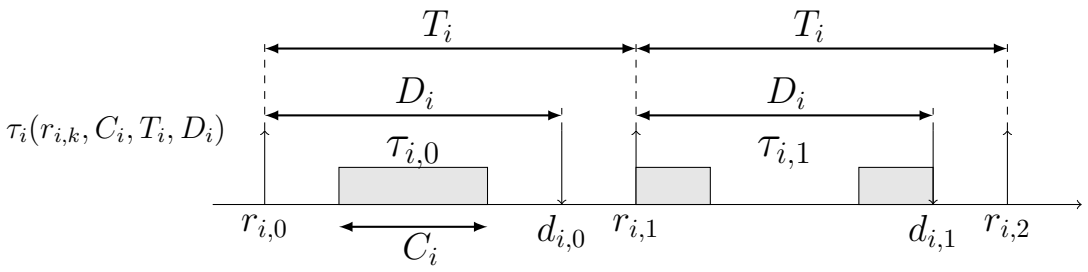


FIGURE 1.2 – Illustration du modèle d'une tâche périodique

On distingue plusieurs catégories de tâches périodiques :

- à *échéances contraintes* : si l'échéance relative est inférieure ou égale à la période ($D_i \leq T_i$).
- à *échéances implicites* (ou à échéance sur requêtes) si l'échéance relative et la période sont égales ($D_i = T_i$) ; c'est donc un cas particulier de tâches périodiques à échéances contraintes.
- à *échéances arbitraires* : s'il n'existe pas de contrainte entre l'échéance et la période.
- *concrètes* : si les dates de réveil initiales sont connues a priori.
- *synchrones* : si elles ont la même date de réveil initial ; dans le cas contraire, on dit qu'elles sont *asynchrones*.
- *critiques* : si les échéances doivent être impérativement respectées.
- *semi-critiques ou fermes* : si le non-respect de quelques échéances est toléré.
- *non critiques* : si elles n'ont pas d'échéances strictes à respecter.

Dans notre thèse, nous considérons une application composée d'un ensemble de tâches périodiques fermes, concrètes, synchrones à l'instant zéro et à échéances contraintes.

Nous renvoyons le lecteur vers des ouvrages tels que [62] [28] [21] pour y trouver une description exhaustive concernant les exécutifs et les tâches temps réel.

1.4 L'ordonnancement temps réel

Dans ce paragraphe, nous rappelons quelques notions importantes pour la suite de la thèse relatives à l'ordonnancement d'un ensemble de tâches temps réel.

1.4.1 Ordonnancement préemptif versus non préemptif

Un algorithme d'ordonnancement est dit *préemptif*, lorsque la tâche en exécution peut être interrompue à chaque moment pour donner le processeur à une autre tâche suivant une règle d'ordonnancement spécifiée. Selon un algorithme d'ordonnancement *non préemptif* (non interruptible), la tâche qui commence son exécution ne peut pas être interrompue jusqu'à ce qu'elle finisse. Toute tâche temps réel, qu'elle soit périodique, sporadique ou apériodique est caractérisée par une date appelée échéance avant laquelle la tâche doit terminer son exécution. Intuitivement, on comprend qu'un ordonnanceur non préemptif n'est pas adapté à ce contexte où l'exécution d'une tâche longue mais peu urgente engendrerait le manquement de l'échéance d'une tâche très urgente. C'est la raison pour laquelle dans la très grande majorité des études et des RTOS, l'ordonnanceur est préemptif.

1.4.2 Ordonnancement en ligne versus hors ligne

Un algorithme d'ordonnancement est dit *hors ligne*, lorsque la séquence d'ordonnance est construite avant le lancement de l'application. L'ordonnancement généré par cette méthode est enregistré dans une table puis exécuté par le séquenceur.
Cette approche statique est très efficace mais très rigide car elle suppose que tous les paramètres, y compris les dates de réveil, soient figées. Elle n'est donc pas adaptée pour effectuer une reconfiguration du logiciel et/ou du matériel en présence de pannes matérielles ou de fautes logicielles.

Un algorithme est dit *en ligne*, lorsque la décision d'ordonnancement se prend à chaque fois qu'une tâche se réveille, se bloque dans l'attente d'un événement ou d'une ressource occupée ou qu'une tâche se termine.
Le seul inconvénient d'un ordonnanceur en ligne par rapport à un ordonnanceur hors-ligne réside dans les surcoûts d'exécution appelés *overhead*. C'est la raison pour laquelle tout algorithme d'ordonnancement temps réel doit être évalué par sa *complexité algorithmique* qui donne une indication sur la quantité d'overhead temporel engendré à la mise en oeuvre.
Comme indiqué précédemment, l'énorme avantage d'un ordonnanceur en ligne tient dans sa grande flexibilité et son adaptabilité pour prendre en compte l'arrivée imprévisible de tâches, la variation des paramètres temporels des tâches comme leur durée d'exécution effective. Ceci est rendu possible car il effectue une construction progressive de la séquence d'ordonnancement et peut remettre en question une séquence d'ordonnancement à l'arrivée de tout nouvel événement.

Dans cette thèse, nous nous intéressons à des systèmes temps réel pouvant être sujets à des fautes et des surcharges de traitement imprévisibles. D'où la nécessité de se focaliser sur des ordonnanceurs préemptifs en ligne, assez flexibles pour s'adapter dynamiquement à ces situations.

1.4.3 Autres caractéristiques d'un ordonnanceur

Ordonnancement oisif versus non oisif : Un algorithme d'ordonnancement en ligne est dit *non oisif* ou *conservatif*, lorsqu'il ne laisse jamais le processeur inactif dans le cas où il y aurait au moins une tâche prête. Dans le cas contraire, il est dit *oisif* ou *non conservatif*. La très grande majorité des ordonnanceurs intégrés aux exécutifs temps réel sont non oisifs.
Ordonnancement clairvoyant versus non clairvoyant : Pour mesurer la performance d'un algorithme en ligne, on peut l'évaluer en la comparant à celle d'un ordonnanceur qui aurait à tout instant connaîssance du futur et pourrait ainsi prendre la meilleure décision d'ordonnancement. On parle alors d'*ordonnanceur clairvoyant*. Au contraire, un ordonnanceur qui prend ses décisions uniquement sur la base du présent ou du passé car ne connaissant pas l'avenir est dit *non clairvoyant*.

1.4.4 Définitions

Donnons quelques définitions fondamentales relatives à une séquence d'ordonnancement puis à l'ordonnançabilité.

Séquence : Une séquence donne la description de l'activité du processeur au cours du temps (diagramme de Gantt) résultant de la mise en oeuvre d'un algorithme d'ordonnancement sur un ensemble de tâches.

Séquence valide : Un algorithme d'ordonnancement délivre une séquence d'ordonnancement dite *valide* pour un ensemble de tâches donné si toutes les tâches respectent leurs contraintes temporelles exprimées en termes de respect d'échéances.

Période d'activité du processeur : Il s'agit d'un intervalle de temps durant lequel le processeur exécute des tâches sans discontinuité.

Période d'oisiveté ou de passivité du processeur : Il s'agit d'un intervalle de temps pendant lequel le processeur est continuement inactif. Cela correspond à un intervalle pendant lequel aucune tâche n'est prête (ordonnanceur non oisif) ou pendant lequel l'ordonnanceur choisit délibérément de n'exécuter aucune tâche (ordonnanceur oisif).

Ensemble de tâches ordonnançable : Un ensemble de tâches est dit *ordonnançable* s'il existe au moins

un algorithme capable de fournir une séquence valide pour cet ensemble.

Condition nécessaire d'ordonnançabilité : C'est la condition associée à un algorithme d'ordonnancement donné qui doit impérativement être vérifiée pour que cet algorithme produise une séquence valide sur un ensemble de tâches donné. Cette condition est aussi dite *condition de faisabilité* car elle porte sur la capacité du processeur à supporter l'exécution de l'ensemble des tâches.

Condition suffisante d'ordonnançabilité : C'est la condition associée à un algorithme d'ordonnancement donné qui, lorsqu'elle est satisfaite nous garantit que cet algorithme produit une séquence valide sur un ensemble de tâches donné.

Test exact d'ordonnançabilité : C'est une condition d'ordonnançabilité qui est à la fois nécessaire et suffisante.

Test d'admission (dit aussi d'acceptation ou de garantie) : Un événement peut conduire à demander l'exécution d'une tâche supplémentaire et ce sans qu'on sache initialement à quel moment interviendra cet événement. Si cette tâche doit s'exécuter dans le respect d'une échéance, se pose donc le problème de tester en ligne la faisabilité de la nouvelle application. Pour une réponse oui au test, il s'agira d'admettre la nouvelle tâche car n'induisant pas une surcharge de traitement et garantissant le respect des échéances. Pour une réponse non, il s'agira de répondre à la situation de surcharge par des décisions dont la nature dépend du caractère de l'application. Dans le cas d'une application à contraintes fermes, on devra par exemple décider des instances de tâches à supprimer (voir chapitre suivant) pour revenir à une situation de sous-charge et maintenir une Qualité de Service acceptable.

1.4.5 Performance d'un ordonnanceur

La performance d'un ordonnanceur se mesure différemment selon le type d'application dans lequel on l'utilise c'est à dire si l'on s'intéresse à une application temps réel à contraintes dures, fermes ou légères. Dans cette thèse, nous nous focalisons sur les applications temps réel fermes pour lesquelles la mesure de performance s'exprime par une ou plusieurs grandeurs de QdS.

Dans une application temps réel ferme où chaque tâche est caractérisée par une échéance portant sur sa fin d'exécution au plus tard, **la QdS rendue par l'ordonnanceur se mesure trés généralement par le nombre d'échéances respectées rapporté au nombre total d'échéances qui auraient dû être respectées idéalement.** La QdS s'exprime donc par un ratio compris entre 0 et 100.

Ordonnancement optimal (cas de contraintes fermes) : Dans le cadre d'une application temps réel à contraintes fermes, un algorithme d'ordonnancement est dit *optimal* s'il maximise la Qualité de Service exprimée par le ratio d'échéances satisfaites.

Ordonnancement optimal (cas de contraintes dures) : Dans le cadre d'une application à contraintes dures, on part de l'hypothèse que la totalité des échéances doivent être satisfaites. L'optimalité s'exprime donc différemment : On dira ainsi qu'un ordonnanceur est optimal si, chaque fois qu'il ne peut pas construire une séquence valide alors aucun autre ordonnanceur n'est capable d'en construire une.

1.4.6 Overheads d'un ordonnanceur

La performance d'un ordonnanceur a trait au taux de respect des contraintes temporelles. Plus grand est ce taux c'est à dire plus grande est la QdS, plus performant est l'ordonnanceur temps réel. Cependant cette seule mesure du comportement d'un ordonnanceur n'est pas suffisante. Nous ne devons pas oublier que

dans le cadre de notre étude, nous étudions des ordonnanceurs en ligne qui prennent leurs décisions dynamiquement au fur et à mesure que l'application se déroule. Il s'avère donc primordial aussi d'évaluer le temps requis pour la mise en oeuvre de l'ordonnanceur. Il s'agit en effet d'une fonction du système d'exploitation dont on doit minimiser les coûts d'exécution.

De plus, nous nous intéressons aux ordonnanceurs préemptifs. Chaque préemption de tâche engendre donc un délai rendu nécessaire pour la sauvegarde du contexte de la tâche préemptée et la restitution du contexte de la tâche activée. A performances égales, plus faible sera le nombre de préemptions, plus efficace sera l'ordonnanceur. L'ensemble de tous les surcôuts associés à la mise en oeuvre d'un ordonnanceur est appelé *overhead*.

En résumé, les overheads d'un ordonnanceur doivent faire l'objet d'une évaluation qui porte d'une part sur le temps d'exécution de l'algorithme d'ordonnancement (overhead d'exécution) et sur le taux de préemption (overhead de préemption).

Dans notre travail de simulation, outre la QdS, nous mesurerons les overheads des ordonnanceurs temps réel proposés.

1.5 Ordonnancement de tâches périodiques

Bon nombre d'applications temps réel ne comportent que des tâches strictement répétitives. Pour ordonnancer ces tâches périodiques, des algorithmes conduits par une priorité fixe ou dynamique ont été introduits au début des années 70 [64]. Compte tenu de leur simplicité d'une part et leur performance d'autre part, ils sont à la fois très étudiés en recherche et très largement répandus dans l'industrie.
Nous rappelons brièvement les deux ordonnanceurs à priorité fixes les plus notoires : Rate Monotonic (RM) et Deadline Monotonic (DM) puis les ordonnanceurs à priorités dynamiques Earliest Deadline First (EDF) et Least Laxity First (LLF).

1.5.1 Résultat fondamental

Par définition, une application composée de tâches périodiques donne lieu à l'arrivée d'une infinité d'instances ou jobs soumis à échéances. Se pose donc le problème de décider hors-ligne si cette application est capable de respecter toutes ses échéances sur une architecture matérielle donnée. L'analyse d'ordonnançabilité vise à effectuer ce test avant le lancement de l'application. Si la réponse à ce problème de décision est oui, on pourra ainsi être garanti de la bonne marche du système. Et si la réponse est négative, on pourra par exemple modifier les tâches pour les raccourcir ou rallonger leur période ou encore opter pour un processeur plus rapide.
Toutefois, si nous utilisons une condition suffisante d'ordonnançabilité et que celle-ci rend une réponse négative, le seul moyen de tester l'ordonnançabilité de l'application par un ordonnanceur choisi sera de construire la séquence d'ordonnancement. Ce travail de simulation est rendu tout à fait praticable en raison du résultat central suivant prouvé en 1980 par les chercheurs Leung et Merril [68] :

Cyclicité de la séquence : La séquence produite par un ordonnanceur préemptif sur un ensemble de tâches périodiques est aussi périodique de période égale à H, l'hyperpériode.

Autrement dit, l'activité du processeur se répète avec cette hyperpériode. Ceci restreint donc l'intervalle de test à la première hyperpériode si les tâches sont synchrones. Selon que les périodes soient identiques, harmoniques c'est dire multiples l'une de l'autre ou sans relations, la longueur de cette fenêtre temporelle pourra être très courte ou très longue.

1.5.2 Rate Monotonic (RM)

Selon RM, plus grande est la période de la tâche plus haute est sa priorité [64].

Qualités de RM : RM se caractérise par les principaux avantages suivants :

- la prédictibilité : En cas de surcharge traitement, on sait que les échéances non satisfaites seront celles des tâches les moins prioritaires.
- Un faible overhead d'exécution : les priorités sont calculées hors ligne et sont invariables.
- la compatibilité avec le partage de ressources : Des protocoles de contrôle de l'accès à des ressources partagées ont été conçus pour s'utiliser avec l'ordonnanceur RM (voir paragraphes suivants).
- la simplicité d'implémentation : Une simple valeur entière représente la priorité qui est stockée dans le descripteur de tâche. .
- la popularité : RM est disponible dans quasiment tous les systèmes d'exploitation temps réel.

Illustration : Considérons l'ensemble de tâches périodiques suivant :
$\tau = \{\tau_1(2, 2, 8, 8), \tau_2(1, 4, 12, 12), \tau_3(0, 4, 24, 24)\}$.
Nous avons $U_p = \frac{2}{8} + \frac{4}{12} + \frac{4}{24} = 0.75$, où U_p désigne le taux d'utilisation du processeur. La séquence produite par RM est illustrée sur la Figure 1.3 entre les instants $t = 0$ et $t = 26$ qui représente le régime permanent du système ou *hyperpériode*.

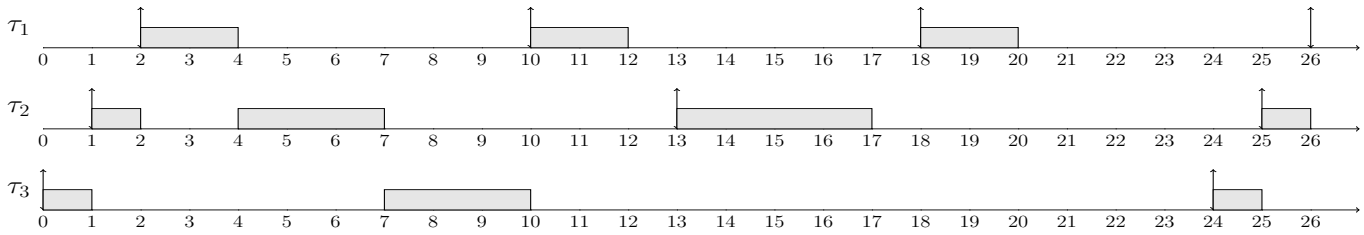


FIGURE 1.3 – Séquence d'ordonnancement selon RM

1.5.3 Deadline Monotonic (DM)

Selon DM, plus petite est l'échéance relative (délai critique), plus haute est la priorité [82]. Ainsi, lorsque les tâches sont à échéances implicites, DM se confond avec RM. Cet ordonnanceur à priorités fixes possède donc les mêmes qualités que RM.

Optimalité de DM : DM est optimal dans la classe des ordonnanceurs à priorités fixes pour l'ordonnancement de tâches périodiques, indépendantes, synchrones, à échéances contraintes.

Illustration : Considérons l'ensemble de tâches suivant :
$\tau = \{\tau_1(0, 2, 10, 10), \tau_2(0, 3, 15, 8), \tau_3(0, 4, 30, 24)\}$.

Nous avons $\frac{2}{10} + \frac{3}{8} + \frac{4}{24} = 0.74$. Le système est donc sous-chargé. La séquence produite par DM est illustrée par la Figure 1.4 sur la première hyper-période c'est à dire entre les instants $t = 0$ et $t = 30$.

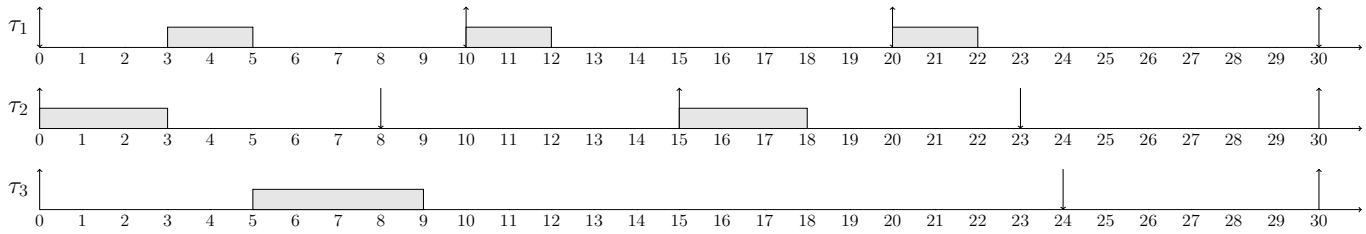


FIGURE 1.4 – Séquence d'ordonnancement selon DM

1.5.4 Earliest Deadline First (EDF)

Par définition, Earliest Deadline First (EDF) est un algorithme à priorité dynamique qui donne à tout instant la priorité la plus élevée à la tâche dont l'échéance absolue est la plus proche c'est à dire à la tâche la plus urgente [64]. C'est d'ailleurs pour cette raison que cet ordonnanceur est parfois appelé *Relative Urgency*.

Notons que l'ordonnanceur EDF peut s'utiliser pour toutes tâches munies d'échéance, que ces tâches soient périodiques, sporadiques ou apériodiques. Par la suite, nous ne donnons que les propriétés d'EDF appliqué à des tâches périodiques. A chaque fois qu'une instance de tâche périodique est réveillée, la priorité de son instance courante est calculée. La priorité est donc changeante pour une tâche périodique mais elle est fixe pour chaque instance (job). Malgré cette différence, la manipulation des instances des tâches reste la même que dans le cas des algorithmes RM et DM dans le fait qu' à chaque instant de temps, l'instance exécutée est celle qui a actuellement la plus haute priorité parmi toutes les instances réveillées des tâches. EDF n'est donc pas plus compliqué à implémenter qu'un ordonnanceur à priorité fixe.

Optimalité de EDF : EDF est optimal, signifiant que EDF est le meilleur ordonnanceur pour des tâches périodiques indépendantes. Cette définition de l'optimalité concerne les systèmes à contraintes dures.
La performance de l'algorithme EDF en fait un sujet d'étude récurrent depuis plus de 40 ans. Comme DM et RM, il peut aussi s'implémenter conjointement à des protocoles de contrôle d'accès à des ressources. Il peut s'adapter aussi à des contraintes de précédence. Le lecteur pourra trouver une étude détaillée sur EDF dans l'ouvrage très référencé [21].

Avantages/inconvénients de EDF : En plus de sa haute performance en termes de taux de respect des échéances et de son implémentation aisée, EDF présente un très gros avantage concernant les overheads de préemptions. En effet, EDF engendre un nombre minimal de préemptions en comparaison aux autres ordonnanceurs conduits par la priorité. De plus, EDF s'applique à toute tâche munie d'une échéance. Son optimalité est donc élargie à des tâches sporadiques et des tâches apériodiques. Il sera donc possible d'ordonnancer conjointement l'ensemble de ces tâches. Il suffira de mettre dans une seule liste ordonnée instances de tâches périodiques, instances de tâches sporadiques et tâches apériodiques.
En situation de surcharge de traitement, un algorithme d'ordonnancement tel que RM ou DM aura un comportement déterministe puisqu'on connaît a priori les tâches qui violeront leur échéance, celles ayant la plus basse priorité. En revanche, pour un ordonnanceur à priorité dynamique comme EDF, nous ne pouvons a priori déterminer quelles seront les tâches affectées par la surcharge puisque la priorité d'une tâche périodique varie au cours du temps. L'imprédictabilité du comportement de EDF en pareille situation est donc son principal défaut. Et c'est très probablement pour ce défaut que la communauté industrielle lui préfère des ordonnanceurs à priorité fixe.

Illustration : Reprenons l'exemple précédent. La figure 1.5 décrit la séquence d'ordonnancement produite par EDF.

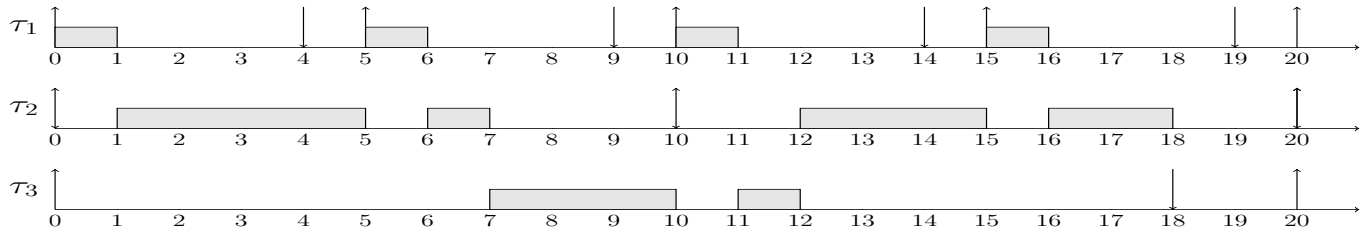


FIGURE 1.5 – Séquence d'ordonnancement selon EDF

1.5.5 Least Laxity First (LLF)

Par définition, l'algorithme d'ordonnancement LLF (Least Laxity First), proposé par Mok et Dertouzos [90], est un algorithme d'ordonnancement à priorité dynamique qui à tout instant donne la priorité à la tâche dont la date de début d'exécution au plus tard (sans violer son échéance) est la plus proche.
Autrement dit, LLF donne à tout instant la priorité la plus élevée à la tâche qui a la *laxité* la plus petite. La laxité d'une tâche τ_i à un instant t donné correspond à la durée maximale durant laquelle le processeur peut rester passif à partir de t sans remettre en question le respect d'échéance de τ_i. En d'autres termes, la laxité de τ_i à un instant t est égale à la différence entre l'échéance de l'instance en cours de τ_i et la charge de travail restante, soit $L(t, \tau_i) = d_i - (t + C_i(t))$. Notons que $C_i(t)$ représente la durée d'exécution restante de la tâche à l'instant t. Pour éviter une perte d'échéance, nous devons avoir pour chaque tâche τ_i : $\forall t, L(t, \tau_i) \geq 0$.

Illustration : Soit l'ensemble de tâches $\tau = \{\tau_1(0, 2, 5, 4), \tau_2(0, 4, 10, 10), \tau_3(0, 1, 20, 8)\}$. L'ordonnancement par LLF de cet ensemble de tâches est présenté sur la Figure 1.6.

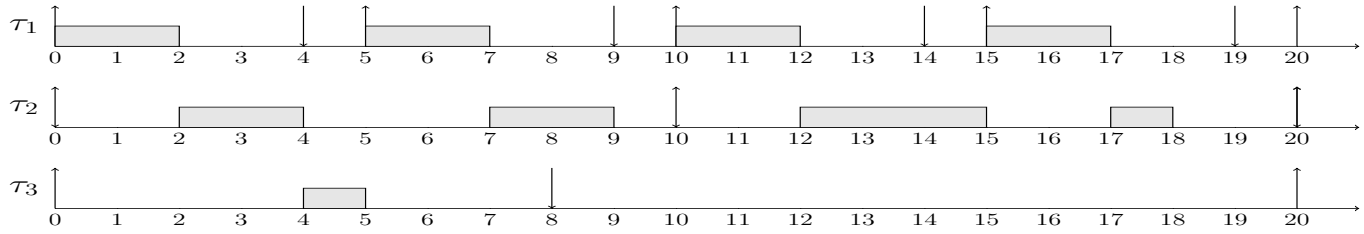


FIGURE 1.6 – Séquence d'ordonnancement selon LLF

Avantages/inconvénients de LLF : Le travail présenté dans [28] montre que les conditions d'ordonnançabilité par l'algorithme LLF sont identiques à celles de EDF. Ceci s'explique par le fait que EDF et LLF sont tous deux optimaux.

Toutefois, contrairement à EDF, LLF s'avère inexploitable pour les principales raisons suivantes : Il engendre un nombre prohibitif de préemptions d'où des overheads également prohibitifs. La priorité est dynamique non seulement pour chaque tâche mais aussi pour chaque instance de tâche périodique. En effet, la laxité est une grandeur dynamique. Pour une tâche, elle est constante si la tâche s'exécute et diminue sinon, d'où la nécessité de mises à jour de cette grandeur à chaque fois qu'il y a préemption, réveil et terminaison d'une tâche.

Dans cette thèse, nous nous focaliserons sur l'ordonnanceur optimal EDF.

1.6 Analyse d'ordonnançabilité pour EDF

Il existe plusieurs techniques et méthodes pour vérifier la faisabilité temporelle d'un ensemble de tâches temps réel sur un système monoprocesseur. Ces techniques se fondent sur le facteur d'utilisation, le temps de réponse ou la demande processeur.
Dans la suite de cette section, nous rappelons les tests les plus notoires de par leur simplicité de mise en oeuvre ou leur puissance lorsqu'ils fournissent des conditions exactes d'ordonnançabilité.

1.6.1 Analyse basée sur le facteur d'utilisation

CNS d'ordonnançabilité : Un ensemble de tâches périodiques à échéances implicites est ordonnançable par EDF si et seulement si $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.
Cette condition met bien en évidence la puissance de EDF puisqu'il optimise la capacité de traitement du processeur. Toutes les échéances d'une application peuvent être respectées alors que le processeur exécute des tâches sans aucune période d'oisiveté.

CS d'ordonnançabilité : Un ensemble de tâches périodiques à échéances contraintes est ordonnançable par EDF si $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1$.
Cette condition bien que suffisante présente l'avantage d'être extrèmement simple à mettre en oeuvre dans un analyseur d'ordonnançabilité.

1.6.2 Analyse basée sur le temps de réponse

Ce test consiste à vérifier que l'échéance de toute instance de chacune des tâches périodiques est satisfaite. On calcule donc pour chaque tâche son pire temps de réponse défini comme le plus grand intervalle de temps séparant le réveil d'une instance de sa fin d'exécution. Puis on compare ce temps à l'échéance relative de la tâche.

Quand on utilise un ordonnanceur non oisif comme EDF, un dépassement d'échéance ne peut jamais survenir lorsqu'il existe un ou plusieurs intervalles d'inactivité du processeur entre le réveil d'une tâche et sa fin d'exécution.

Instant critique : La plus longue période d'activité du processeur se produit lors du réveil simultané de toutes les tâches du système. Cet instant est appelé *instant critique*.

Request Bound Function : La fonction $RBF(\tau_i, [0, t[)$ (Request Bound Function) est la durée d'exécution cumulée des instances de τ_i dont la date de réveil est dans l'intervalle $[0, t[$ et l'échéance pas nécessairement avant t.
On l'obtient par la formule suivante :

$$RBF(\tau_i, [0, t[) = \max\left(0, \lceil \frac{t - r_i}{T_i} \rceil\right) .C_i \tag{1.1}$$

La durée totale maximale d'activité du processeur sur τ, dans $[0, t[$, notée $RBF(\tau, [0, t[)$ s'obtient par : $RBF(\tau, [0, t[) = \sum_{i=1}^{n} RBF(\tau_i, t)$ et donc :

$$RBF(\tau, [0, t[) = \sum_{i=1}^{n} \max\left(0, \lceil \frac{t - r_i}{T_i} \rceil\right) .C_i. \tag{1.2}$$

Dans le cas de tâches synchrones, nous avons donc : $RBF(\tau,[0,t[) = \sum_{i=1}^{n} \max\left(0, \lceil \frac{t}{T_i} \rceil\right).C_i$ soit aussi :

$$RBF(\tau,[0,t[) = \sum_{i=1}^{n} \lceil \frac{t}{T_i} \rceil.C_i \tag{1.3}$$

La fonction de travail du processeur $RBF(\tau,[0,t[)$ est une fonction de temps en escalier. La fonction linéaire $f(t) = t$ correspond à la capacité maximale de traitement du processeur. Lorsque $RBF(\tau,[0,t[) = t$, le processeur à cet instant a terminé l'exécution de toutes les instances réveillées dans l'intervalle $[0,t[$.

Theorem 1 *[98] le pire temps de réponse R_i d'une tâche τ_i est défini par le plus petit point fixe de l'équation :*

$$RBF(\tau,[0,t[) = t. \tag{1.4}$$

Le respect des échéances est testé en vérifiant que : $R_i \leq D_i$, pour toute tâche τ_i, $1 \leq i \leq n$.
Le pire temps de réponse peut être calculé en faisant une simulation d'ordonnancement EDF.

1.6.3 Analyse basée sur la demande processeur

L'analyse de la demande processeur, comme l'analyse des temps de réponse, repose sur la caractérisation du pire comportement, ici en considérant la charge globale du système et non en analysant les tâches séparément.

Demand Bound Function : La fonction $DBF(t_1,t_2)$ (Demand Bound Function) représente la durée d'exécution cumulée des instances dont la date de réveil et l'échéance sont dans l'intervalle $[t_1,t_2]$.

La demande de traitement pour une tâche τ_i est donnée par $\max\left(0, \lfloor \frac{t-r_i-D_i}{T_i} \rfloor + 1\right).C_i$.
Ainsi, la demande processeur de toutes les instances, dans $[0,t]$ avec des échéances inférieures ou égales à t s'obtient par : $DBF(0,t) = \sum_{i=1}^{n} \max\left(0, \lfloor \frac{t-r_i-D_i}{T_i} \rfloor + 1\right).C_i$.

Pour des tâches synchrones, $r_i = 0$ et $D_i \leq T_i$, nous pouvons simplifier le calcul de la fonction $DBF(0,t)$ par :

$$DBF(0,t) = \sum_{i=1}^{n} \max\left(0, \lfloor \frac{t+T_i-D_i}{T_i} \rfloor\right).C_i \tag{1.5}$$

Theorem 2 *[54] Un ensemble de tâches périodiques à échéances contraintes est ordonnançable par EDF si et seulement si pour tout L, $L \geq 0$,*

$$L \geq \sum_{i=1}^{n} \lfloor \frac{L}{T_i} \rfloor. \tag{1.6}$$

1.7 EDF au plus tard

L'algorithme d'ordonnancement *Earliest Deadline as Late as possible* (EDL) [25] revient à exécuter les tâches périodiques au plus tard mais en ordonnant les tâches selon la règle de EDF. Pour ce faire, il est nécessaire de connaître les caractéristiques temporelles des tâches de sorte à calculer les périodes pendant lesquelles le processeur sera occupé à les exécuter. On pourra ensuite en déduire les temps libres appelés *temps creux du processeur* rendus disponibles pour, soit laisser le processeur inactif, soit exécuter d'autres tâches au plus tôt.

Dans [25], Chetto et Chetto proposent une méthode simple pour déterminer la localisation et la durée des temps creux d'une séquence EDL. La terminologie utilisée par les auteurs est la suivante : f_Y^X représente la fonction de disponibilité définie pour une configuration de tâches Y et un algorithme d'ordonnancement X,

$$f_Y^X = \begin{cases} 1 & \text{si} \quad le\ processeur\ est\ inactif\ à\ t \\ 0 & \text{sinon} \end{cases} \tag{1.7}$$

Ainsi, l'intégrale $\int_{t_1}^{t_2} f_Y^X(t)dt$ fournit le nombre total d'unités de temps creux disponibles durant l'intervalle de temps [t1, t2].

Calcul des temps creux statiques : Le calcul de f^{EDL} hors-ligne repose sur deux vecteurs :
- Le vecteur des échéances qui représente l'ensemble des instants auxquels débute un temps creux sur une hyper-période H correspondant à l'intervalle [0,H[.
- Le vecteur des temps creux qui représente la durée de temps creux associés aux différents instants dans le vecteur des échéances.

Soit une configuration de *n* tâches périodiques à échéances sur requêtes τ=$\{\tau_j(\ C_j, D_j, T_j), j = 1\ à\ n\}$. C_j est le temps d'exécution pire-cas, D_j est l'échéance relative et T_j est la période. Le vecteur statique des échéances est noté $\mathcal{K} = (k_0, .., k_i, .., k_q)$. Il contient l'ensemble des instants qui précédent un temps creux sur une hyper-période. Ce vecteur se construit à partir des échéances des instances de tâches périodiques [127]. Les instants k_i de départ des intervalles de temps creux dans le vecteur des échéances $\mathcal{K} = (k_0, ..., k_i, ..., k_q)$sont définis comme suit [25] : $k_i = x.D_j$ avec $x = 1, ..., \frac{H}{T_j}$, $k_i < k_{i+1}$, $k_0 = 0$ et $k_q = H - min(T_j; 1 \leq j \leq n)$.

Le vecteur statique des temps creux s'écrit comme suit : $\mathcal{D}^* = (\Delta_0^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$. Δ_i^* représente la longueur du temps creux débutant à l'instant k_i. Le vecteur $\mathcal{D}^* = (\Delta_0^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ s'obtient comme suit [25] : $\Delta_q^* = min\{T_j; 1 \leq j \leq n\}$, $\Delta_i^* = sup(0, F_i)$ avec

$$F_i = (H - k_i) - \sum_{j=1}^{n} \left\lceil \frac{H-k_i}{T_j} \right\rceil C_j - \sum_{l=i+1}^{q} \Delta_l^*.$$

Exemple illustratif
Nous considérons l'ensemble de tâches périodiques $\tau = \{\tau_1(3,\ 10,\ 10),\ \tau_2(4,\ 15,\ 15)\}$. Les tâches sont synchrones et réveillées simultanément à l'instant 0. Nous appliquons les équations précédentes pour calculer hors-ligne, le vecteur statique des échéances et le vecteur statique des temps creux. Ainsi $\mathcal{K} = \{0,\ 9,\ 12,\ 18,\ 24,\ 27\}$ et $\mathcal{D}^* = \{5,\ 0,\ 3,\ 2,\ 0,\ 2\}$. La séquence EDL calculée est représentée par la Figure 1.7.

Calcul des temps creux dynamiques : Le calcul de f^{EDL} en-ligne repose sur deux vecteurs, le vecteur dynamique des échéances et le vecteur dynamique des temps creux.

Le vecteur dynamique des échéances à un instant *t*, $\mathcal{K}(t)$, représente les instants supérieurs ou égaux à *t* qui précèdent un temps creux. Il se construit à partir des échéances distinctes des tâches périodiques supérieures ou égales à l'instant *t*. Ainsi $\mathcal{K}(t) = (t, k_{h+1}, ..., k_i, ..., k_q)$ sachant que la valeur de *t* vient remplacer l'index *h* qui est le plus grand index présent dans le vecteur statique des échéances tel que $k_h = sup(d, d < t)$.

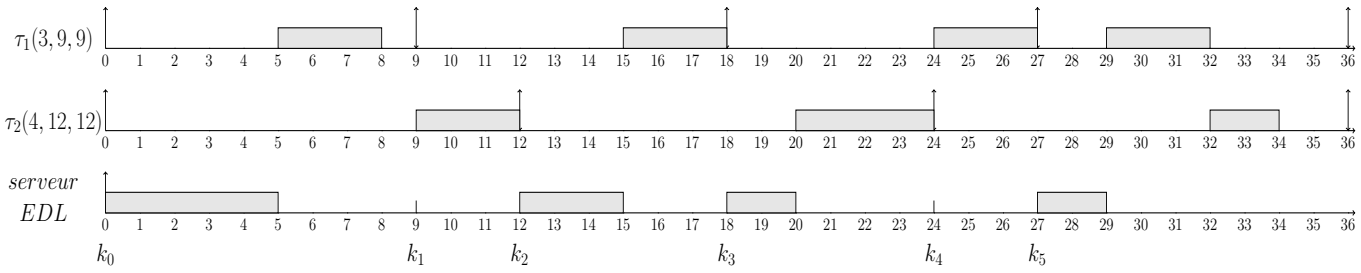


FIGURE 1.7 – Calcul des temps creux statiques avec l'algorithme EDL

Le vecteur dynamique des temps creux $\Delta^*(t) = (\Delta^*_h(t), \Delta^*_{h+1}(t), ..., \Delta^*_i(t), ..., \Delta^*_q(t))$ représente les durées des temps creux associés aux instants du vecteur $\mathcal{K}(t)$. Soit *M* la plus grande échéance parmi celles de toutes les instances périodiques actives à l'instant *t*. Par conséquent l'indice *f* fait référence au plus petit index dans le vecteur des temps creux tel que $k_f = min\{k_i; k_i > M\}$. On calcule les éléments de $\Delta^*(t)$ comme suit [25] : $\Delta^*_i(t) = \Delta^*_i$, pour i de q à f,
$\Delta^*_i(t) = sup(0, F_i(t))$ pour i de f-1 à h+1
où $F_i(t) = (H - k_i) - \sum_{j=1}^{n} \left\lceil \frac{H-k_i}{T_j} \right\rceil C_j + \sum_{j=1 d_j > k_i}^{n} A_j(t) - \sum_{k=i+1}^{q} \Delta^*_k(t)$
avec $\Delta_h(t) = (H - t) - \sum_{j=1}^{n} (\left\lceil \frac{H-t}{T_j} \right\rceil C_j - A_j(t)) - \sum_{k=i+1}^{q} \Delta^*_k(t)$.

$A_j(t)$ représente la quantité de temps processeur déjà allouée à l'instance courante de τ_j jusqu'à l'instant *t* et d_j est l'échéance absolue de la tâche périodique τ_j.

Exemple illustratif :
Nous considérons toujours le même ensemble de tâches périodiques τ = $\{\tau_1(3,\ 10,\ 10),\ \tau_2(4,\ 15,\ 15)\}$. Nous appliquons les équations précédentes pour calculer en-ligne le vecteur dynamique des échéances et le vecteur dynamique des temps creux à l'instant 4. Ainsi $\mathcal{K} = \{4,\ 9,\ 12,\ 18,\ 24,\ 27\}$ et $\mathcal{D}^* = \{2,\ 2,\ 4,\ 2,\ 0,\ 2\}$. La séquence EDL calculée est représentée par la Figure 1.8.

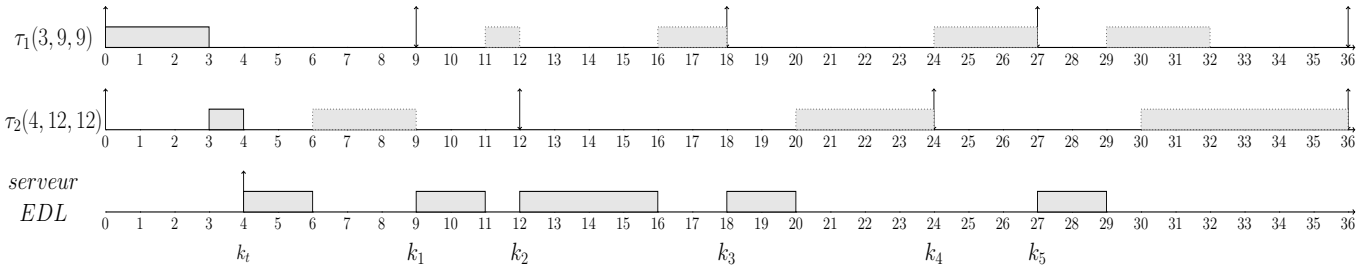


FIGURE 1.8 – Calcul des temps creux dynamiques selon EDL

1.8 Conclusion

Nous avons fait, dans ce chapitre, un bref rappel concernant l'informatique temps réel. Nous avons mis en évidence la caractéristique essentielle d'une application temps réel, à savoir la périodicité d'exécution des fonctions implémentées sous forme de programmes séquentiels, les tâches. L'exécution de celles-ci dans le respect des échéances impose l'intégration dans le système d'exploitation d'un ordonnanceur. Nous avons

donc rappelé les algorithmes d'ordonnancement les plus performants dans un contexte de sous-charge et donné pour ceux-ci les tests d'ordonnançabilité.

Nous avons souligné que dans une application temps réel à contraintes fermes, un certain ratio d'échéances violées peut être toléré. C'est pourquoi, la Qualité de Service pour cette catégorie d'application s'exprime d'abord par le pourcentage d'échéances effectivement respectées.

Le prochain chapitre de cette première partie *état de l'art* s'attachera à rappeler les notions de tolérance aux fautes temporelles puis à décrire les principales techniques de tolérance aux fautes temporelles existantes.

2

Gestion contrôlée des fautes temporelles

2.1 Présentation du chapitre

Les systèmes temps réel réalisent des traitements complexes que l'on qualifie de critiques car ils sont soumis à des contraintes strictes qui portent d'une part sur la justesse des résultats issus des calculs et d'autre part sur les dates auxquelles ces résultats doivent être disponibles. Le non respect de ces exigences peut alors avoir des conséquences catastrophiques du point de vue économique, matériel voire humain (perte d'argent, de temps, ou de vies humaines dans le pire cas). Pour cette raison, ils doivent mettre en oeuvre des *mécanismes de tolérance aux fautes*.

Dans ce chapitre, nous revenons sur les concepts de base de la tolérance aux fautes pour l'informatique temps réel. Nous nous focalisons ensuite sur les fautes dites temporelles. Et nous décrivons les principales méthodes qui se donnent comme objectif de limiter l'impact de ces fautes temporelles sur la qualité du service rendu dans un système temps réel surchargé.

2.2 Surcharge et faute temporelle

2.2.1 Notion de surcharge

Considérons une architecture monoprocesseur qui doit supporter l'exécution d'un ensemble de tâches périodiques. Celle-ci inflige au processeur une charge de traitement définie par le pourcentage de temps pendant lequel le processeur travaille. Dans un système dit *sous-chargé*, ce taux appelé aussi facteur d'utilisation est inférieur ou égal à un. Nous avons vu dans les paragraphes précédents les principaux ordonnanceurs reconnus pour leur haute performance.

Toutefois, nous devons noter que ces ordonnanceurs ont un comportement satisfaisant et déterministe uniquement lorsque les tests d'ordonnançabilité ont été effectués et s'avèrent satisfaits. Dans le cas contraire, ces ordonnanceurs peuvent avoir un comportement complètement indéterministe ce qui est le cas de EDF. En effet, si sur un intervalle de temps, la charge infligée au processeur est supérieure à sa capacité de traitement, l'identité des tâches qui violeront leur échéance est difficilement prévisible. Les algorithmes classiques tel que RM et EDF doivent donc être remplacés par des ordonnanceurs plus adéquats en cas de surcharge.

Surcharge : On dit qu'un processeur est en surcharge si la capacité de traitement du processeur est inférieure à la charge de traitement qu'on lui inflige.

Surcharge temporaire : On dit qu'un processeur est en surcharge temporaire sur un intervalle de temps donné si sa capacité de traitement est inférieure à la charge qu'on lui afflige sur cet intervalle.

L'origine de la surcharge d'un processeur peut avoir lieu dans plusieurs circonstances. Par exemple, il peut s'avérer très difficile d'estimer le WCET de certaines tâches. Si la valeur du WCET a été mal estimée ou si le test d'ordonnançabilité a été effectué sur la base de valeurs moyennes de la durée d'exécution, des surcharges de traitement intermittentes pourront avoir lieu. Il va de soi que ces surcharges ne sont tolérées que dans des systèmes temps réel souples ou fermes.

2.2.2 Notion de faute temporelle

La *défaillance* ou *panne* d'un système est la conséquence d'une *erreur* qui elle-même est la conséquence d'une faute activée.

La faute : La faute dans un système informatique peut correspondre à un défaut dans un composant matériel, ou dans un composant logiciel de ce système. Elle est le plus souvent de nature accidentelle à cause des phénomènes physiques ou de l'imperfection naturelle de l'homme. Une faute peut dormir jusqu'à ce qu'un événement provoque son activation (par exemple, une division par zéro).

Les fautes dans un système temps réel sont divisées en trois catégories selon leur persistance temporelle : fautes permanentes, fautes transitoires et fautes intermittentes. Beaucoup d'auteurs affirment que la fréquence d'apparition des fautes transitoires et intermittentes est très grande par rapport à celle de fautes permanentes [40]. On rapporte que les fautes transitoires surviennent 30 fois plus que les fautes permanentes dans un système temps réel [129].

Les fautes peuvent avoir une nature fonctionnelle ou temporelle :

- *Faute fonctionnelle* : La valeur délivrée par le système est fausse c'est à dire non conforme à sa spécification, ou en dehors d'une plage de valeurs attendues.

- *Faute temporelle* : L'instant auquel la valeur est délivrée se trouve en dehors de l'intervalle de temps spécifié. Dans le cas d'un système temps réel, une faute temporelle correspond à la non délivrance du résultat à l'échéance spécifiée.

Dans la suite de cette thèse, nous nous focalisons sur les fautes intermittentes de nature temporelle.

2.2.3 Origine des fautes temporelles

La *tolérance aux fautes temporelles* représente la propriété d'un système à pouvoir délivrer un service conforme à sa spécification en présence de fautes temporelles.

Dans un système temps réel, chaque instance de tâche périodique doit idéalement produire un résultat final exact c'est à dire se terminer, avant son échéance. Un système temps réel idéal serait donc celui qui, sur toute la durée de vie de l'application, ne présente aucune faute temporelle. En d'autres termes, c'est un système où chaque échéance de tâche produit un résultat exact avant son échéance.

Les causes de l'occurrence de fautes temporelles sont multiples :

- défaillance du processeur : par exemple, un des circuits subit des perturbations électromagnétiques temporaires,
- défaillance du lien de communication : la tâche attend indéfiniment une donnée en provenance d'un autre processeur pour pouvoir continuer son traitement,
- idem que précédemment, la tâche attend indéfiniment une mesure à partir d'un capteur défaillant,
- le processeur se trouve en surcharge de traitement. Il ne peut satisfaire toutes les échéances des tâches étant donné sa capacité de traitement limitée,
- le code exécuté par une tâche boucle indéfiniment pour certaines données d'entrée.
- etc.

En résumé, nous devons donc distinguer deux origines distinctes à la faute temporelle :

- la surcharge de traitement,
- et la défaillance d'origine logicielle ou matérielle.

La plupart des méthodes présentées dans la littérature pour la conception des systèmes tolérants aux fautes se concentrent sur la défaillance du matériel (processeurs et liens de communications) qui engendrent des fautes fonctionnelles et non des fautes temporelles. Ces méthodes utilisent la *réplication* appelée aussi *redondance* pour concevoir des systèmes *sûrs de fonctionnement* en toutes circonstances.

2.2.4 Redondances logicielles

La redondance logicielle consiste à répliquer les tâches en plusieurs versions de code, identiques ou non. Si une version est l'objet d'une faute temporelle, au moins une autre version de la même tâche doit être exempte de faute temporelle.

Il existe deux approches pour implémenter la redondance logicielle dans un système informatique :

1. *La redondance active* impose que toutes les versions d'une même tâche soient systématiquement exécutées.

2. *La redondance passive* suppose qu'une seule des versions de chaque composant logiciel, appelée *version primaire*, est exécutée à la fois. Les autres versions sont appelées *versions de sauvegarde ou versions secondaires* (en anglais back-up). Le nombre de versions secondaires dépend du nombre maximum de fautes pouvant survenir. Une version secondaire n'est exécutée que consécutivement à l'arrivée d'une faute concernant une autre version de la même tâche. C'est pourquoi, on parle de *redondance dynamique passive*.

L'inconvénient majeur de la redondance active réside dans le supplément de charge de traitement qu'elle engendre. De ce fait, cela augmente la probabilité d'engendrer des fautes temporelles sur une architecture matérielle monoprocesseur car les versions ne peuvent pas s'exécuter en parallèle.

2.3 Fautes temporelles et temps réel souple

Dans les applications à contraintes souples, aucun niveau minimal de QdS n'est requis. D'où le caractère optionnel d'un test de faisabilité hors-ligne. L'ordonnanceur fera au mieux pour ne produire aucune faute temporelle ce qui reste toutefois impossible sur un processeur en surcharge de traitement ou sujet à des défaillances. Les ordonnanceurs généralement utilisés dans ce contexte sont ceux décrits ci-dessous.

2.3.1 Approche Best Effort

Un ordonnanceur *Best-Effort* accepte systématiquement les tâches qui arrivent sans test d'acceptation et les insère dans la file des tâches prêtes. Ainsi, les tâches sont ordonnancées jusqu'à leur terminaison ou jusqu'à dépassement d'échéance. Comme il n'y a aucune prédiction de surcharge, l'ordonnancement est basé sur les priorités données aux tâches. Ce type d'approche est souvent sensible à l'effet Domino.

FIGURE 2.1 – Ordonnanceurs meilleur effort.

Exemple : L'ordonnanceur D-over
Dans [57], Koren et Shasha proposent un algorithme d'ordonnancement en ligne, appelé $D-over$ adapté aux systèmes temps réel surchargés. $D-over$ est optimal dans le sens où il donne le meilleur facteur de compétitivité parmi tous les algorithmes en ligne. Ce facteur est de $0,25$. Rappelons que *le facteur de compétitivité* renseigne sur la performance d'un algorithme en ligne comparée à celle d'un algorithme clairvoyant qui a une connaissance de l'avenir.

Dans le cas où aucune surcharge n'est détectée, $D-over$ se comporte comme l'algorithme EDF. La détection d'une surcharge se fait lorsqu'une tâche prête atteint sa date de démarrage au plus tard souvent appelée LST (Latest Start Time) donnée par l'échéance minorée de la durée d'exécution. A cet instant, une tâche doit être abandonnée : soit la tâche qui a atteint son LST soit une autre tâche.

L'ensemble des tâches prêtes se divise en deux sous-ensembles disjoints : les tâches privilégiées et les tâches en attente. Chaque fois qu'une tâche est préemptée, elle devient une tâche privilégiée. Cependant, à chaque fois qu'une tâche est exécutée au plus tard sur son LST, toutes les tâches prêtes (si elles sont préemptées ou jamais exécutées) deviennent des tâches en attente.
Lorsqu'une surcharge est détectée sur l'atteinte par une tâche τ_z de son LST, la valeur de τ_z est comparée à la valeur totale V_{priv} de toutes les tâches privilégiées (y compris la valeur v_{curr} de la tâche en cours d'exécution).

2.3.2 Approche à garantie

Chaque tâche qui arrive subit un test d'acceptation en ligne (en anglais, guarantee ou admission test) empêchant le système d'être surchargé. Si le test réussit, la tâche est acceptée ; sinon elle est rejetée. Ces ordonnanceurs ont l'avantage par rapport aux ordonnanceurs *Best-Effort* de rejeter toute nouvelle tâche pouvant causer la surcharge du système. Cependant, il ne prennent pas en compte les valeurs associées à l'importance des tâches. Une variante de EDF avec test d'acceptation appelé GED (pour Guaranteed Earliest Deadline) est présenté dans [16].

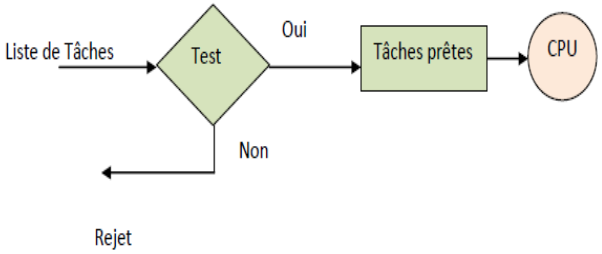


FIGURE 2.2 – Ordonnanceurs à garantie.

2.3.3 Approche robuste

Dans un ordonnanceur robuste, l'ordonnancement est contrôlé par deux politiques séparées. Les tâches sont ordonnancées selon leurs échéances et rejetées selon leur importance. En d'autres termes, en cas de refus d'une tâche, l'ordonnanceur tente d'abandonner une ou plusieurs tâches de moindre importance. Les tâches rejetées pourront être reprises plus tard si la charge du système le permet. Les ordonnanceurs robustes sont en général plus performants que ceux basés sur les deux modèles précédents [16].

Cette stratégie peut être considérée comme une amélioration de la politique précédente en ce sens qu'une tâche rejetée à un instant donné pourra être acceptée ultérieurement. Par exemple, supposons un test effectué sur la base des WCET. Si les tâches consomment beaucoup moins de temps d'exécution, la tâche refusée pourra ensuite être acceptée, le processeur s'étant rendu disponible plus tôt.

Bien que simples à mettre en oeuvre, ces techniques ne sont pas adaptées à des contraintes temps réel fermes. Elles ne garantissent pas un niveau de Qualité de Service minimum avant le lancement de l'application. En plus de la Qualité de Service, il faut également veiller à ce qu'en toutes circonstances, nous observions une répartition de traitement qui soit conforme aux spécifications. Il pourra s'agir d'un traitement équitable entre toutes les tâches ou au contraire d'un traitement qui privilégie certaines tâches par rapport à d'autres. Cela suppose donc d'adjoindre un mécanisme de contrôle de l'identité des tâches qui sont autorisées à supporter des fautes temporelles c'est à dire violer leur échéance à un instant donné.

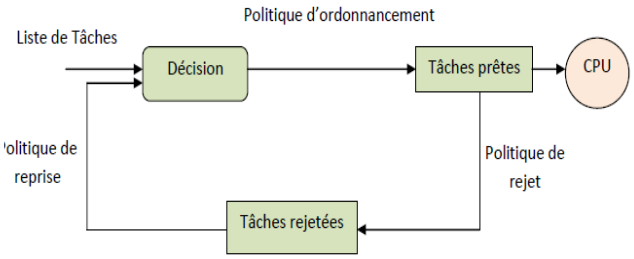


FIGURE 2.3 – Ordonnanceurs robustes.

2.4 Fautes temporelles et temps réel dur

2.4.1 Problématique

Pour une application temps réel dur, idéalement le concepteur doit anticiper toute possibilité de fautes temporelles et de surcharge. En d'autres termes, cent pour cent des instances de tâches doivent fournir leurs données de sortie avant échéance. La non réalisation de cet objectif est considéré comme inacceptable. Pour ce faire, il doit surdimensionner le système en sélectionnant un processeur de vitesse suffisamment élevée pour garantir un taux d'utilisation inférieur à un. Ce taux d'utilisation est celui qualifié de pire cas qui consiste à considérer le pire temps d'exécution d'une tâche.

Une surcharge ne pourra donc se produire que dans des situations marginales tel qu'une mauvaise estimation des durées d'exécution ou l'arrivée non anticipée d'événements demandant un traitement en urgence comme celui lié à une alarme. Des fautes temporelles pourront toutefois se produire non pas pour cause de surcharge mais comme résultant de fautes de programmation, de panne matérielle qui entraîne le blocage permanent d'une tâche sur un message qui n'arrive pas pour cause de défaillance du réseau, etc.

Comme nous nous intéressons à une machine monoprocesseur ne possédant aucune redondance matérielle, la seule façon de faire face à l'occurrence de fautes temporelles dans un système temps réel dur sera d'accepter *un fonctionnement temporaire dit dégradé*. Un tel fonctionnement se caractérise par une adaptation en ligne de la charge infligée au processeur de sorte que le processeur reste toujours sous-chargé.

Les principales techniques connues sont les suivantes :

- celles où la tâche produit toujours un résultat avant l'échéance mais avec un raccourcissement éventuel de sa durée d'exécution, le résultat étant alors approché mais toutefois acceptable et exploitable. C'est le cas du **Calcul Imprécis**.

- celles où la tâche produit toujours un résultat exact en évitant la surcharge par une modification en ligne de ses paramètres temporels. C'est le cas du **modèle élastique**.

- celles où la tâche produit toujours un résultat avant l'échéance mais avec une autre version de code, le résultat étant alors de qualité dégradée mais toutefois acceptable et exploitable. C'est le cas du **Mécanisme à Échéance** précisément décrit dans la suite du chapitre.

2.4.2 Le Calcul Imprécis

On conçoit le code de la tâche en deux parties. Une dite *obligatoire* doit impérativement être exécutée en premier pour fournir un résultat de précision minimale. L'autre dite *optionnelle* s'exécute chaque fois que possible de façon à améliorer la précision du résultat calculé par la partie obligatoire. Il s'agit de la méthode du Calcul Imprécis (Imprecise Computation) largement étudiée dans [71] [66] [39] [67] [65].
Le calcul imprécis convient seulement aux tâches qui mettent en oeuvre des algorithmes itératifs où la précision du résultat du calcul croit avec le nombre d'itérations effectuées.
A noter que le modèle classique de Liu et Layland peut être considéré comme un cas particulier du modèle du Calcul Imprécis, avec une partie optionnelle de durée nulle. Dans le modèle de calcul imprécis, chaque tâche τ_i est caractérisée par le quadruplet suivant : $\tau_i(r_i, d_i, m_i, o_i)$, où r_i, d_i, m_i, et o_i désignent respectivement, la date de réveil, l'échéance, la durée d'exécution de la sous-tâche obligatoire et la durée d'exécution de la sous-tâche optionnelle [71] [65]. Supposons que e_i désigne la durée totale d'exécution de τ_i, alors, $e_i = m_i + o_i$.

L'objectif d'un ordonnanceur appliqué à des tâches conformes au Calcul Imprécis, sera d'obtenir une erreur totale minimum pondérée tout en garantissant l'absence de fautes temporelles. Dans ce modèle, *un ordonnancement est dit faisable*, si le processeur est capable d'exécuter chaque sous-tâche obligatoire m_i avant son échéance. En d'autres termes, le système doit être sous-chargé en ne considérant que les sous-tâches obligatoires. *Un ordonnancement est dit précis*, si l'erreur moyenne sur l'ensemble de tâches est égal à zéro c'est à dire toutes les sous-tâches obligatoires et optionnelles sont achevées avec succès.

Illustration : Considérons un ensemble de trois tâches : une tâches classique τ_3 et deux tâches τ_1 et τ_2 comportant deux sous-tâches obligatoires et deux sous-tâches optionnelles des temps d'exécution respectivement respectivement 1 et 3.

Task	r_i	T_i	e_i	m_i	o_i
τ_1	0	16	5	4	1
τ_2	0	32	6	3	3
τ_3	0	8	4	4	0

TABLE 2.1 – Illustration du modèle de calcul imprécis.

RM ordonnance les trois tâches τ_3 et les deux sous-tâches obligatoires de τ_1 et τ_2. Les sous-tâches optionnelles s'exécutent avec la plus basse priorité c'est à dire dans les périodes laissées libres du processeur.
Nous constatons que :
Entre l'instant 15 et l'instant 16, la tâche τ_1 trouve un temps disponible pour exécuter sa sous-tâche optionnelle.
Entre l'instant 28 et l'instant 29, la tâche τ_1 trouve un temps disponible pour exécuter sa sous-tâche optionnelle pour sa deuxième instance.
C'est à l'instant 29 que la tâche τ_2 trouve du temps disponible pour exécuter sa sous-tâche optionnelle.

La figure 2.4 montre la séquence produite.

2.4.3 Le modèle élastique

Le taux d'utilisation du processeur par une tâche périodique dépend d'une part de sa durée d'exécution et d'autre part de sa période. Diminuer en ligne la charge imposée au processeur pour retrouver un état

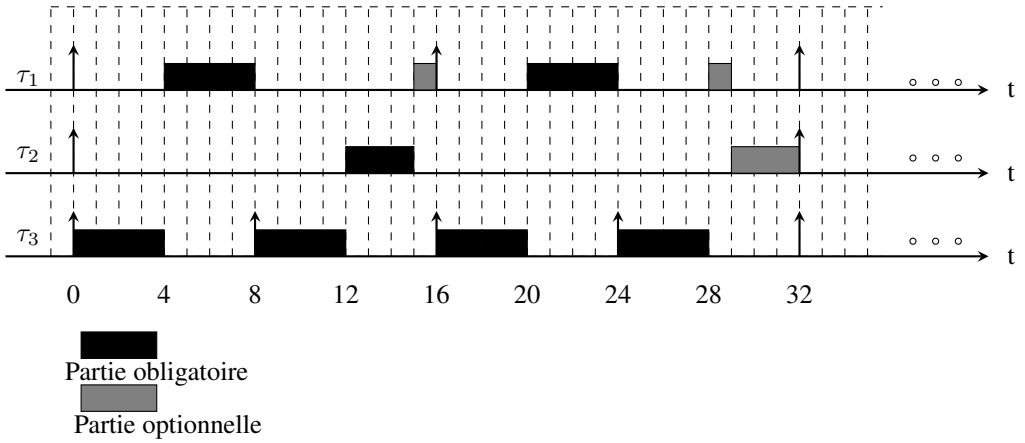


FIGURE 2.4 – Illustration du Calcul Imprécis

sous-chargé peut donc se faire par une augmentation de l'échéance ou/et de la période.

Dans la littérature, il existe plusieurs approches pour traiter le problème de surcharge grâce l'adaptation des périodes des tâches. Nous présenterons ici la plus connue, le *modèle élastique*.

Le modèle élastique [9] [15] revient à considérer que chaque tâche est flexible, comme un ressort élastique avec un coefficient de rigidité et une longueur de contraintes spécifiées.
Cependant, toute variation fait l'objet d'une garantie élastique et n'est acceptée que s'il existe un ordonnancement faisable dans lequel toutes les autres périodes sont considérées.

Selon ce modèle, chaque tâche τ_i est caractérisée par quatre paramètres :

- une durée d'exécution pire cas C_i,
- une période nominale T_i0 (considérée comme la période minimale),
- une période maximale T_imax,
- et un coefficient d'élasticité $E_i \geq 0$.

Le coefficient d'élasticité détermine la possibilité de la tâche à faire varier son facteur d'utilisation. Plus grande est la valeur E_i, de plus d'élasticité la tâche τ_i dispose.

Soit T_i la période effective de la tâche τ_i, qui doit être comprise dans l'intervalle $[T_i0, T_imax]$. Etant donné un ensemble de n tâches périodiques avec un facteur d'utilisation totale $U_p > U_{max}$, l'objectif dans le modèle élastique, est la garantie d'exécution des tâches en faisant compresser leurs facteurs d'utilisation de façon à réaliser une nouvelle utilisation souhaitée $U_d \leq U_{max}$ à condition que toutes les périodes restent dans leur plage d'acceptabilité.
Chaque fois qu'une nouvelle tâche ne peut pas être garantie sans faute temporelle, au lieu de la rejeter, le système peut tenter de réduire les facteurs d'utilisation d'autres tâches (en augmentant leur période de manière contrôlée). La conséquence en sera une diminution de la charge totale.
Dès qu'une situation de surcharge temporaire se termine, les tâches compressées peuvent récupérer leur période nominale.

Autres modèles adaptatifs

Dans [150], Wang et Lin proposent un modèle adaptatif en ce sens que l'on fait changer dynamiquement les paramètres temporels de l'application que sont l'échéance et la date de réveil.

L'objectif visé est d'obtenir un espacement temporel constant entre les instances adjacentes de tâches. La gigue d'exécution est fortement réduite avec ce modèle. Alors qu'elle peut varier de zéro à deux fois la période d'une tâche périodique dans un ordonnancement classique [29].

2.5 Fautes temporelles et temps réel ferme

2.5.1 Problématique

Concernant un système temps réel ferme, l'utilisateur spécifie le niveau minimum de Qualité de Service exigée ce qui correspond à un ratio seuil d'échéances à respecter ou autrement dit un ratio plafond de fautes temporelles. Ce ratio pourra être spécifié globalement pour l'ensemble des tâches ou individuellement pour chaque tâche du logiciel d'application. On doit disposer d'un test de faisabilité hors ligne qui permet de garantir que le système pourra toujours fournir une qualité de service au moins égale à celle spécifiée.

La conception d'un système temps réel ferme sujet à des fautes temporelles nécessite donc de répondre à deux questions clés :

1. Comment choisir une stratégie de tolérance aux fautes temporelles capable de garantir le respect de la QdS spécifiée lors du fonctionnement opérationnel ?

2. Déterminer le test de faisabilité hors ligne à satisfaire pour affirmer avant le lancement que dans la pire situation, la QdS observée sera toujours meilleure que la QdS spécifiée et ce, avec l'ordonnanceur sélectionné ?

2.5.2 Résolution de surcharge par pertes

Par définition, dans un environnement temps réel à contraintes fermes, la présence de fautes temporelles est toléré dans certaines limites.
Les *approches dites à pertes contraintes* sont donc tout à fait adaptées. Elles sont basées sur la spécification hors ligne des instances de tâche autorisées à être supprimées. Ainsi, la gestion de surcharge doit être anticipée et va reposer sur l'abandon contrôlé d'instances de tâches parfaitement sélectionnées pour diminuer l'utilisation effective du processeur. Les algorithmes se basant sur cette méthode devront donc maintenir un nombre de fautes temporelles inférieur à une valeur limite spécifiée, sous peine d'entraîner un dysfonctionnement inacceptable voire dangereux.

Les principales approches à pertes contraintes connues sont répertoriées dans les paragraphes suivants.

Le modèle Skip-Over

Chaque tâche périodique $\tau_i(C_i, T_i, s_i)$ est caractérisée par une durée d'exécution pire-cas C_i, une période T_i, une échéance relative égale à sa période, et un paramètre de pertes s_i, $2 \leq s_i \leq \infty$. Celui-ci fournit la tolérance de la tâche à manquer ses échéances. Introduit par Koren et Shasha dans [58], le modèle Skip-Over sera décrit en détail dans la section suivante car au cente du nouveau modèle que nous proposons.

Le modèle (m, k)-firm

La notion de contrainte (m, k)-firm a été introduite par Hamdaoui et Ramanathan [50]. L'ordonnancement à garantie (m, k)-firm est une technique de gestion de surcharge qui consiste à écarter l'exécution d'un certain nombre d'instances de tâches. Chaque tâche τ_i de l'ensemble est alors caractérisée par deux paramètres m_i et k_i, où m_i représente le nombre minimum d'instances qui doivent respecter leur échéance dans n'importe quelle fenêtre de k_i instances. Une tâche sous contrainte temps réel (m, k)-firm peut se trouver dans 2 états distincts : *normal* ou *échec dynamique*. Une tâche est dite en état d'échec dynamique à un instant t, si à cet instant il existe plus de $(k - m)$ instances ayant manqué leurs échéances parmi les k dernières instances de la tâche. La présence d'au moins une tâche en état d'échec met le système lui-même en état d'échec.

Le modèle Weakly-Hard

Le modèle dit *weakly-hard* en anglais [12] est assez proche du modèle (m, k)-firm, à la différence qu'il considère des fenêtres glissantes : n instances (éventuellement successifs) parmi m instances successifs respectent (ou ne respectent pas) leur échéance. En ce sens, il s'agit d'une généralisation du modèle précédent. Un inconvénient majeur des algorithmes basés sur le modèle (m, k)-firm est que les contraintes (m, k)-firm sont garanties seulement sur des fenêtres de temps fixées. De plus, ces modèles sont assez restrictifs dans le sens où toutes les tâches sont à échéances sur requêtes et possèdent la même fenêtre m. Les systèmes temps réel weakly-hard peuvent eux tolérer que certaines échéances ne soient pas respectées, pourvu que leur nombre soit borné et garanti hors-ligne.

Le modèle DWCS (Dynamic Window Constrained Scheduling)

Le modèle DWCS [153] a été défini comme discipline de service pour l'ordonnancement de paquets réseau ayant pour but de maximiser le taux d'utilisation du serveur réseau en présence de multiples paquets possédant des contraintes temporelles de type (m, k)-firm. Dans ce modèle étendu de DWCS [149], chaque tâche possède une échéance relative D_i et une tolérance de pertes spécifiée par le rapport x_i d'instances pouvant être abandonnés ou retardées sur toute fenêtre de y_i instances à exécuter pour une tâche τ_i.

Le modèle $(p + i, k)$-firm

Le modèle $(p + i, k)$-firm introduit par C. Montez et J. Fraga dans [92] est une extension du modèle (m, k)-firm incluant quelques unes des propriétés du calcul imprécis [39]. Une tâche possède une partie obligatoire (imprécise) et une partie optionnelle (précise). Selon ce modèle, p exécutions précises et i exécutions imprécises doivent être réalisées sur k activations de la tâche. Une attribution des priorités mais aussi un test d'acceptation de précision permettent de sélectionner la version précise ou imprécise de la tâche qui sera exécutée [39].

Le modèle (m, k)-hard

La notion de contrainte (m, k)-hard a été introduite par Bernat et Burns [12]. Cette contrainte impose que pour chaque tâche au moins m échéances sur k activations successives doivent être respectées, sous peine d'entraîner une défaillance au niveau du système. L'approche consiste à utiliser un schéma d'exécution à double priorité (*Dual priority* [43]) reposant sur un algorithme à réservation de bande de faible overhead, visant à ordonnancer de façon conjointe des tâches critiques (*hard*) et des tâches non critiques (*soft*) où la capacité inutilisée du processeur est réservée à l'exécution des tâches non critiques. Cependant, cette approche nécessite que les contraintes temporelles des tâches soient garanties par des tests d'ordonnançabilité hors-ligne.

Dans le cadre de cette thèse, nous traiterons du problème des fautes temporelles principalement issues de surcharge sur une architecture monoprocesseur pour une application temps réel ferme ou dure.

2.6 Le Mécanisme à Échéance

Nous décrivons ici la méthode la plus connue pour faire face aux fautes temporelles de toutes origines dans les systèmes temps réel durs, le Mécanisme à Échéance.

2.6.1 Description du Mécanisme à Échéance

Le Mécanisme à Échéance (en anglais Deadline Mechanism) a été initialement proposée par Liestman et Campbell [36] [61] et destinée aux systèmes temps réel durs. Elle se base sur la redondance logicielle dynamique passive et s'inspire des blocs de recouvrement [36]. Ce mécanisme convient aussi bien à des défaillances d'origine structurelle (surcharge du processeur) que matérielle (panne d'un capteur).

Chaque tâche se caractérise par deux versions indépendantes : une *version primaire* (en anglais, primary) et une *version de secours* (en anglais, alternate).

La version primaire fournit un résultat de la plus grande précision. Mais on suppose que celui-ci peut entrainer une faute temporelle c'est à dire le non respect de son échéance. En effet, la version primaire d'une tâche temps réel doit terminer son exécution au plus tard à son échéance. Le non respect de cette contrainte peut avoir plusieurs origines déjà évoquées précédemment. Par exemple, la tâche attend une donnée capteur qui n'arrive pas à cause d'une défaillance matérielle de celui-ci et se trouve bloquée indéfiniment. Le système est en état de surcharge de traitement et une faute temporelle se produit car le primaire ne peut terminer son exécution avant échéance. La durée d'exécution pire cas de la version primaire peut ne pas être connue. [25], [27].

La version secondaire fournit un résultat de moindre précision. Elle utilise un algorithme de plus petite complexité et son temps d'exécution pire cas est toujours connu et très inférieur à celui de sa version primaire. Par hypothèse, l'exécution de cette version ne conduit à aucune faute logicielle et temporelle. En d'autres termes, il s'agit d'une version fiable tant du point de vue fonctionnel que temporel.
De façon à implémenter correctement le mécanisme à échéance au sein d'un système d'exploitation temps réel, il devient nécessaire de concevoir une stratégie d'ordonnancement adéquate pour tenir compte de la présence de ces deux versions par tâche.

2.6.2 Implémentation pour le temps réel dur

A notre connaissance, le mécanisme à échéance n'a été étudié que sous l'hypothèse de systèmes temps réel durs : pour chaque instance de tâche périodique, au moins une des deux versions *doit réussir* c'est à dire terminer avant échéance. Autrement dit, cent pour cent des échéances sont satisfaites soit par des primaires soit par des secondaires.

L'objet d'un algorithme d'ordonnancement revient à déterminer à tout instant quelle version de quelle tâche à exécuter. Il faut donc concevoir :

- une règle pour ordonner les versions primaires entre elles

- une règle pour ordonner les versions secondaires entre elles
- une règle pour définir quant interrompre l'exécution d'un primaire au profit d'un secondaire ou vice versa.

Deux techniques d'ordonnancement peuvent s'appliquer :

- *Ordonnancement de Première Chance :* exécuter systématiquement la version secondaire d'abord pour se garantir un résultat même si de moindre qualité. Puis tenter l'exécution de la version primaire.
- *Ordonnancement de Dernière Chance :* tenter d'exécuter d'abord les primaires tout en se garantissant l'exécution réussie de tout secondaire dont le primaire a échoué.

2.6.3 Ordonnancement de la Première Chance

Selon cette approche, le résultat de calcul d'une version secondaire doit être gardé jusqu'à savoir si la version primaire est exempt de faute temporelle.
Dans le cas où le primaire a réussi, son résultat sera utilisé plutôt que celui de secondaire car de meilleure qualité. Cette approche possède l'avantage d'une implémentation simple : tous les secondaires sont dans une liste ordonnée selon EDF, DM ou RM par exemple. Lorsque celle-ci devient vide, on exécute les primaires. En d'autres termes, tout secondaire a une priorité supérieure à tout primaire. L'inconvénient réside dans l'exécution inutile des secondaires dont les primaires ont réussi. Et donc au lieu de régler un problème de surcharge, on rajoute de la surcharge au système.

Illustration : Considérons un ensemble de trois tâches : deux tâches classiques mono-versions τ_1 et τ_2 et une tâche τ_3 comportant une version primaire et une version secondaire. La version primaire de τ_3 a des temps de calcul qui diffèrent à chaque instance. Considérons pour les premières instances de τ_3, que les temps d'exécution sont successivement $(4, 4, 6, 6)$.
On note C_i^p et C_i^a la durée d'exécution du primaire respectivement du secondaire de τ_i. La caractérisation de cet ensemble de tâches se trouve dans le tableau suivant : Ici, τ_1 et τ_2 ont des secondaires qui se confondent

Task	r_i	T_i	C_i^p	C_i^a
τ_1	0	16	2	0
τ_2	0	32	6	0
τ_3	0	8	4,4,6,6	2

TABLE 2.2 – Tâches à double versions.

à leurs primaires. L'ordonnancement est basé sur RM pour les trois tâches : τ_1, τ_2 et secondaires de τ_3^i.

La technique de la Première Chance conduit à la séquence d'ordonnancement présentée sur la figure suivante :

L'exécution des quatre instances de τ_3 donne lieu aux remarques suivantes :
Instance 1 : Il n' y a pas de temps libre pour exécuter la version primaire de τ_3.
Instance 2 : La version primaire de τ_3 a réussi.
Instance 3 : Il n' y a pas un temps libre suffisant pour réussir le primaire de τ_3.
Instance 4 : La version primaire de τ_3 a réussi.

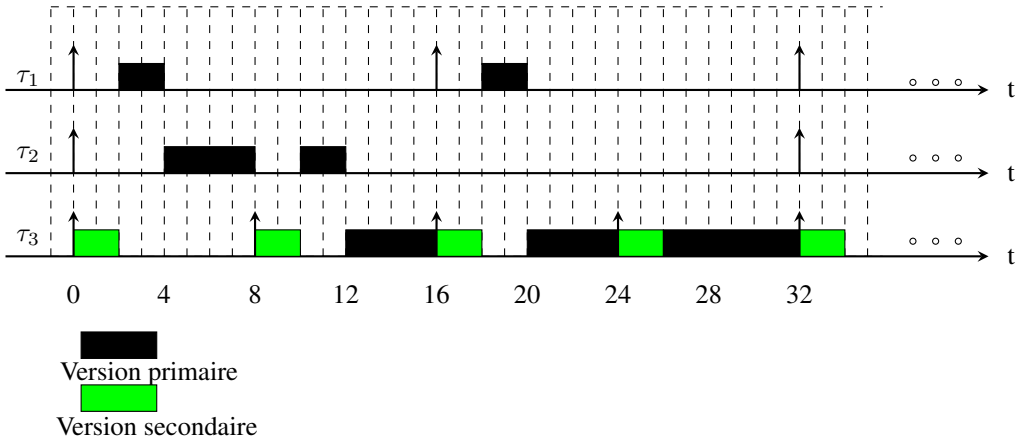


FIGURE 2.5 – Illustration de la stratégie de la Première Chance

2.6.4 Ordonnancement de la Dernière Chance

La stratégie d'ordonnancement de la Dernière Chance a été proposée par Campbell et al dans [36] [61] puis développée par Chetto et Chetto dans [25], [27].
Selon cette approche, un ordonnanceur tel que EDF, RM ou autre ordonne les versions secondaires entre elles mais ces versions sont exécutées le plus tard possible tout en garantissant le respect des échéances. Cet ordonnancement est appelé EDL (Earliest Deadline as Late as Possible) lorsque les versions sont ordonnancées par EDF au plus tard. Les auteurs montrent dans [25] qu'en pratiquant de cette manière, on maximise le temps du processeur pour exécuter les primaires au plus tôt. Les primaires sont ordonnancés ente eux au plus tôt selon une règle d'ordonnancement tel que EDF, RM ou autre. Et les intervalles de temps réservés à leur exécution est mémorisée dans une table. Celle ci est mise à jour chaque fois qu'un primaire réussit car l'exécution de son secondaire devient inutile.
Ainsi, chaque secondaire peut interrompre l'exécution d'une version primaire à n'importe quel moment, pour qu'elle commence son exécution à son temps réservé, et ceci afin de respecter l'échéance.

Cette technique présente l'avantage de n'introduire une charge supplémentaire de traitement que lorsque cela s'avère nécessaire. Toutefois cette méthode engendre des surcoûts d'implémentation plus importants. Ces surcoûts sont de deux ordres :

- surcoûts d'exécution de l'ordonnanceur : implémenter l'ordonnanceur EDL suppose de maintenir à jour cette séquence pour connaître l'instant de démarrage du prochain secondaire en tenant compte des échecs et des succès des primaires

- surcoûts mémoire : l'ordonnanceur EDL nécessite de la place mémoire pour y stocker les intervalles de temps réservés à l'exécution des secondaires. La longueur de cette table dépend de la longueur de l'hyper-période.

Illustration : Considérons un ensemble de trois tâches : deux tâches classiques $\tau_1(0, 4, 16, 16)$ et $\tau_2(0, 6, 32, 32)$ et une tâche τ_3 avec une version primaire et une version secondaire similaires à l'exemple précédent. Les paramètres classiques de la version secondaire de τ_3 sont fixés comme $(0, 2, 8, 8)$.
Les durées d'exécution effective des primaires de τ_3 sont différents à chaque instance. On suppose qu'ils sont donnés par $(4, 4, 6, 6)$ pour les quatre premières instances.

En ordonnançant l'ensemble de tâches données dans le tableau précédent selon la stratégie de la Dernière Chance avec RM, nous obtenons la séquence d'ordonnancement présentée sur la figure suivante :

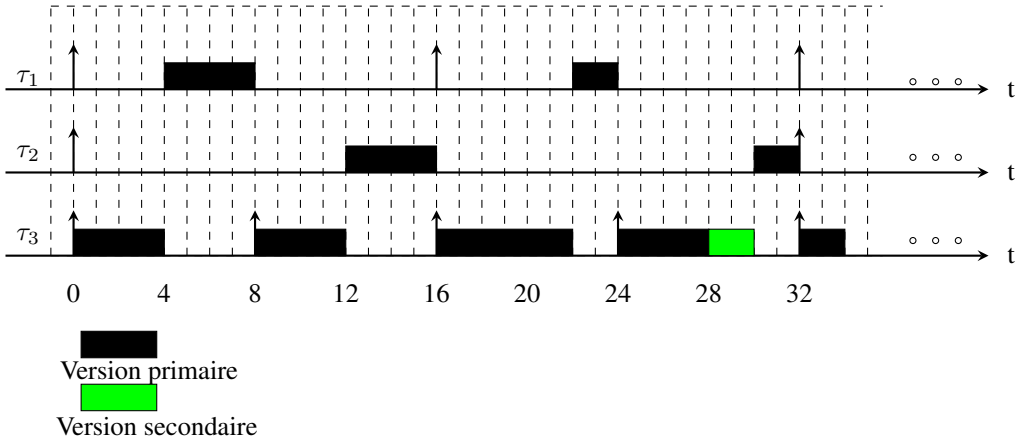


FIGURE 2.6 – Illustration de la stratégie de la Dernière Chance

L'exécution des quatre instances de τ_3 donne lieu aux remarques suivantes :
Instance 1 : la version secondaire de τ_3 ne s'exécute pas car sa version primaire a réussi.
Instance 2 : la version secondaire de τ_3 ne s'exécute pas car sa version primaire a réussi.
Instance 3 : la version secondaire de τ_3 ne s'exécute pas car sa version primaire a réussi.
Instance 4 : La version primaire de τ_3 est préemptée parce qu'il n' y a pas un temps suffisant pour son exécution et sa version secondaire est exécutée.
D'une manière générale, la stratégie de la Dernière Chance est meilleure par rapport à celle de la première chance en terme de qualité de service. Dans un système temps réel dur, la QdS se mesure d'abord par le pourcentage de primaires réussis.

Jusqu'à aujourd'hui, le Mécanisme à Échéance a été implémenté dans plusieurs systèmes d'exploitation temps réel dont en particulier CLEOPATRE. Cet O.S. propose des composants logiciels à source ouvert incluant le Mécanisme à Échéance. Les études portant sur le Mécanisme à Échéance ont uniquement concerné des systèmes temps réel durs.

Nous proposons dans cette thèse d'étendre l'utilisation du Mécanisme à Échéance pour gérer les fautes temporelles engendrées par des surcharges ou des blocages permanents liés à des pannes de capteurs par exemple, et ce pour les systèmes temps réel fermes.

2.7 Le modèle Skip-over

2.7.1 Description

La méthode Skip-over initialement conçue par Koren et Shasha [58] est une méthode de tolérance aux fautes temporelles pour des tâches périodiques de type temps réel ferme. Cette méthode anticipe une surcharge de traitement en choisissant judicieusement l'identité des instances de tâche autorisées à produire une faute temporelle. En d'autres termes, une diminution de la charge s'obtient par la non exécution de certaines instances de tâches choisies convenablement à partir de spécifications données par l'utilisateur.

Selon cette approche, toute instance de tâche possède l'une des deux couleurs : rouge ou bleue. Une instance rouge signifie qu'elle doit être exécutée impérativement avant son échéance, alors qu'une instance bleue peut être éliminée à tout moment provoquant ainsi une faute temporelle car ne produisant aucun résultat.
Une tâche se caractérise par ses paramètres temporels habituels à savoir, une durée d'exécution pire cas C_i, une période T_i, une échéance relative D_i et en plus un nouveau paramètre appelé *paramètre de perte* (en anglais, skip factor) s_i, $2 \leq s_i \leq \infty$. Cette valeur représente la distance minimale autorisée entre deux pertes consécutives d'une même tâche. Une tâche peut ainsi avoir un facteur de perte différente d'une autre tâche en fonction de l'importance pour l'application à disposer de résultats de calcul aussi récents que possible.
Le paramètre de perte renseigne sur la qualité de service minimale requise pour un fonctionnement jugé acceptable même si celui-ci est dégradé. Dans le pire cas, chaque tâche τ_i exécutera seulement $s_i - 1$ instances toutes les s_i périodes. En effet, la distance entre deux pertes consécutives d'une tâche est obligatoirement supérieure ou égale à s_i périodes.

Nous notons que le modèle skip-over présente l'avantage de modéliser une application temps réel dure pour laquelle aucune faute temporelle n'est tolérée. Cent pour cent des instances doivent s'exécuter dans le respect des échéances. Pour cela, il suffit de fixer s_i à l'infini pour interdire toute perte pour chacune des tâches.

Règles sur s_i :

- Quand une instance bleue d'une tâche produit une faute temporelle c'est à dire ne s'exécute pas avec succès avant échéance, les $(s_i - 1)$ instances suivantes sont nécessairement rouges c'est à dire devront impérativement s'exécuter avec succès.

- Quand une instance bleue d'une tâche est complètement exécutée avant son échéance, l'instance suivante devient bleue.

- Les $(s_i - 1)$ premières instances de chaque tâche sont supposées rouges.

2.7.2 Analyse d'ordonnançabilité

En environnement temps réel ferme il est indispensable de pouvoir garantir un fonctionnement respectant les spécifications caractérisées par le facteur de perte de chaque tâche. Les théorèmes suivants nous donnent des résultats très utiles pour effectuer une analyse d'ordonnançabilité.

Theorem 3 *[58] Un ensemble des tâches périodiques indépendantes $\tau = \{\tau_i(C_i, T_i, s_i), 1 \leq i \leq n\}$ conformes au modèle skip-over est ordonnançable seulement si*

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

Ce théorème signifie tout simplement que le taux d'utilisation dans la situation pire cas de pertes doit être inférieur à la capacité de traitement du processeur, soit 1.
Le théorème suivant introduit par Caccamo et Buttazzo [22], donne une condition suffisante d'ordonnançabilité basée sur la demande processeur : Dans tout intervalle de temps, la demande processeur pire cas requise par les instances rouges doit être inférieure ou égale à la longueur de cet intervalle.

Theorem 4 *[22] Un ensemble des tâches périodiques indépendantes* $\tau = \{\tau_i(C_i, T_i, s_i)1 \leq i \leq n\}$ *conformes au modèle skip-over est ordonnançable si* $\forall L \geq \sum_{i=1}^{n} D(i,[0,L])$ *avec*

$$D(i,[0,L]) = (\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{T_i.s_i} \rfloor)$$

Le théorème suivant fournit une condition nécessaire et suffisante d'ordonnançabilité en introduisant un facteur d'utilisation U_p^* appelé *facteur d'utilisation équivalent* qui tient compte des pertes.

Theorem 5 *[22] Un ensemble des tâches périodiques indépendantes* $\tau = \{\tau_i(C_i, T_i, s_i)1 \leq i \leq n\}$ *conformes au modèle skip-over est ordonnançable si et seulement si* $U_p^* \leq 1$ *où :*

$$U_p^* = min_{L \geq 0} \sum_{i=1}^{n} \frac{D(i,[0,L])}{L}$$

2.7.3 Stratégies d'ordonnancement

Les concepteurs de ce modèle ont proposé deux algorithmes d'ordonnancement basiques appelés respectivement RTO (Red Tasks Only) et BWP (Blue When Possible).

Stratégie d'ordonnancement RTO

Selon l'algorithme RTO [58], on exécute seulement les instances rouges et donc toutes les instances bleues sont systématiquement rejetées. On peut donc dire que RTO fournit une QdS effective égale à la QdS spécifiée c'est à dire conduit au plus mauvais comportement tolérable en termes d'échéances violées.
Les instances rouges sont ordonnées entre elles par l'algorithme d'ordonnancement EDF.
Illustration : Considérons l'ensemble de tâches donné par le tableau suivant : La séquence d'ordonnance-

Task	r_i	T_i	C_i	s_i
τ_1	0	3	1	4
τ_2	0	4	2	3
τ_3	0	12	5	∞

TABLE 2.3 – Tâches acceptant des pertes.

ment RTO est présentée sur la figure suivante :

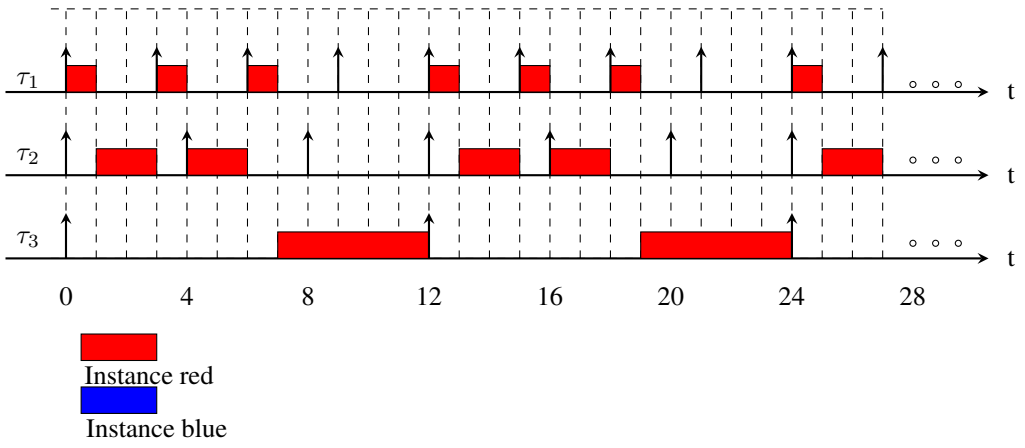


FIGURE 2.7 – Illustration de l'ordonnancement RTO

Bien que RTO présente une implémentation très aisée, cette stratégie ne donne pas satisfaction puisque conduisant à une QdS juste acceptable.

Stratégie d'ordonnancement BWP

Dans [58], les auteurs proposent un autre algorithme d'ordonnancement appelé BWP (Blue When Possible) plus performant que le précédent : On tente l'exécution des instances Bleues chaque fois qu'aucune instance rouge ne demande à s'exécuter. Autrement dit, BWP exécute les instances bleues en arrière-plan par rapport aux instances rouges.
Illustration : Considérons un ensemble de tâches dont les paramètres sont donnés par le tableau suivant : La séquence d'ordonnancement BWP est illustrée sur la figure suivante :

Task	r_i	T_i	C_i	s_i
τ_1	0	3	1	3
τ_2	0	4	2	3
τ_3	0	12	5	∞

TABLE 2.4 – Ensemble de tâches avec paramètres des pertes.

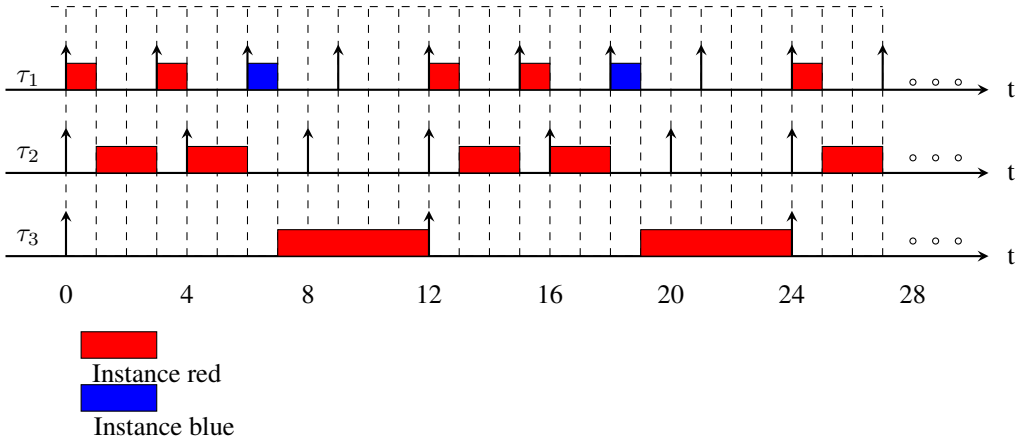


FIGURE 2.8 – Illustration de l'ordonnancement BWP

Stratégie d'ordonnancement RLP

Dans l'objectif de surmonter le grand inconvénient de BWP qui concerne le gaspillage du temps processeur chaque fois qu'on interrompt définitivement des instances bleues quasi au terme de leur exécution, dans [87] on propose deux solutions améliorantes.
La première proposition s'appelle RLP (Red instances as Late as Possible). Elle consiste à appliquer l'algorithme d'ordonnancement EDL (Earliest Deadline as Late as possible) aux instances rouges leur garantissant une exécution au plus tard sans fautes temporelles. La technique consiste à

- réserver les intervalles d'exécution au plus tard des instances rouges grâce à EDL (voir chapitre précédent),
- utiliser les intervalles de temps processeur laissés disponibles pour exécuter les instances bleues au plus tôt,
- chaque fois qu'une instance bleue réussit, on met à jour la séquence EDL,
- dans la situation particulière où aucune instance bleue ne demande son exécution, les instances rouges sont exécutées non pas au plus tard mais au plus tôt.

On montre qu'avec cette méthode, on maximise la capacité du processeur dédiée aux instances bleues et on optimise la QdS résultante.

Stratégie d'ordonnancement RLP/T

Dans le but d'améliorer les performances en terme de QdS, dans [87], on propose un autre algorithme appelé RLP/T. Son principe est de mettre en oeuvre un test d'acceptation avant tout lancement d'une instance bleue.
L'objectif de ce test est de garantir l'exécution à terme de chaque instance et par conséquent diminuer le gaspillage de temps processeur constaté dans l'algorithme RLP dû aux instances partiellement exécutées ne produisant aucun résultat. Les instances rouges sont acceptées systématiquement pour être mises dans la liste prête d'exécution.
Le test d'acceptation repose sur le calcul exact du temps processeur disponible à l'arrivée de chaque instance bleue. Cela est fait toujours comme dans l'algorithme RLP par l'utilisation de la théorie démontrée dans les travaux [25] [26] et par la prise en considération des instances rouges déjà prêtes et les instances bleues déjà acceptées.
Les résultats de simulations montrent la supériorité de RLP/T en termes de QdS mais ce, au détriment de la place mémoire requise et des overheads d'exécution de l'ordonnanceur.

2.7.4 Commentaires

Le modèle Skip-Over présente de nombreux avantages : il permet de prendre en compte la consécutivité des fautes temporelles, il permet la modélisation de tâches à contraintes dures ($s_i = \infty$) et des ordonnanceurs sophistiqués comme RLP/T nous permettent d'optimiser la qualité de Service.

2.8 Conclusion

Nous avons restreint nos rappels aux fautes temporelles manifestées par le viol des échéances associées aux fins d'exécution normales des tâches temps réel.

Pour pallier à l'occurrence des fautes temporelles, des approches d'ordonnancement et des modèles ont été proposés. L'utilisation de ces derniers va dépendre de l'origine des fautes temporelles prises en considération.

L'état de l'art présenté dans cette partie a permis de mettre en évidence différents modèles et approches d'ordonnancement adaptés aux systèmes temps réel fermes tolérants aux fautes temporelles en particulier lors de surcharges temporaires. Ces approches choisissent en fonction des spécifications données hors-ligne l'identité des instances de tâches autorisées à subir des fautes temporelles (autrement dit, les tâches en échec vis à vis de leur échéance). Au lieu que ces fautes temporelles se produisent de façon indéterministe et non contrôlée, on les choisit adéquatement. On diminue dynamiquement de manière astucieuse la charge infligée au processeur. Cela peut se faire par :

- **la suppression de certaines instances de tâches** ce qui engendre des fautes temporelles vu l'absence de résultats de sortie pour ces instances. **C'est le principe du modèle skip-over**. Cette approche convient plus particulièrement aux systèmes temps réel fermes puisque par définition, un système temps réel dur impose la production d'au moins un résultat de calcul par instance de tâche.

- **l'exécution incomplète de certaines instances de tâches** avec la production de résultats de sortie de moindre qualité.La tâche s'exécute partiellement mais il y a production d'une donnée de sortie avant échéance ce qui garantit un mode de fonctionnement dégradé.

    1. On garantit l'exécution d'une partie de code minimale, la partie obligatoire. Ainsi un fonctionnement dégradé est assuré même en présence de fautes temporelles intervenant lorsque l'autre partie de code ne peut se terminer. **C'est le principe du modèle Calcul Imprécis**. Cette méthode s'applique aussi bien dans un contexte temps réel dur (pas de partie optionnelle) que temps réel ferme. Intéressante, cette approche ne se prête qu'à des tâches implémentées sous forme d'algorithmes itératifs, ce qui limite considérablement le champ applicatif (traitement d'image, cryptographie par exemple).
    2. Pour chaque instance de tâche, au moins un résultat est produit. celui-ci est soit normal lorsque produit par l'algorithme primaire (absence de faute temporelle), soit de qualité dégradée lorsque produit par l'algorithme de secours (présence d'une faute temporelle). **C'est le principe du Mécanisme à Échéance.** Cette technique de tolérance aux fautes temporelles a donc un champ plus élargi d'application que le Calcul Imprécis. De plus, il permet de faire face également à des fautes logicielles. Jusqu'ici, ce mécanisme n'a été utilisé que dans un contexte de temps réel dur.

Nous proposons dans la partie suivante de cette thèse, un nouveau modèle, BGW, basé sur le modèle skip-over et sur le principe du Mécanisme à Echéance pour gérer de façon contrôlée et anticipée l'occurrence de fautes temporelles.

Contribution à la modélisation de tâches temps réel

3

BGW : un nouveau modèle de tâche

3.1 Présentation du chapitre

Dans ce chapitre, nous allons proposer un modèle de tâche plus général que ceux présentés dans la partie État de l'Art. Ce modèle appelé BGW(Black Grey White) permet de prendre en compte trois niveaux de qualité de service fournis par une instance de tâche périodique :

- Dans un *fonctionnement en mode normal*, l'instance s'exécute et fournit un résultat de haute précision issu de l'exécution de sa version primaire,
- Dans un *fonctionnement en mode semi-dégradé*, l'instance fournit un résultat de moindre précision issu de l'exécution de sa version de secours,
- Dans un *fonctionnement en mode dégradé*, l'instance ne fournit aucun résultat.

Ce chapitre décrit ce nouveau modèle que l'on peut décliner en trois variantes respectivement appelées BGW1, BGW2 et BGW3.

3.2 Le modèle

Notre objectif premier est de proposer un nouveau modèle qui permet de caractériser avec davantage de précisions la Qualité de Service minimale que l'on attend d'un ordonnancement de tâches périodiques. En d'autres termes, nous souhaitons par un unique modèle intégrer :

- la présence de deux versions par tâche. La première fournit un résultat de très grande qualité alors que la seconde fournit un résultat de moindre qualité mais avec un temps d'exécution beaucoup plus court.
- la possibilité sous certaines conditions d'espacement temporel de n'exécuter aucune version, d'exécuter soit la version primaire ou la version secondaire ou finalement de devoir impérativement exécuter la version primaire. Ces espacements temporels seront caractérisés par deux grandeurs.

Nous proposons donc un nouveau modèle qui s'inspire à la fois du mécanisme à échéance et du modèle skip-over présentés dans le chapitre précédent.
Nous supposons que ce modèle ne s'applique qu'à des tâches périodiques (voire sporadiques). Chaque tâche consiste en un ensemble infini d'instances qui s'exécutent donc avec une périodicité exacte (ou bornée par une valeur minimale).

A un instant donné, une instance de tâche se caractérise par sa couleur, à savoir *Black*, *Grey* ou *White*. La couleur d'une instance indique la contrainte qui porte sur son exécution. Ainsi, une instance est

- **black** si l'on doit impérativement exécuter sa version primaire avant échéance
- **Grey** si l'on doit exécuter au moins une version c'est à dire soit sa version secondaire, soit sa version primaire.
- **White** si l'on peut ne pas l'exécuter c'est à dire la supprimer.

On suppose que pour toute tâche, la durée d'exécution de la version secondaire est inférieure à celle de la version primaire.
Nous avons donc un modèle qui caractérise pour chaque tâche :

- la fréquence minimale à laquelle cette tâche peut ne produire aucun résultat,
- la fréquence minimale à laquelle cette tâche peut ne produire qu'un résultat de moindre qualité même si acceptable,
- la fréquence minimale à laquelle cette tâche doit produire un résultat de la plus grande qualité.

3.3 Ordonnancement sous le modèle BGW

3.3.1 Objectif d'un algorithme d'ordonnancement

Tout ordonnanceur destiné à des tâches BGW a pour objectif de garantir le respect des contraintes exprimées par les paramètres de QdS. Par exemple, concernant BGW1, il s'agira de produire une séquence où pour chaque tâche τ_i, au moins une version primaire sur n_i instances est exécutée avec succès ainsi qu'au moins une version secondaire sur l_i instances.

A tout instant, l'ordonnanceur doit donc construire la séquence de sorte que :

1. l'échéance de toute instance Black soit respectée par la version primaire,
2. l'échéance de toute instance Grey soit respectée par, soit la version primaire, soit la version secondaire,
3. le nombre total de pertes soit le plus petit possible,
4. le nombre total de versions primaires réussies soit le plus grand possible,
5. ou le nombre total de versions (primaires et secondaires) soit le plus grand possible.

3.3.2 Mise en oeuvre d'un ordonnanceur

Mettre en oeuvre un ordonnanceur pour gérer des tâches BGW suppose d'implémenter plusieurs listes car différents types d'instances peuvent se trouver à l'état prêt. A tout instant, chaque instance de tâche réveillée et non terminée est soit White, soit Grey, soit Black.

Nous sommes donc amenés à définir une règle d'ordonnancement qui doit tenir compte de deux critères :

- **l'urgence** d'une instance matérialisée par son échéance absolue,
- **et l'importance** d'une instance, matérialisée par sa couleur qui nous indique le niveau de dégradation autorisée (Black : pas de dégradation autorisée, Grey : dégradation partielle autorisée, White : dégradation totale autorisée c'est à dire exécution optionnelle de l'instance)

3.4 Secteurs d'utilisation du modèle BGW

Différents domaines d'applications temps réel sont concernés par le modèle BGW :

- **Les applications médicales.** Considérons un système de surveillance d'un patient où l'une des tâches est responsable de la mesure et de l'affichage périodiques de ses principales données physiologiques (température du corps, fréquence cardiaque, pression artérielle, niveau de glucose dans le sang,...). La version primaire de cette tâche sera le programme qui effectuera la lecture et l'affichage de tous les paramètres. Alors que l'on définira la version de secours comme le programme qui effectuera la lecture et l'affichage de la donnée physiologique considérée comme la plus importante à savoir la fréquence cardiaque. Cette version de secours aura une durée d'exécution réduite et x fois inférieure à celle de version primaire en charge de x données physiologiques.

- **Les applications industrielles.** Imaginons un véhicule auto-guidé qui est instrumenté à l'aide de caméras et de plusieurs capteurs de proximité. Nous pouvons alors considérer les deux versions comme suit : La version primaire prend en compte les images délivrées par la caméra et les données délivrées par les capteurs pour calculer sa trajectoire. L'algorithme est ainsi bien plus compliqué mais conduit à un calcul de la meilleure trajectoire. Alors que la version secondaire s'appuiera uniquement sur des données capteur pour construire une trajectoire à l'aide d'un algorithme plus simple et donc plus rapide.

- **Les applications temps réel de service.** En général, le modèle BGW pourra être utilisé dans tout système temps réel dont les tâches peuvent être codées de deux façons différentes, la première conduisant à un service de la meilleure qualité et la seconde un service de moindre qualité mais exigeant moins de temps de traitement. Ces applications de service concernent par exemple le multimédia, la visioconférence, ...

Plusieurs variantes du modèle BGW vont être décrites dans le paragraphe suivant. Ces variantes diffèrent entre elles selon la nature des contraintes d'exécution que l'on peut exiger des instances d'une même tâche périodique.

3.5 Variante BGW1

3.5.1 Présentation

Dans la variante BGW1, chaque tâche τ_i est caractérisée par les deux paramètres de Qualité de Service définis comme suit :

- n_i : La distance maximale (nombre maximal de périodes) autorisée entre deux exécutions successives de la version primaire.
- l_i : La distance maximale entre deux exécutions réussies, que ce soit par sa version primaire ou par sa version secondaire.

Par conséquent, chaque tâche τ_i du modèle BGW1 est caractérisée par (voir figure 3.1) :
$(r_i, C_i^a, C_i^p, D_i, T_i, n_i, l_i)$ avec

- r_i,D_i et T_i les paramètres temporels comme définis dans le chapitre précédent
- C_i^a et C_i^p la durée d'exécution pire cas de la version de secours et celle de la version primaire avec $C_i^a \leq C_i^p$
- n_i et l_i les deux paramètres de QdS définis ci-dessus avec $0 \leq l_i \leq n_i$

Notons que $\{B\}$, $\{G\}$ et $\{W\}$ signifient respectivement instance Black, Grey et White sur la figure 3.1.

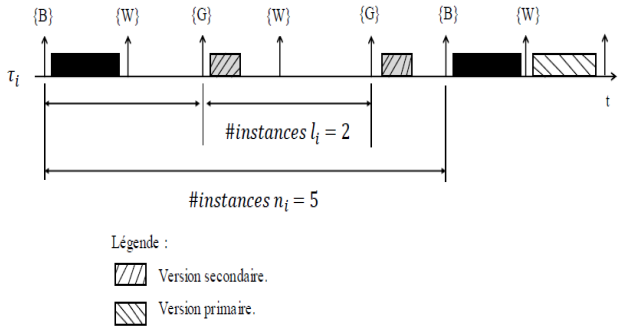


FIGURE 3.1 – Illustration des paramètres du modèle BGW1.

3.5.2 Test de faisabilité pour BGW1

Pour démontrer la faisabilité d'une application temps réel où les tâches respectent le modèle BGW1, nous proposons un test. Rappelons que dans le cas du modèle classique d'une application temps réel dur, le test revient à calculer que le taux d'utilisation donné par la somme des taux d'utilisation de chaque tâche puis à vérifier que celui-ci est inférieur à un. Nous montrons comment modifier un tel test pour l'adapter au modèle BGW1, ce qui revient à calculer pour BGW1 le facteur d'utilisation minimale de chaque tâche.

Theorem 6 *Un ensemble Γ de tâches périodiques conformes au modèle BGW1 est faisable sur un système monoprocesseur seulement si $\sum_{i=1}^{n} U_{\tau_i} \leq 1$ avec $U_{\tau_i} = \frac{1}{n_i.T_i}.C_i^p + \frac{1}{n_i.T_i}.\lfloor \frac{n_i-1}{l_i} \rfloor.C_i^a$.*

Preuve.
Par la définition du modèle BGW1, la situation pire cas se produit lorsque le système exécute pour chaque tâche τ_i, dans une période de n_i instances successives, une version primaire et $\lfloor \frac{n_i-1}{l_i} \rfloor$ versions secondaires. Autrement dit, la partie $\frac{1}{n_i.T_i}.C_i^p$, correspond au facteur d'utilisation du nombre d'instances Black à exécuter pour chaque tâche τ_i, selon la contrainte n_i du modèle BGW1 pour l'exécution des versions primaires.
Le nombre $\frac{1}{n_i.T_i}.\lfloor \frac{n_i-1}{l_i} \rfloor$ correspond au nombre d'instances Grey qui doivent exécuter au moins leurs versions secondaires selon la contrainte d'exécutions du modèle BGW1. D'où, la grandeur $\frac{1}{n_i.T_i}.\lfloor \frac{n_i-1}{l_i} \rfloor.C_i^a$ qui correspond au facteur d'utilisation pire cas de ce nombre d'instances à exécuter selon la contrainte de l_i de BGW1.
Nous commençons l'exécution par une version primaire pour chaque tâche. Ensuite, nous devons exécuter une version secondaire dans chaque $l_i - 1$ instance successive et ceci jusqu'à rencontrer la prochaine version primaire à exécuter obligatoirement selon la contrainte donnée par n_i.
Lorsqu'il y a une occurrence simultanée des contraintes de n_i et de l_i, nous devons prendre en considération seulement celle de n_i.

Finalement, vu que le facteur d'utilisation total d'un système monoprocesseur ne doit pas excéder 100%, le théorème est alors prouvé.

3.5.3 Cas particuliers

En fonction des valeurs des paramètres n_i et l_i, une tâche BGW1 peut être de type temps réel dur ou ferme.

- Si $l_i = 1$ et $n_i = 1$, τ_i est une tâche temps réel dur qui n'accepte aucune dégradation, ni qualitative (résultats produits de la plus grande qualité), ni quantitative (pas de pertes).
- Si $l_i = 1$ et $n_i = \infty$, τ_i est une tâche temps réel dur qui accepte une dégradation qualitative, mais pas une dégradation quantitative (toutes les instances sont Grey c'est à dire produisent au moins un résultat).

Remarque :
Il est à noter qu'on peut avoir un logiciel d'application qui contient des tâches de type temps réel dur n'admettant aucune perte mais admettant éventuellement une dégradation qualitative des résultats et des tâches de type temps réel ferme admettant des pertes.

3.5.4 Exemple illustratif

Illustrons la modélisation et la condition d'ordonnançabilité des tâches BGW1, par l'exemple donné dans le tableau suivant : Le facteur d'utilisation de Γ est donné par :

Task	T_i	C_i^p	C_i^a	l_i	n_i
τ_1	6	3	2	2	4
τ_2	10	4	1	3	4
τ_3	15	6	2	1	3

TABLE 3.1 – Exemple illustratif du modèle BGW1.

Pour τ_1 : $\frac{1}{n_1}.\frac{C_1^p}{T_1} + \frac{1}{T_1.n_1}.\lfloor \frac{n_1-1}{l_1} \rfloor.C_1^a$, d'où $U_{\tau_1} = 0.2083$.

Pour τ_2 : $\frac{1}{n_2}.\frac{C_2^p}{T_2} + \frac{1}{T_2.n_2}.\lfloor \frac{n_2-1}{l_2} \rfloor.C_2^a$, d'où $U_{\tau_2} = 0.125$.

Pour τ_3 : $\frac{1}{n_3}.\frac{C_3^p}{T_3} + \frac{1}{T_3.n_3}.\lfloor\frac{n_3-1}{l_3}\rfloor.C_3^a$, d'où $U_{\tau_3} = 0.1777$.

Le facteur d'utilisation total U_Γ de l'ensemble de tâches Γ est donné par $U_{\tau_1}+U_{\tau_2}+U_{\tau_3}$, soit $U_\Gamma = 0.511$. Nous en déduisons que Γ est un *ensemble de tâches BGW1 ordonnançable*. De plus, nous remarquons clairement qu'une petite augmentation du nombre de versions primaires obligatoires à travers le paramètre n_i conduirait à un système non ordonnançable.
D'une manière identique, faire diminuer le nombre de pertes autorisées pour τ_i en réduisant la valeur de l_i conduirait à la non faisabilité de Γ.

3.6 Ordonnancement basique pour BGW1

Nous décrivons ici l'algorithme d'ordonnancement appelé BGa1 dans lequel nous appliquons les règles suivantes :

- On exécute seulement pour chaque tâche, les instances Black, imposées par la contrainte n_i et les versions secondaires des instances Grey, imposées par la contrainte l_i .
- Les instances Black et les versions secondaires des instances Grey sont ordonnancées entre elles par l'algorithme d'ordonnancement classique EDF.
- Toutes les instances White sont systématiquement rejetées c'est à dire non exécutées.

Cette stratégie associée au modèle BGW1, peut être considérée comme la plus mauvaise stratégie d'ordonnancement acceptable car produisant la Qualité de Service minimale requise.

Condition exacte d'ordonnançabilité de BGa1

Soit $\Gamma = \{\tau_i(C_i, T_i, D_i, n_i, l_i)\}$ un ensemble de tâches BGW1.

Définition : On définit la demande processeur de Γ et on note $DBF_{BGa1}(\Gamma, L)$, par la quantité de temps de traitement requis par l'exécution des instances Black et Grey de Γ dans l'intervalle $[0, L]$ définie par $DBF_{BGa1}(\Gamma, L) = \sum_{i=1}^{n} D(i, [0, L])$ avec

$$D(i,[0,L]) = \left(\lceil\frac{L}{n_i.T_i}\rceil\right).C_i^p + \left(\lfloor\frac{L}{n_i.T_i}\rfloor.\lfloor\frac{n_i-1}{l_i}\rfloor + \lfloor\frac{X}{l_i.T_i}\rfloor\right).C_i^a \tag{3.1}$$

où $X = L - \lfloor\frac{L}{n_i.T_i}\rfloor.n_i.T_i + T_i$

La grandeur $\lceil\frac{L}{n_i.T_i}\rceil$ représente le nombre d'instances Black de τ_i actives et ayant leur échéance dans $[0, L]$. $\left(\lfloor\frac{L}{n_i.T_i}\rfloor.\lfloor\frac{n_i-1}{l_i}\rfloor + \lfloor\frac{X}{l_i.T_i}\rfloor\right)$ représente le nombre d'instances Grey dans $[0, L]$. La somme des deux termes donne le nombre total d'instances Black et Grey qui doivent s'exécuter impérativement dans $[0, L]$.
Ces deux termes multipliés respectivement par C_i^p et C_i^a, nous donnent le temps d'exécution total nécessaire pour l'exécution des instances Black et de la version secondaire des instances grey dans $[0, L]$.

Définition : On définit la *charge processeur équivalente* de Γ par la quantité :

$$Load_1^*(\Gamma) = max_{L\in[D_{min},P)}\frac{DBF_{BGa1}(\Gamma, L)}{L} \tag{3.2}$$

Nous calculons DBF_{BGa1} seulement pour des valeurs de L correspondant aux échéances absolues des instances Black et Grey sur $[0, H(\Gamma)]$.

Nous énonçons alors le théorème suivant :

Theorem 7 *Un ensemble de tâches périodiques Γ conformes au modèle BGW1 est ordonnançable par BGa1 si et seulement si sa charge processeur équivalente ne dépasse pas 1 c'est à dire $Load_1^*(\Gamma) \leq 1$.*

Preuve
Si. Par hypothèse, les instances de tâches se réveillent à l'instant 0 et sont Black. Selon l'algorithme BGa1 associé au modèle BGW1, pour chaque tâche τ_i, chaque instance White est systématiquement ignorée.
Toutes les autres instances (instances Black et Grey) sont exécutées selon EDF. $D(i, [t_1, t_2])$ représente la demande de traitement de τ_i dans l'intervalle $[t_1, t_2]$ selon EDF, à savoir le temps total de traitement nécessaire pour l'exécution de toutes les instances Black et Grey de τ_i qui sont réveillées et doivent s'exécuter dans l'intervalle $[t_1, t_2]$.
Supposons que la condition $Load_1^*(\Gamma) \leq 1$ soit vérifiée et qu'à un certain instant t, une instance manque son échéance.
Soit $t_a \geq 0$ la dernière fois (avant t) où le processeur a été laissé inactif par l'algorithme d'ordonnancement ($t_a = 0$, s'il n'y en a pas). Soit $t_b \geq 0$ la dernière fois (avant t) où le processeur était occupé à exécuter un instance de tâche possédant une échéance après t ($t_b = 0$ s'il n'y en a pas).
Soit $t' = \max(t_a, t_b)$. Le motif de pertes de l'algorithme assure que $D(i, [t', t]) \leq D(i, [0, t - t'])$.

L'instant t' possède la propriété que seules les instances rouges de tâche qui ont été activées après t' avec une échéance absolue inférieure ou égale à t sont exécutées sur $[t', t]$. Il n'y a aucun moment dans $[t', t]$ où le processeur est inactif. Le fait que l'échéance ait été manquée signifie que la demande de traitement $C[t', t]$ au cours de cet intervalle est supérieure à la longueur de l'intervalle. Autrement dit, $t - t' < C[t', t]$. Ce qui nous conduit à,

$$t - t' < C[t', t] = \sum_{i=1}^{n} D(i, [t', t]) \leq DBF_{BGa1}(\Gamma, [0, t - t']) \leq (t - t').Load_1^*(\Gamma) \tag{3.3}$$

et donc $Load1^*(\Gamma) > 1$ qui nous amène à une contradiction.

Seulement si. Supposons que $Load_1^*(\Gamma) > 1$ sur un intervalle $[0, t]$. La charge de traitement $C[0, t]$ nécessaire à l'exécution des instances Black dans le pire cas est $C[0, t] = DBF_{BGa1}(\Gamma, t)$.
Selon la définition de $Load_1^*(\Gamma)$, $\exists t$ tel que $Load_1^*(\Gamma).t = DBF_{BGa1}(\Gamma, t)$.
Par conséquent, $C[0, t] = DBF_{BGa1}(\Gamma, t) = DBF_{BGa1}(\Gamma, t) > t$. Comme le temps de traitement sur $[0, t]$ dépasse la longueur de ce même intervalle alors Γ n'est pas faisable.

Il est suffisant de vérifier la condition du Théorème 7 pour les instants t correspondant aux échéances absolues des instances. De plus, il est inutile de tester cette condition pour des instants supérieurs à $H(\Gamma)$.

Illustration

Nous illustrons ci-dessous l'ordonnancement BGa1 sur des tâches BGW1.

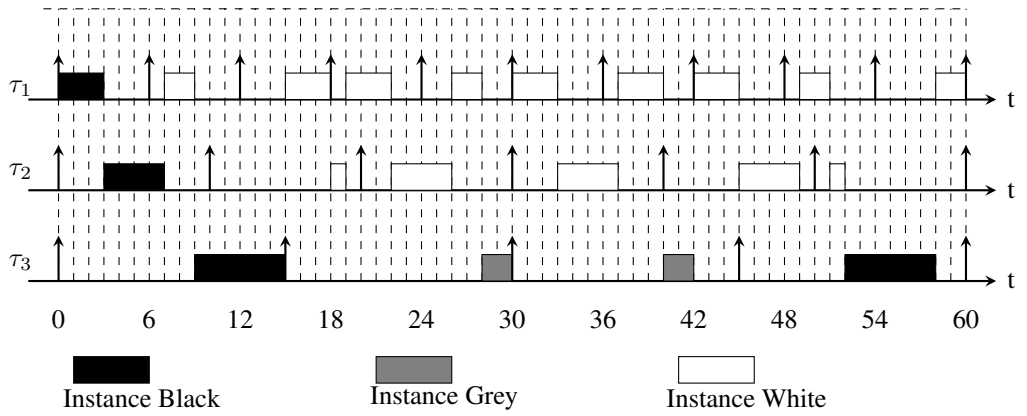


FIGURE 3.2 – Illustration de l'ordonnancement BGa1

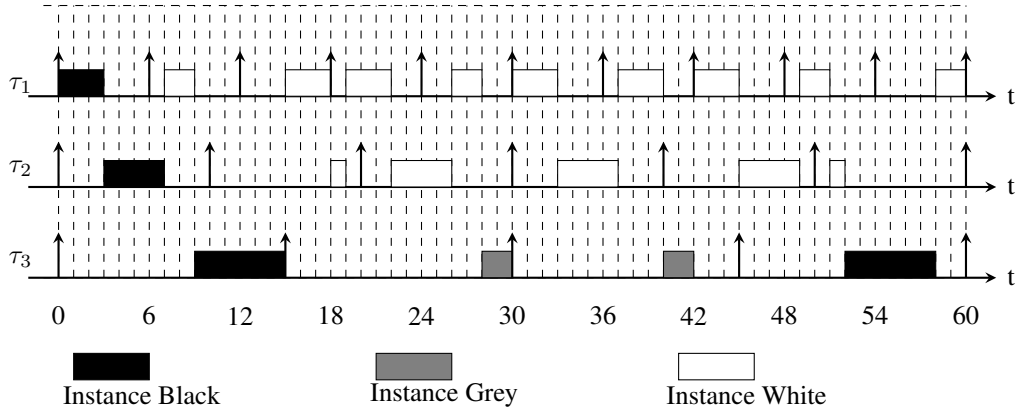


FIGURE 3.3 – Illustration de l'ordonnancement de tâches BGW1

3.7 Variante BGW2

3.7.1 Présentation

Dans le modèle BGW2, chaque tâche τ_i est caractérisée par les deux paramètres de Qualité de Service définis comme suit :

- n_i : La distance maximale autorisée entre deux exécutions successives réussies de la version primaire.
- s_i : Le nombre minimal d'instances devant être exécutées par la version de secours entre deux pertes successives.

3.7.2 Test de Faisabilité pour BGW2

Le test de faisabilité du modèle BGW2 est formulé dans le théorème suivant. La situation pire cas correspond à l'exécution régulière des primaires avec la période constante égale à $n_i.T_i$ et avec une distance entre deux pertes successives égale à la période constante $(s_i - 1).T_i$ unités de temps.

Theorem 8 *Un ensemble Γ de tâches périodiques conformes au modèle BGW2 est faisable sur un système monoprocesseur seulement si $\sum_{i=1}^{n} U_{\tau_i} \leq 1$ avec $U_{\tau_i} = \frac{1}{T_i}.(\frac{1}{n_i}C_i^p + (\frac{s_i-1}{s_i} - \frac{1}{n_i} + \frac{1}{PPCM(n_i,s_i)})C_i^a)$.*

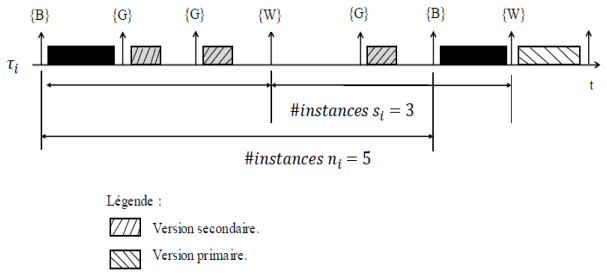


FIGURE 3.4 – Illustration des paramètres du modèle BGW2.

Preuve.
Selon BGW2, la situation pire cas se produit lorsque le système exécute, dans chaque période de s_i instances successives, au moins $s_i - 1$ versions secondaires successives et lorsqu'il exécute une seule version primaire parmi chaque n_i instances successives.

Considérons que le système commence par l'exécution d'une version primaire et par une perte d'une version secondaire. Ensuite, avant la prochaine perte, il doit exécuter s_i versions secondaires.
La première coïncidence entre une version primaire et une perte de version secondaire est supposé être juste au début de l'exécution. La deuxième coïncidence sera à l'instant pour lequel il existe deux nombres entiers k_1 et k_2 qui vérifient l'équation suivante : $k_1.n_i = k_2.s_i$.
Autrement dit, dans chaque période de $PPCM(n_i, s_i).T_i$, nous avons une coïncidence entre une version primaire obligatoire et une perte en versions secondaires.
En plus, nous devons enlever le nombre de répétitions de versions (On ne doit pas considérer la version secondaire lorsqu'il y a une coïncidence avec une version primaire). Ce nombre est égal à $\frac{1}{n_i} - \frac{1}{PPCM(n_i,s_i)}$.
Finalement, en rappelant que la condition nécessaire d'ordonnançabilité est que l'utilisation du processeur ne peut pas dépasser $100\%100$, le théorème est prouvé.

3.7.3 Cas particuliers

Le modèle BGW2 permet de modéliser des tâches de type temps réel dur ou ferme selon la valeur donnée aux paramètres n_i et s_i.

- Si $s_i = \infty$ et $n_i = \infty$, τ_i est une tâche de type temps réel dur qui accepte une dégradation qualitative de QdS (toutes les instances sont Grey) et qui n'accepte pas de dégradations quantitative de QdS (pas de perte).
- Si $s_i = \infty$ et $n_i = 1$, τ_i est une tâche de type temps réel dur qui n'accepte ni des dégradations qualitatives, ni des dégradations quantitatives de QdS (toutes les instances sont Black).
- Si $s_i = \infty$ et $n_i > 1$, τ_i est une tâche de type temps réel dur, avec une exigence sur le nombre de versions primaires à exécuter impérativement (au moins une sur n_i instances). Dans ce cas, les instances sont soit Black, soit Grey.
- Si $s_i > 0$ et $n_i = \infty$, τ_i est une tâche de type temps réel ferme (toutes les instances sont Grey ou White).
- Si s_i a une valeur donnée non nulle et $n_i = 1$, nous avons une contradiction impossible à prendre en compte.

Notons qu'on a une formule particulière simple pour exprimer le facteur d'utilisation total U_Γ dans ce modèle :

- Si $n_i = \infty$ alors $U_\Gamma = \sum_{i=1}^{n} \frac{s_i - 1}{s_i.T_i}.C_i^a$
- Si $l_i = \infty$ alors $U_\Gamma = \sum_{i=1}^{n} \frac{1}{n_i.T_i}.C_i^p$

3.7.4 Exemple illustratif

Nous illustrons le modèle BGW2 et sa condition de faisabilité par l'application décrite dans le tableau suivant : Notons que les pertes de τ_1 peuvent être modélisées par le paramètre l_i dans la variante BGW1.

Task	T_i	C_i^p	C_i^a	s_i	n_i
τ_1	6	3	1	2	4
τ_2	10	4	1	10	4
τ_3	15	6	2	∞	3

TABLE 3.2 – Exemple illustratif du modèle BGW2.

Par contre, pour la tâche τ_2, nous ne pouvons pas modéliser une seule perte sur 10 instances successives par le paramètre l_i. Nous avons donc besoin du paramètre s_i du modèle BGW2. La tâche τ_3 est de type temps réel dur et n'accepte aucune perte d'instance.
Le facteur d'utilisation est donné par :
Pour τ_1 : $\frac{1}{T_1}(\frac{1}{n_1}C_1^p + (\frac{s_1-1}{s_1} - \frac{1}{n_1} + \frac{1}{PPCM(n_1,s_1)})C_1^a)$, d'où $U_{\tau_1} = 0.2083$.
Pour τ_2 : $\frac{1}{T_2}(\frac{1}{n_2}C_2^p + (\frac{s_2-1}{s_2} - \frac{1}{n_2} + \frac{1}{PPCM(n_2,s_2)})C_2^a)$, d'où $U_{\tau_2} = 0.17$.
Pour τ_3 : $\frac{1}{T_3}(\frac{1}{n_3}C_3^p + (\frac{s_3-1}{s_3} - \frac{1}{n_3} + \frac{1}{PPCM(n_3,s_3)})C_3^a)$, d'où $U_{\tau_3} = 0.2444$.
Le facteur d'utilisation total U_Γ de l'ensemble de tâches Γ est $U_\Gamma = 0.2083+0.17+0.2444$ soit $U_\Gamma = 0.6227$.
Cet ensemble de tâches BGW2 est donc faisable.

3.8 Ordonnancement basique pour BGW2

Nous définissons ici l'algorithme appelé BGa2 pour lequel nous appliquons les règles suivantes :

- On exécute seulement pour chaque tâche, les instances Black, imposées par la contrainte n_i et les versions secondaires des instances Grey, imposées par la contrainte s_i .
- Les instances Black et les versions secondaires des instances grey sont ordonnancées entre elles par l'algorithme à priorité dynamique EDF.
- Toutes les instances White sont systématiquement rejetées.

Condition exacte d'ordonnançabilité de BGa2

Définition : On définit la demande processeur de Γ dans l'intervalle $[0, L]$ notée $DBF_{BGa2}(\Gamma, L)$ par la quantité :

$$DBF_{BGa2}(\Gamma, L) = \sum_{i=1}^{n} D(i, [0, L]) \tag{3.4}$$

avec

$$D(i, [0, L]) = \left(\lceil \frac{L}{n_i.T_i} \rceil \right).C_i^p + \left(\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{s_i.T_i} \rfloor - \lceil \frac{L}{n_i.T_i} \rceil + \lceil \frac{L}{T_i.PPCM(n_i, s_i)} \rceil \right).C_i^a \tag{3.5}$$

Définition : On définit la *charge processeur équivalente* de Γ par la quantité :

$$Load_2^*(\Gamma) = max_{L\in[D_{min},P)} \frac{DBF_{BGa2}(\Gamma, L)}{L} \tag{3.6}$$

$\lceil \frac{L}{n_i.T_i} \rceil$ représente le nombre d'instances Black de τ_i réveillées et ayant leurs échéances dans $[0, L]$. $\left(\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{s_i.T_i} \rfloor - \lceil \frac{L}{n_i.T_i} \rceil + \lceil \frac{L}{T_i.PPCM(n_i,s_i)} \rceil \right)$ représente le nombre d'instances Grey dans $[0, L]$. La somme des deux termes donne le nombre des instances Black et Grey qui doivent s'exécuter impérativement dans $[0, L]$. Ces deux termes multipliés respectivement par C_i^p et C_i^a nous donnent le temps d'exécution nécessaire pour les instances Black (versions primaires) et les versions secondaires des instances Grey dans $[0, L]$. Nous calculons DBF_{BGa2} seulement pour des valeurs de L correspondant aux échéances absolues des instances Black et Grey sur $[0, H(\Gamma)]$.

Nous pouvons énoncer le théorème suivant :

Theorem 9 *Un ensemble de tâches périodiques Γ conformes au modèle BGW2 est ordonnançable par BGa2 si et seulement si sa charge processeur équivalente ne dépasse pas 1 c'est à dire $Load_2^*(\Gamma) \leq 1$.*

Preuve
Si. Par hypothèse (modèle deeply-BGa2), les instances de tâches se réveillent à l'instant 0 et sont Black. Selon l'algorithme BGa2 associé au modèle BGW2, pour chaque tâche τ_i, chaque instance White est systématiquement ignorée.
Toutes les autres instances (instances Black et Grey) sont exécutées selon EDF. $D(i, [t_1, t_2])$ représente la demande de traitement de τ_i dans l'intervalle $[t_1, t_2]$ selon EDF, à savoir le temps total de traitement nécessaire pour l'exécution de toutes les instances Black et Grey de τ_i qui sont réveillées et doivent s'exécuter dans $[t_1, t_2]$.
Supposons que la condition $Load_2^*(\Gamma) \leq 1$ soit vérifiée et qu'à un certain moment t, une instance manque son échéance. Soit $t_a \geq 0$ la dernière fois (avant t) où le processeur a été laissé inactif par l'algorithme d'ordonnancement ($t_a = 0$, s'il n'y en a pas). Soit $t_b \geq 0$ la dernière fois (avant t) où le processeur était occupé à exécuter un instance de tâche possédant une échéance après t ($t_b = 0$ s'il n'y en a pas).
Soit $t' = \max(t_a, t_b)$. Le motif de pertes de l'algorithme assure que $D(i, [t', t]) \leq D(i, [0, t - t'])$. L'instant t' possède la propriété que seules les instances black et grey de tâche qui ont été activées après t' avec une échéance absolue inférieure ou égale à t sont exécutées sur $[t', t]$. Il n'y a aucun moment dans $[t', t]$ où le processeur est inactif. Le fait que l'échéance ait été manquée signifie que la demande de traitement $C[t', t]$ au cours de cet intervalle est supérieure à la longueur de l'intervalle. Autrement dit, $t - t' < C[t', t]$. Ce qui nous conduit à,

$$t - t' < C[t', t] = \sum_{i=1}^{n} D(i, [t', t]) \leq DBF_{BGa2}(\Gamma, [0, t - t']) \leq (t - t').Load_2^*(\Gamma) \tag{3.7}$$

et donc $Load_2^*(\Gamma) > 1$ ce qui nous amène à une contradiction.

Seulement si. Supposons que $Load_2^*(\Gamma) > 1$ sur un intervalle $[0, t]$. la charge de traitement $C[0, t]$ nécessaire pour l'exécution des instances rouges dans le pire cas est $C[0, t] = DBF_{BGa2}(\Gamma, t)$.
Selon la définition de $Load_2^*(\Gamma)$, $\exists t$ tel que $Load_2^*(\Gamma).t = DBF_{BGa2}(\Gamma, t)$.
Par conséquent, $C[0, t] = DBF_{BGa2}(\Gamma, t) = DBF_{BGa2}(\Gamma, t) > t$. Comme le temps de traitement sur $[0, t]$ dépasse la longueur de ce même intervalle alors Γ n'est pas faisable.

Il est suffisant de vérifier la condition du Théorème 9 pour les instants t correspondant aux échéances absolues des instances. De plus, il est inutile de tester cette condition pour des instants supérieurs à $H(\Gamma)$.

3.9 Variante BGW3

3.9.1 Présentation

Le modèle BGW3 caractérise chaque tâche τ_i par les paramètres de QdS à valeurs entières, n_i, m_i et k_i où l'on représente par :

- n_i : la distance maximale autorisée séparant l'exécution réussie de deux versions primaires,
- m_i, k_i l'exigence qu'au moins m_i instances doivent s'exécuter dans chaque intervalle de longueur égale à k_i périodes successives.

Nous avons donc un ensemble de tâches τ_i modélisées par
$\Gamma = \{r_i, C_i^a, C_i^p, D_i, T_i, n_i, m_i, k_i 1 \leq i \leq n\}$

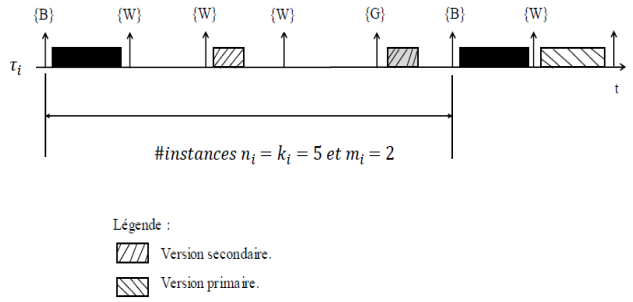


FIGURE 3.5 – Illustration des paramètres du modèle BGW3.

3.9.2 Test de faisabilité

Le test de faisabilité du modèle BGW3 est formulé dans le théorème suivant. La situation pire cas correspond à l'exécution régulière des primaires avec la période constante égale à $n_i.T_i$ et à l'exécution de m_i instances dans chaque intervalle de k_i instances successives.
Pour la détermination de la condition nécessaire de faisabilité du modèle BGW3, nous devons considérer deux cas. Le premier cas correspond à $\frac{1}{n_i} \leq \frac{m_i}{k_i}$. Le deuxième cas correspond donc à $\frac{1}{n_i} > \frac{m_i}{k_i}$.

Theorem 10 *Un ensemble de tâches Γ conforme au modèle BGW3 avec $\frac{1}{n_i} \leq \frac{m_i}{k_i}$ est faisable sur un système monoprocesseur seulement si $\sum_{i=1}^{n} U_{\tau_i} \leq 1$ avec $U_{\tau_i} = \frac{1}{n_i.T_i}.C_i^p + (\frac{m_i}{k_i.T_i} - \frac{1}{n_i.T_i}).C_i^a$.*

Preuve.
Par la définition du modèle BGW3, la situation pire cas apparaît lorsque le système exécute pour chaque tâche τ_i, dans une période de n_i instances successives, une version primaire et $(\frac{m_i}{k_i.T_i} - \frac{1}{n_i.T_i}).C_i^a$ versions secondaires.

Autrement dit, la partie $\frac{1}{n_i.T_i}.C_i^p$ correspond au facteur d'utilisation du nombre d'instances Black à exécuter pour τ_i, selon la contrainte n_i du modèle BGW3 sur les versions primaires.
Le nombre $\frac{m_i}{k_i.T_i} + \frac{1}{n_i.T_i}$ correspond au nombre d'instances à exécuter selon la contrainte d'exécutions du modèle BGW3. D'où, la grandeur $(\frac{m_i}{k_i.T_i} + \frac{1}{n_i.T_i}).C_i^a$ qui correspond au facteur d'utilisation pire cas de ce nombre d'instances à exécuter selon la contrainte de l_i de BGW3. Nous commençons l'exécution par une version primaire pour chaque tâche. Ensuite, nous devons exécuter un nombre m_i d'instances dans chaque intervalle de k_i instances successives et ceci, jusqu'à la fin d'exécution. La contrainte de m_i impose des exécutions et pas un type précis de versions. Par contre, la contrainte de n_i impose des versions primaires. Donc, si nous devons respecter cette dernière contrainte, en même temps, nous prenons cette exécution en considération pour la contrainte de m_i. Finalement, vu que le facteur d'utilisation total d'un système monoprocesseur ne doit pas dépasser 100%, le théorème est prouvé.

Theorem 11 *Un ensemble de tâches τ conformes au modèle BGW3 avec $\frac{1}{n_i} > \frac{m_i}{k_i}$ est faisable sur un système monoprocesseur seulement si $\sum_{i=1}^{n} U_{\tau_i} \leq 1$ avec $U_{\tau_i} = \frac{1}{n_i.T_i}.C_i^p$.*

Preuve.
Par la définition du modèle BGW3, la situation pire cas se produit lorsque le système exécute pour chaque tâche τ_i, dans une période de n_i instances successives, une version primaire.
Autrement dit, la partie $\frac{1}{n_i.T_i}.C_i^p$, correspond au facteur d'utilisation du nombre d'instances Black à exécuter pour τ_i, selon la contrainte n_i du modèle BGW3 sur les versions primaires et elle correspond en même temps au facteur d'utilisation du nombre d'instances à exécuter selon la contrainte de m_i du modèle BGW3. Nous commençons l'exécution par une version primaire pour chaque tâche. Finalement, vu que le facteur d'utilisation total d'un système monoprocesseur ne doit pas dépasser 100%, le théorème est prouvé.

3.9.3 Cas particuliers

En fonction des valeurs de n_i, m_i et k_i, BGW3 nous permet de modéliser une tâche de type temps réel dur ou temps réel ferme.

- Si $n_i = 1$ et $m_i = k_i$, τ_i est une tâche de type temps réel dur qui n'accepte aucune dégradation, ni qualitative, ni quantitative. En d'autres termes, aucune perte n'est acceptée et toutes instances exécutent les versions primaires.
- Si $m_i \leq k_i$ et $n_i = \infty$, τ_i est une tâche de type temps réel dur qui accepte une dégradation qualitative, mais pas de dégradation quantitative (toutes les instances sont Grey, aucune perte n'est autorisée).

3.9.4 Exemple illustratif

Nous illustrons la modélisation BGW3 et sa condition de faisabilité par l'exemple donné dans le tableau suivant : Notons que les pertes de la tâche τ_1 peuvent être modélisées par le paramètre l_i dans la variante

Task	T_i	C_i^p	C_i^a	m_i	k_i	n_i
τ_1	6	3	2	1	2	4
τ_2	10	4	1	4	7	4
τ_3	15	6	2	6	10	3

TABLE 3.3 – Exemple illustratif du modèle BGW3.

BGW1. La tâche τ_3 est temps réel dur et n'accepte aucune perte d'instances.
Le facteur d'utilisation est donné par :

Pour la tâche τ_1 : $\frac{1}{n_1.T_1}.C_1^p + (\frac{m_1}{k_1.T_1} - \frac{1}{n_1.T_1}).C_1^a$, d'où $U_{\tau_1} = 0.2083$.
Pour la tâche τ_2 : $\frac{1}{n_2.T_2}.C_2^p + (\frac{m_2}{k_2.T_i} - \frac{1}{n_2.T_2}).C_2^a$, d'où, $U_{\tau_2} = 0.1321$.
Pour la tâche τ_3 : $\frac{1}{n_3.T_3}.C_3^p + (\frac{m_3}{k_3.T_3} - \frac{1}{n_3.T_3}).C_3^a$, d'où, $U_{\tau_3} = 0.1688$.
Le facteur d'utilisation total U_Γ de l'ensemble de tâches Γ est $U_\Gamma = 0.2083 + 0.1321 + 0.1688$ soit $U_\Gamma = 0.5092$. Alors, cet ensemble de tâches BGW3 est faisable.

3.9.5 Ordonnancement basique pour BGW3

Nous définissons l'algorithme BGa3 par la stratégie d'ordonnancement dans laquelle, nous appliquons ces règles :

- On exécute seulement pour chaque tâche, les instances Black et les versions secondaires des instances Grey, imposées par les contraintes n_i, m_i et k_i.
- Les instances Black et versions secondaires des instances Grey pour l'ensemble de tâches, sont ordonnancées entre elles par EDF.
- Toutes les instances White, sont systématiquement rejetées.

Condition exacte d'ordonnançabilité de BGa3
Définition : Soit un ensemble $\Gamma = \{\tau_i(C_i, T_i, D_i, n_i, m_i, k_i)\}$ de n tâches périodiques conformes au modèle BGW3. La **demande processeur** de Γ, notée $DBF_{BGa3}(\Gamma, L)$, liée à l'exécution des instances Black et Grey dans $[0, L]$ est définie comme suit :

$$DBF_{BGa3}(\Gamma, L) = \sum_{i=1}^{n} D(i, [0, L]) \tag{3.8}$$

Avec

$$D(i, [0, L]) = \left(\lceil \frac{L}{n_i.T_i} \rceil \right).C_i^p + \left(\lfloor \frac{L.m_i}{k_i.T_i} \rfloor - \lfloor \frac{L}{n_i.T_i} \rfloor \right).C_i^a \tag{3.9}$$

$\lceil \frac{L}{n_i.T_i} \rceil$ représente le nombre d'instances Black de τ_i actives et ayant leurs échéances dans $[0, L]$.
$\lfloor \frac{L.m_i}{k_i.T_i} \rfloor - \lfloor \frac{L}{n_i.T_i} \rfloor$ représente le nombre d'instances Grey dans $[0, L]$. La somme des deux termes donne le nombre des instances Black et Grey qui doivent s'exécuter impérativement dans $[0, L]$.
Ces deux termes multipliés respectivement par C_i^p et C_i^a, nous donnent le temps d'exécution nécessaire pour l'exécution des instances Black et Grey secondaires dans $[0, L]$.

Définition : Soit un ensemble $\Gamma = \{\tau_i(C_i, T_i, D_i, n_i, m_i, k_i)\}$ de n tâches périodiques conformes au modèle BGW3, la **charge équivalente** de Γ est définie par :

$$Load_1^*(\Gamma) = max_{L \in [D_{min}, P)} \frac{DBF_{BGa3}(\Gamma, L)}{L} \tag{3.10}$$

Nous calculons DBF_{BGa3} seulement pour des valeurs de L correspondant aux échéances absolues des instances Black et Grey sur $[0, H(\Gamma)]$.

Nous énonçons alors le théorème suivant :

Theorem 12 *Un ensemble de tâches périodiques Γ conformes au modèle BGW3 est ordonnançable par BGa3 si et seulement si sa charge processeur équivalente ne dépasse pas 1 c'est à dire $Load_3^*(\Gamma) \leq 1$.*

Preuve
Si. Par hypothèse (modèle deeply-BGa3), les instances de tâches se réveillent à l'instant 0 et elles sont Black.
Selon l'algorithme BGa3, pour chaque tâche τ_i, chaque instance White est systématiquement ignorée. Toutes les autres instances (instances Black et Grey) sont exécutées selon EDF.
$D(i,[t_1,t_2])$ représente la demande de traitement de τ_i dans l'intervalle $[t_1,t_2]$ selon EDF, à savoir le temps total de traitement nécessaire pour l'exécution de toutes les instances Black et Grey de τ_i qui sont réveillées et doivent s'exécuter dans l'intervalle $[t_1,t_2]$. Supposons que la condition $Load_3^*(\Gamma) \leq 1$ soit vérifiée et qu'à un certain instant t, une instance manque son échéance. Soit $t_a \geq 0$ la dernière fois (avant t) où le processeur a été laissé inactif par l'algorithme d'ordonnancement ($t_a = 0$, s'il n'y en a pas). Soit $t_b \geq 0$ la dernière fois (avant t) où le processeur était occupé à exécuter un instance de tâche possédant une échéance après t ($t_b = 0$ s'il n'y en a pas).
Soit $t' = \max(t_a, t_b)$. Le motif de pertes de l'algorithme assure que $D(i,[t',t]) \leq D(i,[0,t-t'])$. L'instant t' possède la propriété que seules les instances black et grey de tâche qui ont été activées après t' avec une échéance absolue inférieure ou égale à t sont exécutées sur $[t',t]$. Il n'y a aucun moment dans $[t',t]$ où le processeur est inactif. Le fait que l'échéance ait été manquée signifie que la demande de traitement $C[t',t]$ au cours de cet intervalle est supérieure à la longueur de l'intervalle . Autrement dit, $t - t' < C[t',t]$. Ce qui nous conduit à,

$$t - t' < C[t',t] = \sum_{i=1}^{n} D(i,[t',t]) \leq DBF_{BGa3}(\Gamma,[0,t-t']) \leq (t-t').Load_3^*(\Gamma) \tag{3.11}$$

et donc $Load3^*(\Gamma) > 1$, ce qui nous amène à une contradiction.

Seulement si. Supposons que $Load_3^*(\Gamma) > 1$ sur un intervalle $[0,t]$, la charge de traitement $C[0,t]$ nécessaire pour l'exécution des instances black et grey dans le pire cas est $C[0,t] = DBF_{BGa3}(\Gamma,t)$.
Selon la définition de $Load_3^*(\Gamma), \exists t$ tel que $Load_3^*(\Gamma).t = DBF_{BGa3}(\Gamma,t)$. Par conséquent, $C[0,t] = DBF_{BGa1}(\Gamma,t) = DBF_{BGa3}(\Gamma,t) > t$. Comme le temps de traitement sur $[0,t]$ dépasse la longueur de ce même intervalle alors Γ n'est pas faisable.

Il est suffisant de vérifier la condition de ce théorème pour les instants t correspondant aux échéances absolues des instances. De plus, il est inutile de tester cette condition pour des instants supérieurs à $H(\Gamma)$.

3.10 Commentaires

3.10.1 Apports de BGW

Contrairement au Mécanisme à Echéance, le modèle BGW permet de spécifier :

- un taux minimum de versions primaires exécutées (valeur exacte d'une commande, par exemple) (par spécification du paramètre n_i.
- un taux minimum de versions secondaires exécutées (par spécification du paramètre l_i, s_i ou m_i et k_i).
- et un taux maximum de pertes autorisées.

Le modèle BGW présente aussi la particularité de :

- pouvoir intégrer des contraintes de tolérance aux fautes outre celles liées à la gestion de surcharge.
- pouvoir modéliser des applications temps réel dures ne tolérant aucune perte (pas d'instances White) et aucune dégradation sur la qualité des résultats de calcul (pas d'instances Grey).

3.10.2 Quelle variante choisir ?

Le modèle BGW1 permet la modélisation exacte et facile des contraintes de QdS quantitative et qualitative. Il a comme grand avantage, la simplicité de conception, d'analyse de faisabilité et aussi d'implémentation. Cependant, ce modèle ne permet pas la modélisation de niveaux de pertes trés petits.
Le plus petit taux de perte modélisable par BGW1 est celui dans lequel nous avons une perte d'exécution autorisée une instance sur deux, à savoir le cas où $l_i = 2$.

Le modèle BGW2 a été introduit pour surmonter ce problème. En effet, le paramètre s_i permet la modélisation de tous les taux de pertes plus petits. Par contre, BGW2 ne permet pas la modélisation des niveaux de pertes plus élevés. L'inconvénient de BGW2 est qu'il ne permet pas la modélisation de systèmes avec des taux de perte élevés.
Un autre avantage du modèle BGW2 est qu'il permet de modéliser l'exigence d'exécuter une série d'instances consécutives. Cette exigence se retrouve en particulier dans les applications multimédia.

La variante BGW3 est un modèle plus général que les précédents plus apte à modéliser les contraintes sur les pertes. Toutefois, cet avantage engendre un inconvénient important, celui d'une plus grande complexité de l'analyse de faisabilité.

Pour des raisons de temps, nous avons dû restreindre le champ de notre étude à une seule des variantes. Nous avons opté pour le modèle BGW1.

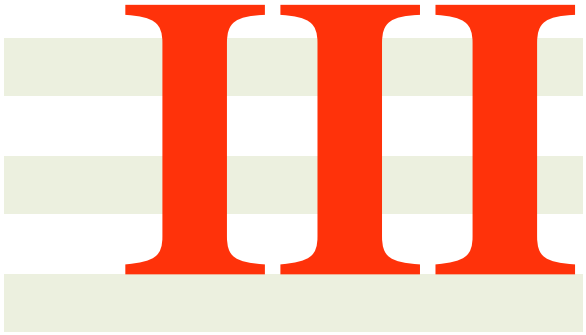
3.11 Conclusion

Ce chapitre a introduit un nouveau modèle de tâche temps réel qui s'inspire des modèles du mécanisme à échéance et skip-over. L'idée est de pouvoir spécifier, pour chaque tâche, la façon dont on souhaite assurer la dégradation de la qualité de service rendu lorsque le système passe en état de surcharge. Ce travail a fait l'objet de la communication suivante :
The BGW model for QdS aware scheduling of real-time embedded systems, Mohamed Ould Sass, Maryline Chetto, Audrey Queudet, Proceedings of the 11th ACM international symposium on Mobility management and wireless access, Pages 93-100, 2013.

Dans le modèle que nous proposons, nous supposons que toute instance de tâche est caractérisée par une couleur parmi trois possibles. Cette couleur est alors utilisée par le système d'exploitation et en particulier l'ordonnanceur pour fournir une QdS sans dégradation ou une QdS avec dégradation.

Pour ce faire, le modèle BGW intègre, outre les paramètres temporels de la tâche, des paramètres dits de Qualité de Service pour spécifier la situation pire cas en matière de dégradation de service autorisée. Nous avons donc proposé trois variantes de la situation pire cas au travers des modèles BGW1, BGW2 et BGW3. Nous avons donné la condition nécessaire de faisabilité associée à chacun d'eux. Puis, se basant sur l'ordonnanceur EDF, nous avons décrit pour chaque variante, une condition exacte d'ordonnançabilité de l'ordonnanceur appelé BGa qui consiste à fournir la QdS minimale exigée.

Dans la partie suivante, nous proposerons des ordonnanceurs spécifiquement dédiés au modèle BGW dont la finalité est d'optimiser la QdS de sortie tout en respectant les contraintes du modèle.



Contribution à l'ordonnancement de tâches BGW

4

Ordonnancements élémentaires pour BGW

4.1 Présentation du chapitre

Ce chapitre porte sur l'analyse du comportement de plusieurs ordonnanceurs d'implémentation simple appliqués à des tâches conformes au modèle BGW, ici dans sa variante BGW1. Nous donnerons d'abord les principaux critères qui nous servirons à effectuer une analyse comparative de ces ordonnanceurs. Nous montrerons les différentes combinaisons possibles pour ordonner les listes d'instances de tâches prêtes sachant que celles ci peuvent avoir trois couleurs et être exécutées par deux versions de code programme. Ceci nous conduira à étudier quatre ordonnanceurs, tous basés sur la technique de première chance et la règle d'ordonnancement EDF.

4.2 Critères de performances adoptés

Dans l'objectif de mesurer la performance effective obtenue par différents ordonnanceurs, nous devons lister les principales mesures à effectuer et ce, sur un échantillon représentatif d'ensembles de tâches, pour divers facteurs d'utilisation et diverses exigences en termes de qualité de service.
La sélection de ces mesures dépend du point de vue adopté. Pour un utilisateur final, la principale mesure de performance concerne la qualité de service exprimée en termes d'échéances satisfaites d'une part et de précision de résultat d'autre part. Alors que du point de vue de l'éditeur de logiciel, celui ci veillera en plus à concevoir un ordonnanceur avec le moins d'overheads possible liés à sa mise en oeuvre (temps d'exécution, nombre de préemptions).

Qualité de Service numéro 1 (QdS1)
Ce paramètre représente la Qualité de Service du système appelée QdS1. On mesure ici le pourcentage d'échéances respectées par les instances, soit par leur version primaire ou par leur version secondaire.

$$NSJ = \frac{\textit{Nombre d'échéances respectées}}{\textit{Nombre total de requêtes}} \tag{4.1}$$

Qualité de Service numéro 2 (QdS2)
Ce paramètre représente la Qualité de Service du système appelée QdS2. On mesure ici le pourcentage d'échéances respectées par les versions primaires uniquement.

$$NPJ = \frac{\textit{Nombre d'échéances respectées par primaires}}{\textit{Nombre total de requêtes}} \tag{4.2}$$

Taux de préemption
C'est la moyenne du rapport entre le nombre total de préemptions et le nombre de demandes d'exécution traitées sur un intervalle de simulation donné.

$$PR = \frac{\textit{Nombre de préemptions}}{\textit{Nombre total de requêtes}} \tag{4.3}$$

Gaspillage du temps processeur
Le critère de gaspillage permet de mesurer le temps processeur passé à exécuter des instances non terminées avant échéance ou dont le résultat n'a été d'aucune utilité.

$$WTR = \frac{\textit{Temps alloué à des instances abandonnées}}{\textit{Temps total de la simulation}} \tag{4.4}$$

Comme nous l'avons mentionné dans le chapitre précédent, le modèle BGW est flexible : il peut être utilisé pour la modélisation des tâches de type temps réel dur et aussi pour des tâches classiques mono-version. Les remarques suivantes expliquent ces paramétrisations particulières des tâches.

Remarque 1 :
Pour la simulation d'un système utilisant le Modèle du Mécanisme à Échéance, il suffit de choisir les paramètres ($n_i = \infty = 10000$ par exemple et $l_i = 0$), c.-à-d. que toutes les instances sont Grey : On exécute soit la version primaire, soit la version secondaire.

Remarque 2 :
Pour la simulation d'un système classique (Un ensemble de tâches avec une seule version qui n'acceptent pas de pertes) il suffit de choisir les paramètres ($n_i = 0$ et $l_i = \infty = 10000$ par exemple), c.-à-d. que toutes les instances sont Black.

4.3 Description schématique d'un ordonnanceur

Les ordonnanceurs présentés dans ce chapitre se basent sur la technique dite de la première chance, plus simple à mettre en oeuvre que celle de la dernière chance.
Dans cette technique, on exécute d'abord, pour toute tâche, sa version secondaire qui élabore un résultat juste acceptable, afin de garantir la production d'au moins un résultat. Ensuite, nous tentons d'exécuter la version primaire, la plus longue, qui élabore un résultat précis et de bonne qualité mais susceptible de ne pas terminer en présence de surcharge.

Pour toute instance Grey ou White, nous exécutons ainsi d'abord la version secondaire sachant que pour chaque instance black, c'est nécessairement la version primaire qui doit s'exécuter.
Dans ce qui suit, on note les listes d'instances comme suit : Liste d'instances Black (B), liste d'instances Grey primaires (Gp), liste d'instances Grey secondaires (Ga), liste d'instances White primaires (Wp) et liste d'instances White secondaires (Wa). Chaque liste est gérée par la règle EDF (Earliest Deadline First).

Nous avons trois possibilités pour appliquer la technique de la première chance sous le modèle BGW :

- B, Ga, Wa, Gp et Wp (Ordonnanceur AbP)
- B, Ga, Gp, Wa et Wp (Ordonnanceur $GbWa$)
- B, Ga, Gp et Wp (Ordonnanceur $GbWp$)

Cela signifie par exemple, que nous n'exécutons une instance (Ga) que, lorsqu'il n'y aucune instance Black prête et que que nous n'exécutons une instance (Wa) que lorsqu'il n'y aucune instance (Ga) prête. Une fois qu'une exécution de version primaire est complètement effectuée, nous n'avons pas besoin d'exécuter la version secondaire correspondante de la même instance.

4.4 Ordonnanceur AbP

Dans le but de maximiser le taux de respect d'échéances global (QdS1), avec n'importe quelle version (primaire ou secondaire), on donne une priorité plus élevée aux instances Wa par rapport aux instances Gp et Wp.
D'où, Priorité de Liste $B >$ Priorité de Liste $Ga >$ Priorité de Liste $Wa >$ Priorité de Liste $Gp >$ Priorité de Liste Wp où $>$ est un opérateur spécifiant un ordre de priorité supérieur.

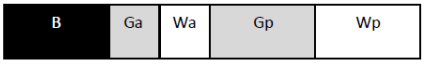


FIGURE 4.1 – Ordre d'exécution entre listes pour AbP.

La figure 4.2 donne le pseudo code de l'ordonnanceur AbP.

4.5 Ordonnanceur $GbWa$

Dans le but d'avoir un équilibrage entre le taux de respect d'échéances global des tâches (QdS1) et le taux de respect d'échéances par des versions primaires (QdS2), on donne une priorité plus élevée aux instances Gp par rapport aux instances Wa et Wp.
D'où, les priorités entres les différentes listes d'instances sont affectées de la manière suivante : *Priorité de Liste* $B >$ *Priorité de Liste* $Ga >$ *Priorité de Liste* $Gp >$ *Priorité de Liste* $Wa >$ *Priorité de Liste* Wp

Autrement dit, dans la stratégie $GbWa$ nous exécutons d'abord les instances B, puis les Ga, puis les Gp, puis les Wa et finalement les Wp.

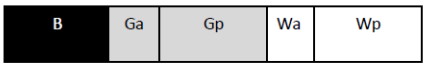


FIGURE 4.3 – Ordre d'exécution entre listes pour $GbWa$.

La figure 4.4 donne le pseudo code de l'ordonnanceur $GbWa$.

4.6 Ordonnanceur $GbWp$

Dans le but d'optimiser d'abord le taux d'exécution des versions primaires des tâches (QdS2), on donne une priorité plus élevée aux instances Gp par rapport aux instances Wp. En remarquant que lorsque nous exécutons des Wp, nous n'avons plus besoin d'exécuter les Wa correspondantes.

Algorithm 1 Algorithme d'ordonnancement AbP

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$

BpList, Liste des instances B prêtes ordonnancée entre elles selon EDF ;
GpList, Liste des instances Gp prêtes ordonnancée entre elles selon EDF ;
GaList, Liste des instances Ga prêtes ordonnancée entre elles selon EDF ;
WpList, Liste des instances Wp prêtes ordonnancée entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancée entre elles selon EDF ;

```
begin
if BList.Head != NULL then
    /* Il y a des instances B prêtes pour s'exécuter */
    exécuter BList.Head et supprimer le, après exécution ;
else
    /* Pas d'instances B prêtes pour s'exécuter */
    if GaList.Head != NULL then
        /* Présence des instances Ga prêtes pour s'exécuter */
        exécuter GaList.Head et supprimer le, après exécution ;
    else
        /* Pas d'instances Ga prêtes pour s'exécuter */
        if WaList.Head != NULL then
            /* Présence des instances Wa prêtes pour s'exécuter */
            exécuter WaList.Head et supprimer le, après exécution ;
        else
            /* Pas d'instances Wa prêtes pour s'exécuter */
            if  GpList.Head != NULL then
                /* Présence des instances Gp prêtes pour s'exécuter */
                exécuter GpList.Head et supprimer le, après exécution ;
            else
                /* Pas d'instances Gp prêtes pour s'exécuter */
                if WpList.Head != NULL then
                    /* Présence des instances Wp prêtes pour s'exécuter */
                    exécuter WpList.Head et supprimer le après exécution ;
                end if
            end if
        end if
    end if
end if
end
```

FIGURE 4.2 – Pseudo-code de AbP

Algorithm 2 Algorithme d'ordonnancement $GbWa$

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$

BpList, Liste des instances B prêtes ordonnancée entre elles selon EDF ;
GpList, Liste des instances Gp prêtes ordonnancée entre elles selon EDF ;
GaList, Liste des instances Ga prêtes ordonnancée entre elles selon EDF ;
WpList, Liste des instances Wp prêtes ordonnancée entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancée entre elles selon EDF ;

```
begin
if BList.Head != NULL then
    /* Il y a des instances B prêtes pour s'exécuter */
    exécuter BList.Head et supprimer le, après exécution ;
else
    /* Pas d'instances B prêtes pour s'exécuter */
    if GaList.Head != NULL then
        /* Présence des instances Ga prêtes pour s'exécuter */
        exécuter GaList.Head et supprimer le, après exécution ;
    else
        /* Pas d'instances Ga prêtes pour s'exécuter */
        if GpList.Head != NULL then
            /* Présence des instances Gp prêtes pour s'exécuter */
            execute GpList.Head and remove it after completion ;
        else
            /* Pas d'instances Gp prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                exécuter WaList.Head et supprimer le, après exécution ;
            else
                /* Pas d'instances Wa prêtes pour s'exécuter */
                if WpList.Head != NULL then
                    /* Présence des instances Wp prêtes pour s'exécuter */
                    exécuter WpList.Head et supprimer le, après exécution ;
                end if
            end if
        end if
    end if
end if
end
```

FIGURE 4.4 – Pseudo-code de $GbWa$

D'où les priorités entres les différentes listes d'instances classées comme suit : *Priorité de Liste* $B >$ *Priorité de Liste* $Ga >$ *Priorité de Liste* $Gp >$ *Priorité de Liste* Wp

FIGURE 4.5 – Ordre d'exécution entre listes pour $GbWp$.

La figure 4.6 donne le pseudo code de l'algorithme d'ordonnancement $GbWp$.

Algorithm 3 Algorithme d'ordonnancement $GbWp$

```
Entrées : Γ = {τ_1, ..., τ_n}
    BpList, Liste des instances B prêtes ordonnancée entre elles selon EDF ;
    GpList, Liste des instances Gp prêtes ordonnancée entre elles selon EDF ;
    GaList, Liste des instances Ga prêtes ordonnancée entre elles selon EDF ;
    WpList, Liste des instances Wp prêtes ordonnancée entre elles selon EDF ;
    WaList, Liste des instances Wa prêtes ordonnancée entre elles selon EDF ;

    begin
    if BList.Head != NULL then
        /* Il y a des instances B prêtes pour s'exécuter */
        exécuter BList.Head et supprimer le, après exécution ;
    else
        /* Pas d'instances B prêtes pour s'exécuter */
        if GaList.Head != NULL then
            /* Présence des instances Ga prêtes pour s'exécuter */
            exécuter GaList.Head et supprimer le, après exécution ;
        else
            /* Pas d'instances Ga prêtes pour s'exécuter */
            if GpList.Head != NULL then
                /* Présence des instances Gp prêtes pour s'exécuter */
                exécuter GpList.Head et supprimer le, après exécution ;
            else
                /* Pas d'instances Gp prêtes pour s'exécuter */
                if WpList.Head != NULL then
                    /* Présence des instances Wp prêtes pour s'exécuter */
                    exécuter WpList.Head et supprimer le, après exécution ;
                end if
            end if
        end if
    end if
    end
```

FIGURE 4.6 – Pseudo-code de $GbWp$

4.7 Environnement de simulation

Nous présentons ici l'outil de simulation que nous avons développé. Le logiciel de simulation a été réalisé en langage C. Celui-ci se compose essentiellement des éléments suivants :

- Un bloc *génération de tâches*.
- Un bloc *initialisation et gestion de listes d'instances*.
- Un bloc *ordonnancement*.
- Un bloc*dispatching*.
- Un bloc *calcul et enregistrement de résultats statistiques*.

4.7.1 Génération des tâches périodiques

Le bloc de génération de tâches a pour rôle de délivrer un ensemble de tâches périodiques avec des contraintes BGW. Ce bloc construit donc un ensemble de tâches périodiques conforme au modèle BGW. Les paramètres de sortie de ce générateur sont des paramètres de caractérisation des tâches (c'est à dire la

période, l'échéance relative, le pire temps d'exécution de la version secondaire et le pire temps d'exécution de la version primaire).
Ce bloc a aussi pour rôle de vérifier que la condition de faisabilité est satisfaite. Le générateur de tâches est utilisé pour produire un ensemble de tâches à partir des paramètres d'entrée suivants :

- le nombre de tâches (n) et les paramètres n_i, l_i des tâches.
- le plus petit commun multiple des périodes $(PPCM)$.
- la charge pire cas des versions primaires (U_p).
- la charge pire cas des versions secondaires (U_a), variable en fonction de la charge (U_p).

Dans toutes les simulations présentées ici, les paramètres n et $PPCM$ prennent des valeurs constantes respectivement de 22 et 3360. Chaque point sur les graphes des données représentera un résultat moyen de 50 expériences.

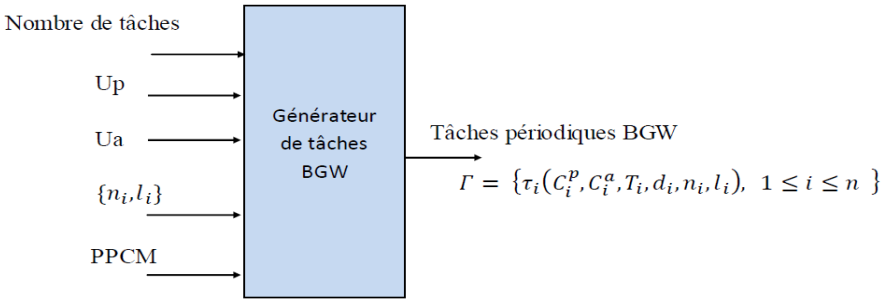


FIGURE 4.7 – Génération de tâches BGW.

4.7.2 Création et gestion des listes d'instances

Ce bloc sert à initialiser et à réserver la mémoire pour chaque liste d'instances. Il dispose de toutes les fonctions nécessaires à la manipulation de ces listes tel que suppression d'une tâche d'une liste ou ajout d'une tâche en fonction d'un ordre donné.

4.7.3 Ordonnancement

Ce bloc essentiel de notre système de simulation sert à classer et à préciser la couleur de chaque nouvelle instance. Les différentes instances des tâches sont identifiées Black, Grey ou White et classées en fonction des instants d'arrivée, des échéances, et des paramètres du modèle BGW utilisé et en fonction des exécutions des instances précédentes de chaque tâche.
Les règles sur lesquelles l'algorithme d'identification et d'ordonnancement des instances s'appuie, peuvent être résumées comme suit :

- La première instance de chaque tâche est considérée Black (elle doit exécuter sa version primaire).
- Si la version primaire de la première instance est bien exécutée, la prochaine instance est considérée comme étant Grey.
- Toutes les instances qui suivent sont considérées Grey.

- Si on arrive à exécuter successivement les $(ni-1)$ versions secondaires des instances Grey, l'instance qui les suit doit être considérée comme Black (elle doit exécuter sa version primaire).
- Le nombre de pertes successives ne doit pas dépasser $l_i - 1$ dans le cas de la variante BGW1, $1/s_i$ dans le cas de BGW2 et $(m_i - 1)/k_i$ dans le cas de BGW3.
- Les instances Black sont prioritaires par rapport aux instances Grey, qui sont à leur tour prioritaires par rapport aux instances White.
- Si le processeur a suffisamment de temps et qu'il n'y a pas d'instances Black ou Grey prêtes, il peut exécuter une version primaire ou une version secondaire d'une instance White.

4.7.4 Dispatcher

Le rôle du dispatcher est de choisir l'instance (et la version) à exécuter en fonction de l'ordonnanceur choisi et en fonction de la technique de la première ou de la seconde chance utilisée. Rappelons qu'avec la technique de la première chance, la version secondaire d'une instance Grey est exécutée avant sa version primaire. Cette dernière sera exécutée après si du temps processeur est disponible.

4.7.5 Affichage et enregistrement des résultats

Dans le bloc de calcul et d'enregistrement des résultats statistiques, nous effectuons les opérations nécessaires pour la mesure des performances.

Le schéma suivant résume l'environnement de simulation adopté dans ce travail. Les parties essentielles de cet environnement et leurs interactions y sont représentées.

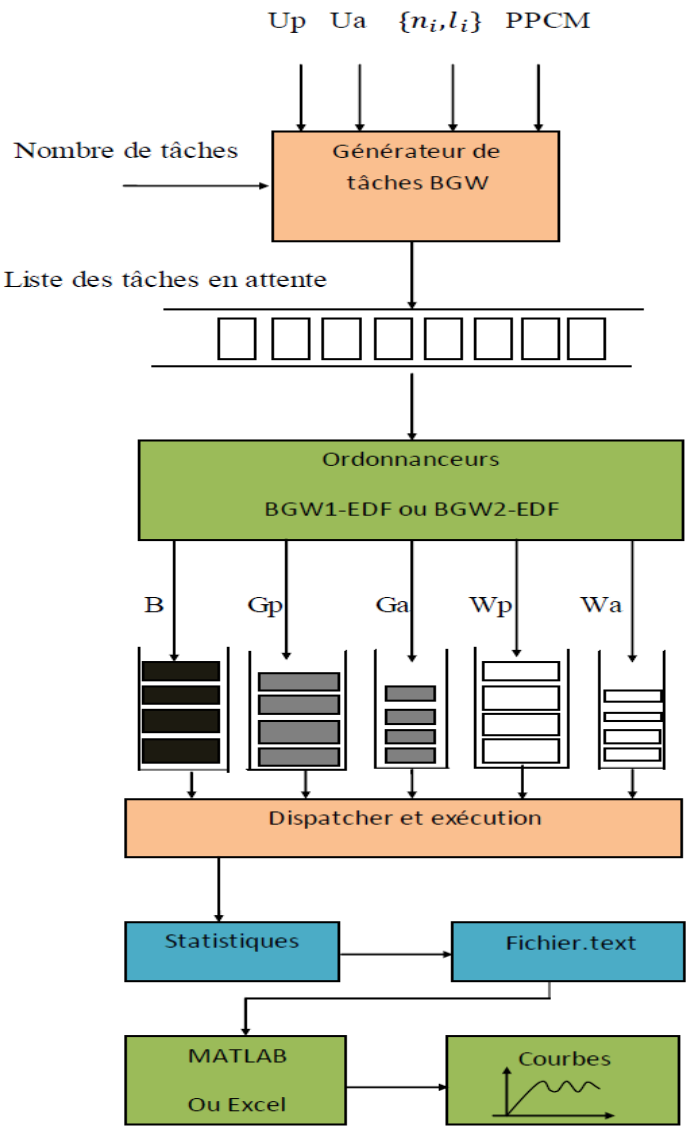


FIGURE 4.8 – Environnement de simulation.

4.8 Analyse de performances par variation de charge

Dans ce paragraphe, nous comparons les performances des ordonnanceurs précédents avec la stratégie classique EDF. Selon cette dernière, toutes les instances prêtes ont une seule version primaire et sont toutes ordonnées selon EDF.

4.8.1 Configuration de l'expérience

Nous avons examiné 14 valeurs de U_p qui varient uniformément de $0,8$ à $2,2$. Pour chacune, deux fonctions de variation de U_a ont été considérées : $U_a = 0.2 * U_p$ et $U_a = 0.6 * U_p$. Plus précisément, C_a^i et C_p^i sont choisies pour être proportionnelle à T_i avec valeur minimale égale à 1. Les métriques effectuées sont celles décrites précédemment.

4.8.2 Analyse du taux de succès global NSJ

Nous analysons dans ce paragraphe le NSJ, qui représente une des deux mesures de QdS.
Les figures 4.9 à 4.14, montrent les variations de NSJ en fonction de la variation de la charge totale des primaires U_p (et par conséquent de la charge totale des secondaires U_a) pour différents types d'applications temps réel.
Nous remarquons sur les quatre courbes, que pour des niveaux de charge élevés, les stratégies AbP et $GbWa$ offrent des performances meilleures que $GbWp$ et EDF.
La stratégie AbP offre un NSJ supérieur à celui fourni par la stratégie $GbWa$, qui est, à son tour, meilleur que celui fourni par $GbWp$.
En effet, sous des charges secondaires (U_a) petites par rapport aux charges primaires (U_p), il y a des chances supplémentaires pour l'exécution des versions secondaires d'instances Grey et White lorsqu'elles ont des priorités plus grandes, dans l'ordre d'exécution des listes.
L'ordonnanceur classique EDF provoque davantage de violations d'échéances que les trois stratégies d'ordonnancement spécifiquement adaptées au modèle BGW. Ces observations confirment l'utilité de notre nouveau modèle de tâches pour la gestion de la surcharge. Pour toutes les stratégies BGW, le nombre de pertes d'échéances augmente lorsque U_p augmente.
Évidemment, pour des grandes valeurs de U_a, davantage de temps processeur se trouve consommé par des secondaires Grey, ce qui conduit ainsi à une augmentation de pertes d'échéances par les primaires Grey.
Nous remarquons clairement, l'impact important du placement de versions secondaires Ga ou Wa dans l'ordre d'exécution des listes, sur les performances des approches BGW, en termes de succès global. D'où l'utilité des deux stratégies AbP et $GbWa$.

Á partir de $U_p = 100\%$ (système surchargé), le NSJ commence à diminuer fortement pour atteindre un taux de 50% pour $U_p = 200\%$ sous l'algorithme EDF, et 40% sous la Stratégie $GbWp$.
Par contre, pour les stratégies AbP et $GbWa$, la diminution de NSJ est négligeable et indépendante de la charge des premières, jusqu'à ce que $U_p = 210\%$ où il commence à diminuer de façon nette.
Les comportements de AbP et de $GbWa$, du point de vue du NSJ, restent presque similaires jusqu'à ce que $U_p = 170\%$, où une petite différence entre les deux stratégies commence à apparaître.

Pour des niveaux de charge élevés, le taux de pertes d'échéances pour l'algorithme EDF est d'environ dix fois celui de la stratégie AbP. Ceci montre comment ces stratégies spécifiques de BGW peuvent améliorer considérablement la QdS résultante sous une surcharge transitoire élevée du processeur.

La stratégie $GbWp$ se comporte quasiment comme EDF lorsque la charge des primaires (U_p) augmente. Par conséquent, l'exécution des instances Wp avant les Wa conduit à une diminution très significative du succès global pour $GbWp$ comparablement avec $GbWa$ car tous simplement, les versions primaires ont des temps d'exécution plus élevés que ceux des secondaires.
Rappelons que $1/(ni+1) = 1/7 = 14.28\%$ et $1/(li+1) = 71.72\%$, ce qui représente respectivement les pourcentages d'instances Black et d'instances Grey qui doivent respecter leurs échéances.

Remarquons que même lorsque la charge des primaire U_p est élevée, toutes les politiques respectent la contrainte de distance n_i du modèle, c-à-d qu'elles exécutent au minimum, le plus petit nombre d'instances Black exigé par ce paramètre. Dans cette situation, les politiques BGW et EDF se comportent d'une façon

identique en termes de taux de succès. Néanmoins, il n'y a pas un moyen de contrôle de pertes d'échéances pour la stratégie EDF (les échéances violées interviennent à des instants non contrôlables) contrairement aux politiques propres à BGW.

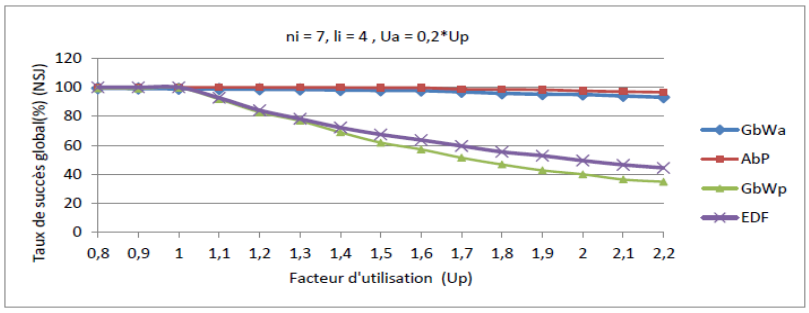


FIGURE 4.9 – NSJ pour des systèmes à fortes contraintes et à U_a léger.

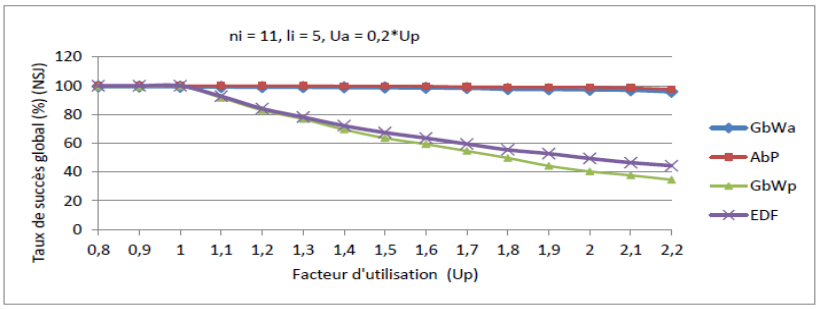


FIGURE 4.10 – NSJ pour des systèmes à moyennes contraintes et à U_a léger.

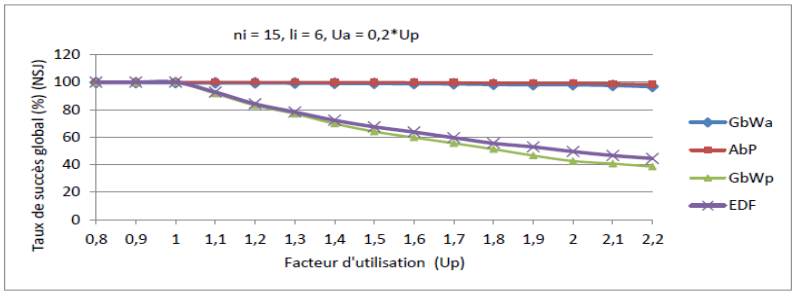


FIGURE 4.11 – NSJ pour des systèmes à faibles contraintes et à U_a léger.

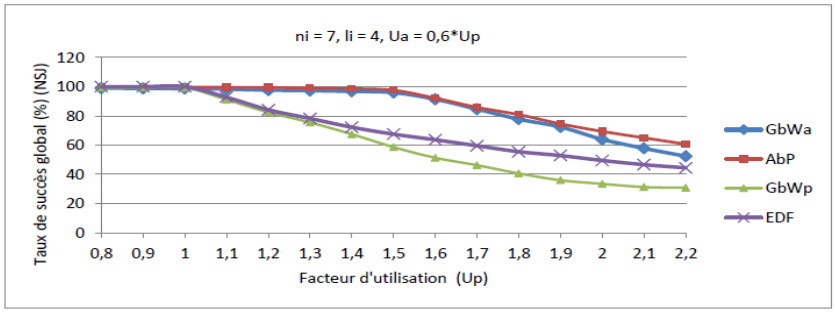


FIGURE 4.12 – NSJ pour des systèmes à fortes contraintes et à U_a lourd.

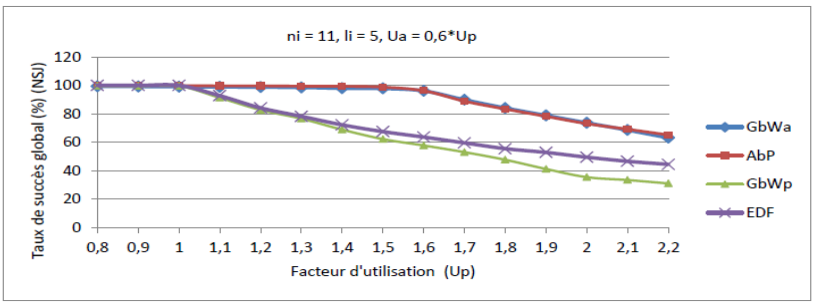


FIGURE 4.13 – NSJ pour des systèmes à moyennes contraintes et à U_a lourd.

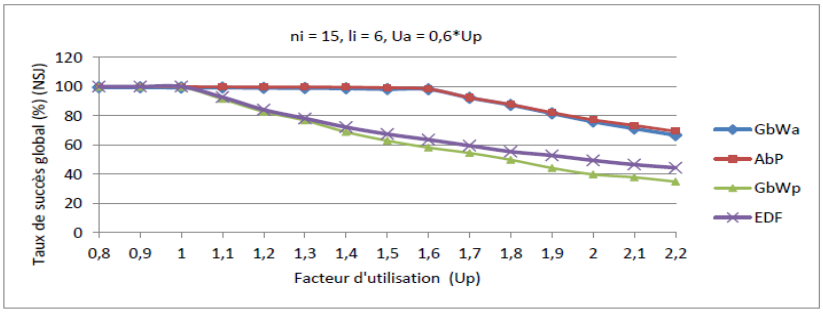


FIGURE 4.14 – NSJ pour des systèmes à faibles contraintes et à U_a lourd.

4.8.3 Analyse du taux de succès des primaires NPJ

Les figures 4.15 à 4.20 représentent les variations de NPJ qui désigne le pourcentage d'échéances respectées uniquement par les primaires.
Pour toutes les valeurs de U_p, la stratégie $GbWp$ offre une performance meilleure qu'un EDF classique.

Sous la stratégie $GbWp$, le NPJ est supérieur à celui de la stratégie $GbWa$ qui est, à son tour, supérieur à celui de AbP.

Il est très intéressant de noter que le NPJ reste plus grand ou égal à $1/n_i = 1/7 = 14.28\%$ d'instances, ce qui correspond au taux d'exécution le plus petit, exigé par la première contrainte du modèle BGW. Donc, nous avons à exécuter au moins $1/(ni)$ des instances par la version primaire. Dans cette étude, nous avons $(ni = 7)$, donc nous devons exécuter au moins $1/7 = 14.28\%$. Cela explique que nous pouvons poursuivre les expériences de simulation jusqu'à ce que $U_p = 700\%$ afin d'observer la diminution de NPJ jusqu'à moins de 14.28%. Vu que cette valeur est grande, nous avons décidé d'arrêter les simulations au niveau du tiers de cette charge.

Il est également intéressant de noter que le comportement d'EDF en termes de NPJ continue à diminuer rapidement tant que U_p augmente, jusqu'à ce qu'il devienne comparable à celui des stratégies AbP et $GbWa$. En fait, pour des taux d'utilisation processeur plus élevés, par exemple, entre $U_p = 210\%$ et $Up = 220\%$ nous observons une très légère différence entre les taux de succès des primaires pour toutes les stratégies.

En regardant les figures 4.1, 4.3, 4.5 qui indiquent les ordres d'exécution entre listes pour les différentes politiques de BGW avec l'utilisation de la technique de la première chance, nous pouvons facilement comprendre que lorsque les temps de calcul des Ga et des Wa tend vers zéro, le comportement de l'ensemble des différentes stratégies d'ordonnancement BGW tend à être le même que celui d'un EDF simple.

Nous remarquons également que tant que la priorité attribuée aux versions secondaires Wa est inférieure, les comportements des politiques BGW et EDF sont proches, en termes de taux de primaires réussis.
Lorsque la charge totale des primaires devient très lourde (plus de 210%), toutes les politiques BGW ne peuvent exécuter que quelques unes des instances Black. Dans ce cas, les politiques BGW et EDF se comportent presque identiquement en termes de NPJ. La même chose peut être observée lorsque nous avons des charges secondaires égales ou approximatives aux charges primaires.

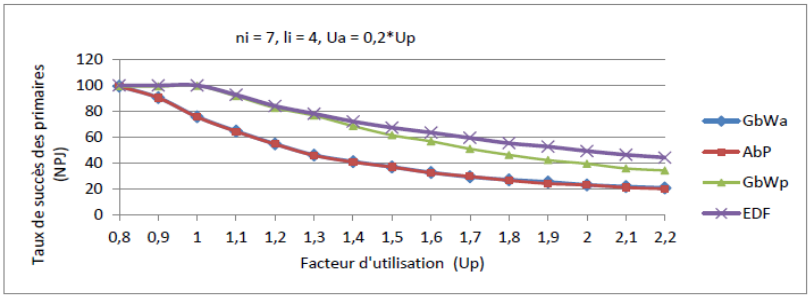


FIGURE 4.15 – NPJ pour systèmes à fortes contraintes et à U_a léger.

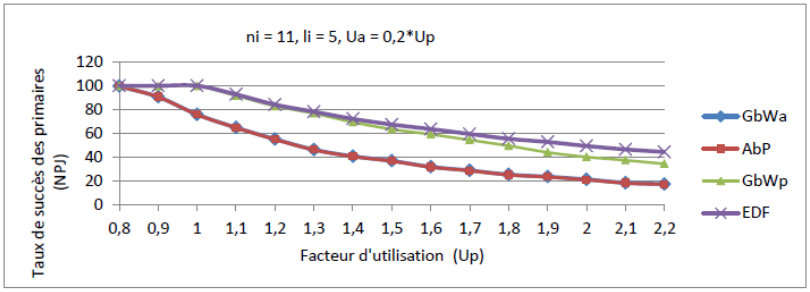


FIGURE 4.16 – NPJ pour systèmes à moyennes contraintes et à U_a léger.

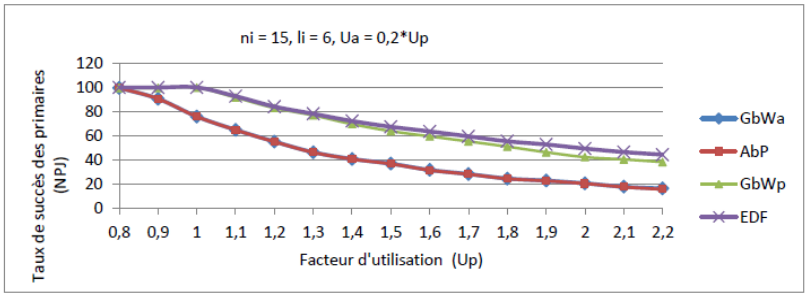


FIGURE 4.17 – NPJ pour systèmes à faibles contraintes et à U_a léger.

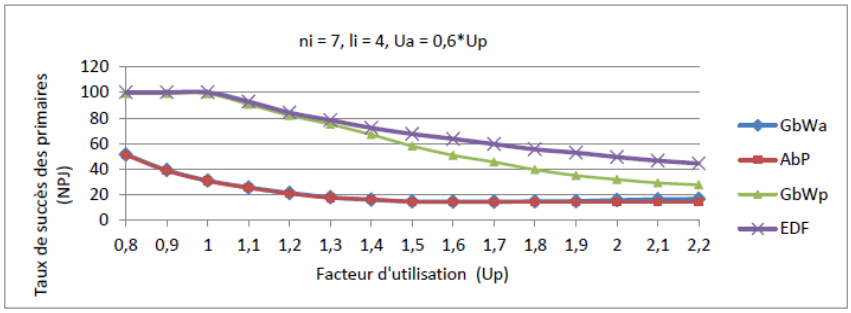


FIGURE 4.18 – NPJ pour systèmes à fortes contraintes et à U_a lourd.

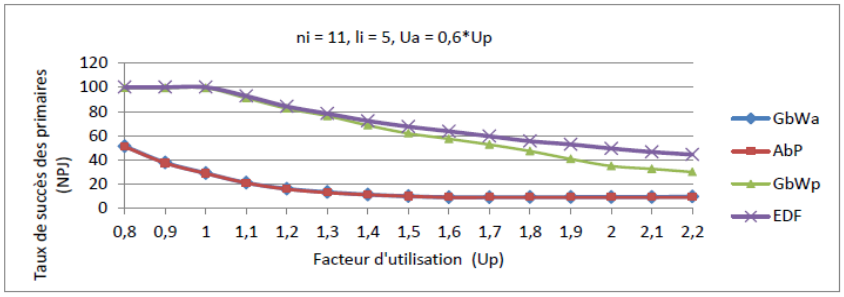


FIGURE 4.19 – NPJ pour systèmes à moyennes contraintes et à U_a lourd.

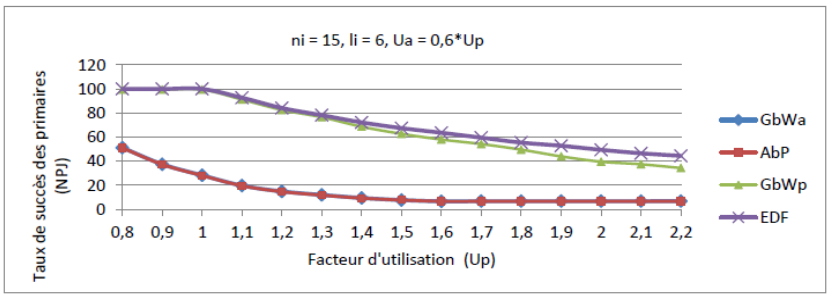


FIGURE 4.20 – NPJ pour systèmes à faibles contraintes et à U_a lourd.

4.8.4 Analyse du temps processeur gaspillé WTR

Le pourcentage de temps gaspillé (WTR) reflète le temps utilisé par le processeur pour ne produire aucun résultat ou produire des résultats inutiles.
Le système utilisant la stratégie d'ordonnancement EDF gaspille du temps dans l'exécution des primaires non terminés à échéance par manque de temps. Pour les politiques BGW, le temps gaspillé est lié à l'exécution des versions primaires des instances Grey et à l'exécution de toutes les versions des instances White.

Les figures 4.21 à 4.26, montrent les variations de WTR. Nous remarquons que pour toutes les valeurs de U_p, la stratégie $GbWa$ offre la meilleure performance en termes de WTR.

Dans des conditions de sous-charge, le WTR est quasiment égal à zéro pour l'ordonnanceur EDF puisque seules les versions primaires sont ordonnancées par EDF et qu'il est un ordonnanceur optimal dans l'absence de surcharge.
Pour $GbWp$, le WTR est aussi égal à zéro jusqu'à $U_p = 110\%$ et augmente avec des valeurs plus petites jusqu'à $U_p = 130\%$. Puis, il continue d'augmenter avec de grandes valeurs jusqu'à 21% au maximum $= 220\%$ de charge.

A des valeurs élevées de la charge primaire U_p, le WTR sous EDF est plus grand que celui sous AbP et sous $GbWa$.

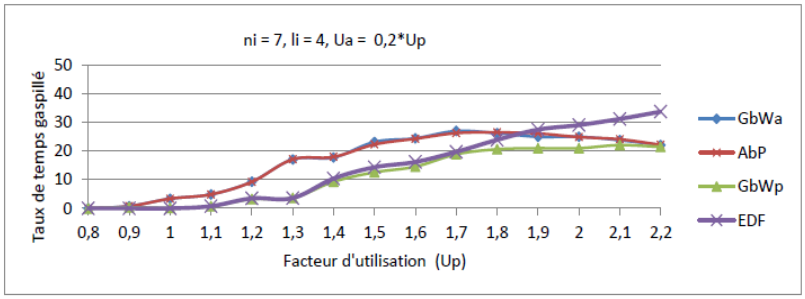


FIGURE 4.21 – WTR pour systèmes à fortes contraintes et à U_a léger.

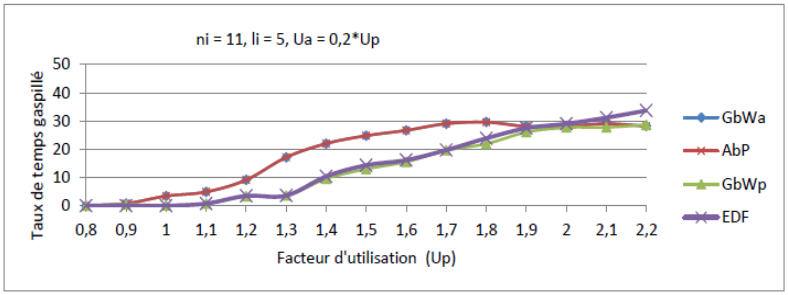


FIGURE 4.22 – WTR pour systèmes à moyennes contraintes et à U_a léger.

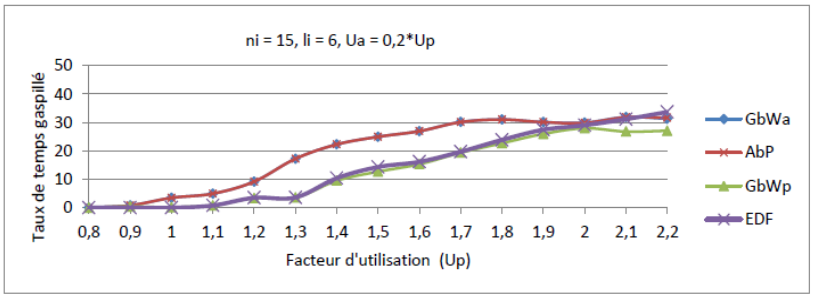


FIGURE 4.23 – WTR pour systèmes à faibles contraintes et à U_a léger.

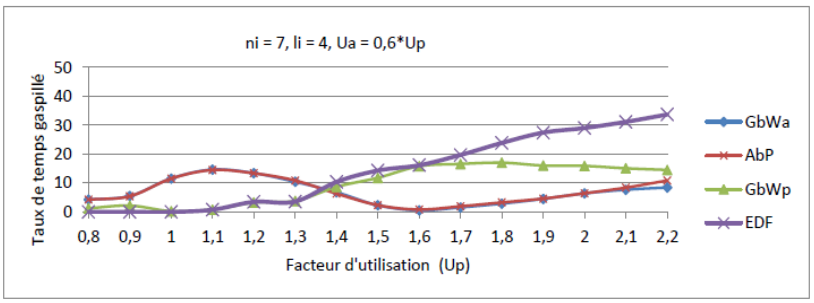


FIGURE 4.24 – WTR pour systèmes à fortes contraintes et à à U_a lourd.

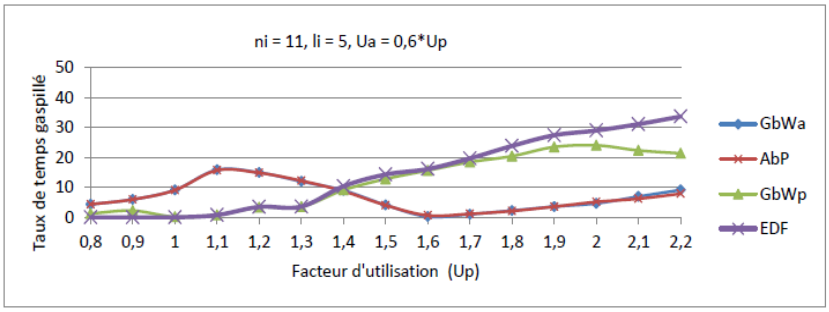


FIGURE 4.25 – WTR pour systèmes à moyennes contraintes et à U_a lourd.

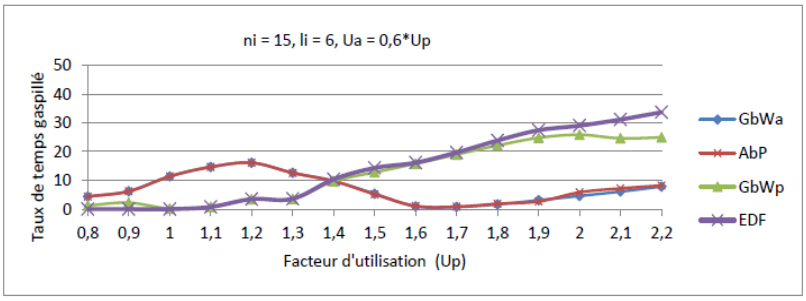


FIGURE 4.26 – WTR pour systèmes à faibles contraintes et à à U_a lourd.

4.8.5 Analyse du taux de préemption PR

Les mesures de QdS deviennent sans signification réelle si nous ne mesurons pas aussi la complexité de mise en oeuvre des différents ordonnanceurs proposés. Cette complexité porte d'abord sur le nombre de

changements de contexte engendrés qui chacun se caractérise aussi par des temps processeur perdus. Ceux-ci peuvent ainsi provoquer des retards dans l'exécution effective des tâches.

Les figures 4.27 à 4.32, montrent les variations du taux de préemption (PR).

Sous AbP et $GbWa$, PR est plus grand que sous $GbWp$ et EDF. Pour la politique EDF, les préemptions se produisent entre toutes les instances en fonction de leur échéance respective. Leur nombre est donc identique à un moment donné au nombre d'instances dans $B \bigcup G \bigcup W$.

Pour les politiques BGW, nous avons des préemptions au sein de chaque liste d'instances comme sous EDF mais en plus entre les listes. Par exemple, prenons une instance qui se réveille avec une échéance inférieure à toutes les autres instances prêtes. Avec EDF, l'arrivée de cette instance ne crée pas de préemption car toutes les instances sont ordonnées par échéance. Cette nouvelle instance ne provoquera aucune préemption car la moins prioritaire. Supposons que cette instance soit black. L'arrivée de celle-ci va entraîner avec BGW la préemption soit d'une instance Grey soit d'une instance White en exécution. Il existe en effet les préemptions supplémentaires suivantes : Préemptions de B sur G et W, et préemptions de G sur W. Ces préemptions sont issues d'instances sur des instances plus prioritaires au sens de l'urgence mais moins prioritaire au sens de la Qualité de Service.

C'est pour cette raison que le PR pour les politiques BGW, en particulier AbP et $GbWa$, est plus élevé que celui de l'algorithme classique EDF. Concernant $GbWp$, comme il n'y a pas de liste Wa, le PR se rapproche de celui du EDF classique. Lorsque le temps de calcul des secondaires tend à être proche de celui des primaires, le comportement des politiques BGW tend à se rapprocher de celui de EDF.

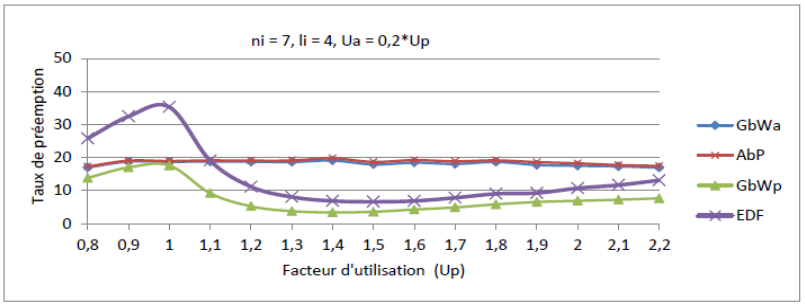


FIGURE 4.27 – PR pour systèmes à fortes contraintes et à U_a léger.

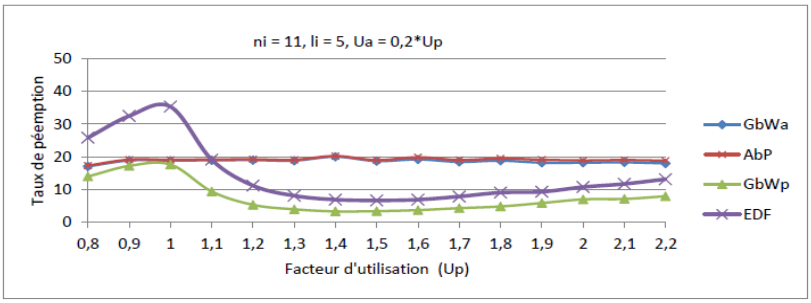


FIGURE 4.28 – PR pour systèmes à moyennes contraintes et à U_a léger.

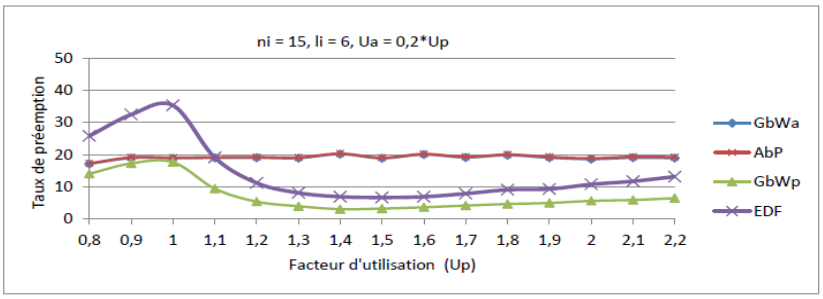


FIGURE 4.29 – PR pour systèmes à faibles contraintes et à U_a léger.

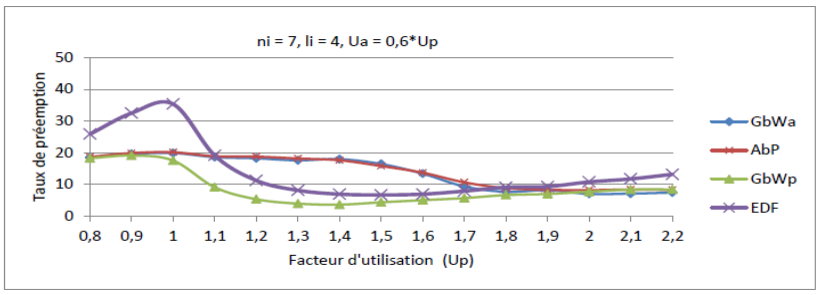


FIGURE 4.30 – PR pour systèmes à fortes contraintes et à U_a lourd.

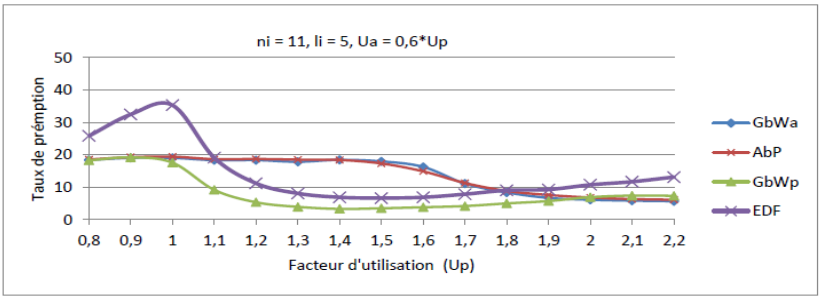


FIGURE 4.31 – PR pour systèmes à moyennes contraintes et à U_a lourd.

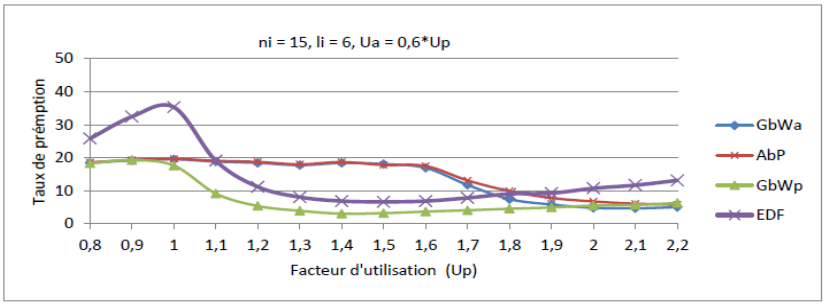


FIGURE 4.32 – PR pour des systèmes à faibles contraintes et à U_a lourd.

4.8.6 Bilan de l'analyse

L'étude des simulations des systèmes de tâches BGW avec une charge de secondaires proportionnelle à la charge de primaires, nous a montré l'utilité de l'approche BGW dans la gestion de la QdS.

On peut donc faire les remarques suivantes :

- Lorsque l'objectif de la conception d'une application est d'avoir une QdS1 maximale, (maximum d'instances qui s'exécutent avant leurs échéances), qui fait donc abstraction de la précision des calculs effectués par les tâches, la stratégie d'ordonnancement AbP s'avère la plus adaptée.

- Dans le cas où l'objectif est plutôt d'avoir une QdS2 maximale (maximum d'exécutions de versions primaires), c'est à dire optimisant le nombre de résultats présentant la plus grande précision, la stratégie $GbWp$ devient la meilleure.

- Dans le cas où l'objectif est d'avoir à la fois une QdS1 et une QdS2 acceptables, la stratégie d'ordonnancement $GbWa$ s'avère la plus adaptée.

4.9 Analyse de performances avec charge secondaire fixe

Pour étudier une gamme la plus large possible d'applications temps réel, nous allons, dans le paragraphe suivant, simuler des applications temps réel où le taux de charge totale des versions secondaires du système est fixé. Nous allons choisir pour cette étude la stratégie d'ordonnancement $GbWa$, vu son utilité pour améliorer à la fois de la QdS1 et de la QdS2.

4.9.1 Configuration de l'expérience

Nous choisissons trois valeurs de U_a qui sont respectivement 0.1, 0.4 et 0.7, indépendantes de la charge des primaires. Nous examinons 14 valeurs U_p qui varient uniformément de $0,8$ à $2,2$. Nous utilisons dans cette étude de simulation, les mêmes métriques que dans l'étude précédente, à savoir : (NPJ), (NJJ), (WTR), (RPC).

4.9.2 Analyse du taux de succès NSJ

La figure 4.33 montre le taux global d'échéances satisfaites pour la stratégie $GbWa$.
Pour de fortes charges, la politique $GbWa$ offre une meilleure performance comparativement à l'algorithme EDF.
Ainsi, il est intéressant de remarquer que plus la charge secondaire est petite, plus le rendement obtenu par la stratégie $GbWa$ est meilleur par rapport à EDF.

La stratégie classique EDF provoque plus de pertes d'échéances par rapport à la politique $GbWa$, avec les différentes valeurs U_a. Ceci est dû au fait que la politique EDF n'ordonnance que les versions primaires des tâches. Evidemment, ces versions demandent un temps processeur très grand dans le cas d'une surcharge de processeur. Ce résultat confirme alors l'utilité de l'approche BGW dans la gestion de surcharge.

Pour la politique $GbWa$, le pourcentage d'échéances perdues augmente tant que les valeurs de U_a et de U_p augmentent. Une grande valeur de U_a provoque plus de consommation de temps processeur par les versions secondaires d'instances Grey. Donc plus de versions primaires de ces instances Grey vont être perdues.

Sous la politique $GbWa$ avec $U_a = 10\%$, le pourcentage de réussite global est toujours supérieur à celui de $GbWa$ avec $U_a = 40\%$, qui est à son tour toujours supérieur à celui avec $U_a = 70\%$. Cela est dû tout simplement au fait que, avec des versions secondaires légères, il y aura plus de possibilités d'exécution des versions d'instances Grey et White. On peut dire que l'impact du temps de calcul des versions secondaires sur la performance de la politique $GbWa$ en termes de QdS1 est significatif.

Au delà de $U_p = 100\%$, le pourcentage global de réussite sous EDF commence à diminuer fortement jusqu'à ce qu'il atteigne une valeur de 50% d'échéances perdues lorsque U_p devient égale à 200%. Nous observons par contre, que sous la politique $GbWa$, la diminution du pourcentage de réussite globale est négligeable, quelle que soit la valeur de la charge totale des secondaires, et ceci jusqu'à ce que $U_p = 160\%$ où il commence à diminuer considérablement, dans le cas de $U_a = 70\%$.

Avec des niveaux de charge très élevés, le taux de pertes pour EDF est environ le double de celui de la politique $GbWa$ avec $U_a = 70\%$. Ceci illustre encore l'intérêt de l'approche BGW même si la charge totale des secondaires est grande.
Selon l'ordre d'exécution entre listes d'instances sous $GbWa$, adopté dans cette étude (voir la figure 4.5), lorsque les temps de calcul des Ga et des Wa tendent à être égaux à ceux des Gp et des Wp, le comportement de la politique $GbWa$ en termes de QdS1 tend à être plus proche de celui de EDF.

Lorsque la charge totale des secondaires et celle des primaires augmentent, la possibilité d'exécution des Gp diminue et par conséquent, celle des Ga aussi.
Ce qui implique aussi qu'il n'y aura pas effectivement, une grande possibilité pour l'exécution des instances Wa. Par conséquent, le taux global de réussite sera très considérablement diminué. Ainsi, lorsque aucune instance White n'est exécutée, cela veut dire qu'il y a un pourcentage de 50% de perte d'échéances. Rappelons que nous avons dans cette étude, un pourcentage minimum égal à $\frac{1}{n_i} = 25\%$ d'instances Black, $\frac{1}{n_i}.\frac{n_i-1}{l_i}$ d'instances Grey et par suite 50% d'instances White.

Lorsque la charge totale des primaires U_p est extrêmement lourde, la politique $GbWa$ ne va permettre que l'exécution des instances Black. Dans ce cas, les politiques EDF et $GbWa$ seront exactement identiques. Elles auront le même comportement en termes de taux global de réussite d'échéances c'est à dire de QdS2. Nous pouvons aussi faire la même remarque lorsque nous avons une charge totale des secondaires approximativement égale à la charge totale des primaires.

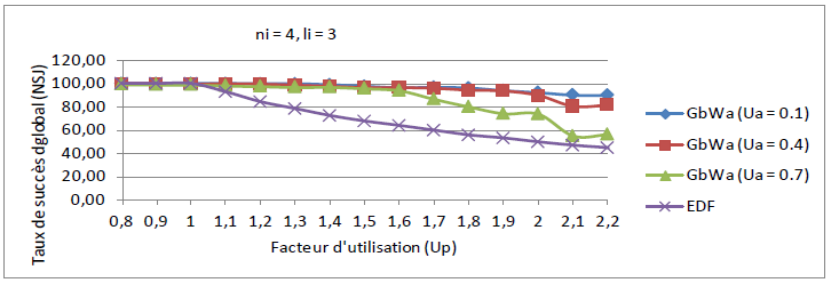


FIGURE 4.33 – NSJ pour systèmes à U_a fixe.

4.9.3 Analyse du taux de succès NPJ

L'axe horizontal représente la charge des primaires tandis que l'axe vertical représente le taux de succès par des primaires, NPJ.
La politique EDF offre une meilleure performance pour toutes les conditions de charges car on n'exécute pas les deux versions d'instances Grey et White, par la ré-exécution de la version primaire après avoir exécuté la version secondaire correspondante.
Pour $GbWa$, à $U_a = 10\%$, le pourcentage de succès des primaires est supérieure à celui avec $U_a = 40\%$, qui est supérieur à celui de $U_a = 70\%$.
Avec une grande valeur de U_a, il y a peu de chances de ré-exécution des versions primaires des instances Grey et White.
Nous observons une petite différence entre EDF et BGW-EDF quand $U_a = 10\%$, en terme de primaires réussis. Donc, nous pouvons dire que la stratégie BGW-EDF a une performance comparable avec EDF, lorsque nous avons des versions secondaires de charge légère.

Il est intéressant de noter que le pourcentage de succès des primaires reste supérieur à 25% des instances. Ce qui correspond au minimum acceptable selon la première contrainte du modèle BGW. Autrement dit, nous devons exécuter au moins, $1 = (ni + 1)$ des instances par leurs versions primaires. Dans cette étude, nous avons choisi $(ni = 3)$, ce qui veut dire que nous devons exécuter au moins $1 = (3 + 1) = 25\%$. C'est ce qui explique que nous pouvons continuer les expériences de simulation jusqu'à $U_p = 400\%$ afin de voir une diminution du ratio de succès des primaires à une valeur moins que 25%. Vue que cette valeur est très grande, nous avons choisi de terminer la simulation à la moitié de cette charge.

Il est également intéressant de noter que la performance d'EDF continue de diminuer rapidement, jusqu'à ce qu'elle devienne comparable à celle de $GbWa$, même lorsque la charge secondaire est plus élevée.

A des valeurs plus élevées du facteur utilisation U_p par exemple, entre $U_p = 210\%$ et jusqu'à $U_p = 220\%$ nous avons à peu près les mêmes valeurs pour toutes les stratégies d'ordonnancement.

La figure 4.3 indique l'ordre d'exécution des instances selon la stratégie $GbWa$, avec l'utilisation de la technique de la première chance. Nous pouvons facilement comprendre que, lorsque le temps d'exécution de Ga et Wa tend vers zéro, le comportement de la politique $GbWa$ tend à être exactement le même que l'ordonnanceur classique EDF.
En d'autres termes, plus les charges secondaires sont petites, plus les comportements de $GbWa$ et EDF en termes de primaires réussis est proche.

Lorsque la charge des primaires est très lourde, la politique $GbWa$ ne peut exécuter que des instances Black. Dans ce cas, les comportements des politiques $GbWa$ et EDF seront exactement identiques en termes de succès des primaires. Nous pouvons dire la même chose dans le cas où nous avons une charge des secondaires approximativement égale à la charge des primaires.

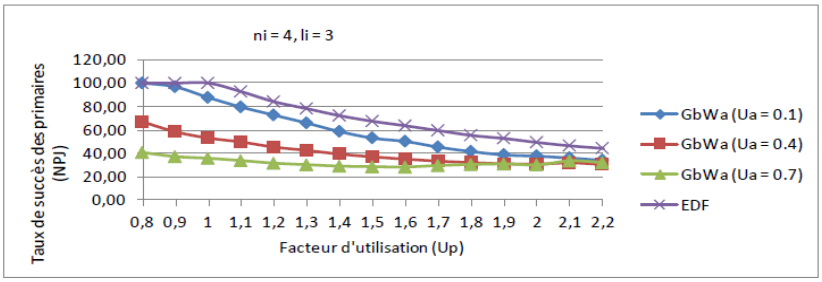


FIGURE 4.34 – NPJ pour systèmes à U_a fixe.

4.9.4 Analyse du temps processeur gaspillé WTR

La figure 4.35 montre les variations du temps gaspillé sous les stratégies $GbWa$ et EDF pour différentes valeurs de charges secondaires U_a. Nous remarquons que :

- au contraire de $GbWa$, WTR continue d'augmenter dans les situations de surcharge pour la stratégie classique EDF.

- à petits niveaux de charges, nous remarquons que EDF offre une meilleure performance en termes de temps gaspillé. Dans des conditions de sous-charge, le gaspillage du temps est égal à 0 pour l'algorithme EDF, en raison du fait que EDF ordonnance seulement des versions primaires et qu'il est optimal lorsqu'il n'y a pas de surcharge.

- WTR a des valeurs très petites jusqu'à $U_p = 150\%$ où l'on observe une forte augmentation de ce ratio jusqu'à 38% pour $= 220\%$ de charge. En effet, lorsqu'on abandonne l'exécution d'une version primaire ou secondaire d'une tâche lourde que nous avons déjà commencée, nous gaspillons de la ressource processeur.

- Comme le montre la figure 4.35, le ratio WTR augmente lorsque la charge des primaires U_p augmente pour la politique EDF plus que pour la stratégie $GbWa$. Le ratio de temps gaspillé avec EDF est plus élevé qu'avec $GbWa$, lorsque la charge des primaires est très lourde.

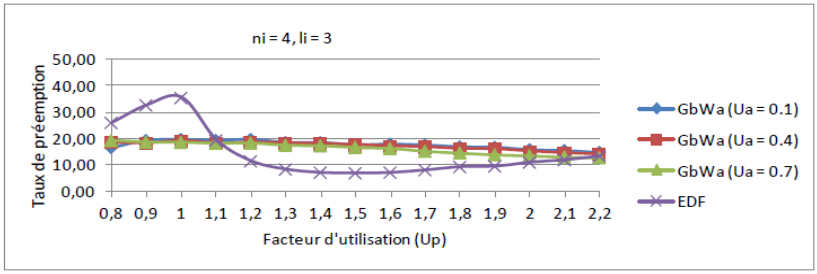


FIGURE 4.35 – WTR pour systèmes à U_a fixe.

4.9.5 Analyse du taux de préemption PR

Les résultats présentés dans la figure 4.36, montrent le taux de préemption pour EDF et $GbWa$.
Globalement, nous pouvons constater que :

- le comportement de EDF et $GbWa$ sont similaires jusqu'à $U_p = 160\%$. Ceci est dû au fait que pour les tâches lourdes (U_a plus grande), la tâche prend du temps à exécuter des versions secondaires de ses instances Grey. Par conséquent, le nombre de G_p, W_a et W_p sélectionnées pour être exécuté diminue.

- le taux de préemption est à peu près constant à 19%. Au-dessus de $U_p = 160\%$, tant que la charge U_a augmente, le pourcentage de préemption diminue très légèrement, pour la stratégie $GbWa$.

- pour des niveaux de charges plus petits, nous constatons que la stratégie $GbWa$ offre des meilleures performances. RPC reste toujours inférieur à 20% dans toutes les conditions. Il continue à diminuer lorsque la charge augmente jusqu'à ce qu'il atteigne 15% pour $U_p = 220\%$.

- à partir de 150%, il commence à augmenter jusqu'à ce qu'il atteigne le même niveau que celui sous la politique $GbWa$. Cette augmentation est dûe au fait que les tâches très lourdes sont plus susceptibles de manquer leurs échéances.

Globalement, nous pouvons dire que le taux de préemption est quasiment constant pour la politique $GbWa$ et est variable pour la stratégie EDF. La différence entre ces deux comportements peut être facilement expliquée comme suit : Sous $GbWa$, nous exécutons les versions secondaires, puis les versions primaires des instances Grey puis les instances White. Ainsi, pour la politique $GbWa$, il y a plus de deux possibilités de préemptions par rapport à la politique classique EDF, à savoir droit de préemption de Ga sur Gp et droit de préemption de Wa sur Wp. C'est pour cela que le pourcentage de préemption sous $GbWa$ est plus élevé que celui sous EDF.
Lorsque la charge totale des versions primaires devient très lourde, la possibilité d'exécution des instances G_p diminue. On ne peut exécuter ni W_a ni W_p. Ce qui implique une diminution du nombre de préemptions sous $GbWa$.
En regardant l'ordre d'exécution entre listes d'instances sous $GbWa$ (voir la figure 4.5), nous constatons que lorsque les temps de calcul des secondaires se rapprochent des temps des primaires, le comportement en termes de taux de préemption de $GbWa$ tend vers celui de EDF.
Lorsque la charge totale des primaires U_p devient lourde, $GbWa$ ne trouve pas de possibilité pour l'exécution des instances White et des versions Gp. Par conséquent, les politiques $GbWa$ et EDF seront exactement identiques en terme de taux de préemption.

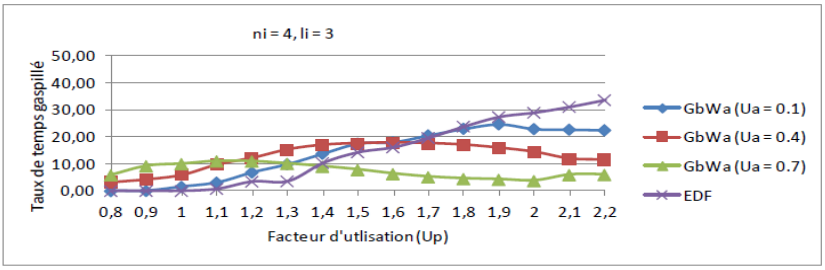


FIGURE 4.36 – PR pour systèmes à U_a fixe.

Nous résumons les résultats de performances des différentes stratégies proposées dans ce chapitre. On note par * une performance faible, ** une performance correcte et *** une trés bonne performance.

Algorithme	$QdS1$	$QdS2$
$GbWa$	**	**
AbP	***	*
$GbWp$	*	***
EDF	*	**

TABLE 4.1 – Synthèse des performances de stratégies d'ordonnancement basiques.

4.10 Conclusion

Dans ce chapitre, nous avons proposé d'étudier le comportement de trois ordonnanceurs basés sur EDF et spécifiquement adaptés au modèle présenté dans le chapitre précédent. Le modèle BGW, a la particularité de permettre à l'exécution, le contrôle des manquements d'échéances ainsi que le contrôle de la qualité des résultats produits, plus particulièrement pour les systèmes temps réel fermes surchargés.

Nous avons rappelé les deux principaux critères de Qualité de Service d'un système temps réel ferme où les tâches respectent les spécifications du modèle BGW. Il s'agit du pourcentage d'échéances respectées par l'ensemble des instances réveillées (QdS1) d'une part, et du pourcentage d'échéances respectées par les versions primaires (QdS2) d'autre part. Nous avons décrit notre environnement de simulation pour comparer quatre ordonnanceurs, trois dédiés à BGW plus EDF qui ordonnance des tâches classiques mono-version (la version primaire).

L'étude de simulation permet de justifier l'existence et l'utilisation du modèle BGW pour les raisons suivantes :

- les stratégies d'ordonnancement AbP et $GbWa$ offrent de très bonnes performances en termes de QdS1 et la stratégie d'ordonnancement $GbWp$ offre de bonnes performances en termes de QdS2.
- Concernant le temps processeur gaspillé, la stratégie d'ordonnancement $GbWp$ offre de bonnes performances pour les systèmes avec versions secondaires légères et les stratégies d'ordonnancement AbP et $GbWa$ offrent des bonnes performances pour les systèmes avec versions secondaires lourdes.

- Concernant les overheads dus aux préemptions, la stratégie d'ordonnancement $GbWp$ offre des très bonnes performances quelque soit le taux de charge secondaire et les stratégies d'ordonnancement AbP et $GbWa$ offrent des bonnes performances aux systèmes avec versions secondaires lourdes, dans les conditions de surcharge très élevées.

L'étude de simulation a aussi permis de mettre en évidence :

- L'ordonnanceur AbP apparait comme le plus performant du point de vue du premier critère de QdS. La QdS1 obtenue par rapport à la QdS spécifiée est de l'ordre de 98% d'instances, qui exécutent dans l'échéance l'une de ses versions primaires ou secondaires.
- L'ordonnanceur $GbWp$ apparait comme le plus performant du point de vue du second critère de QdS. La QdS2 obtenue par rapport à la QdS spécifiée est de l'ordre de 40% d'instances, qui exécutent dans l'échéance leurs versions primaires.

Les ordonnanceurs étudiés, bien que simples à mettre en oeuvre donnent une QdS effective tout à fait intéressante et supérieure à celle de EDF. Néanmoins, lorsque l'on cherche à obtenir une meilleure QdS, il devient nécessaire de se tourner vers des méthodes plus complexes. C'est pourquoi, dans le chapitre suivant, nous proposons de substituer la technique de la première chance à la technique de la dernière chance, afin de tendre vers la maximisation des deux critères de Qualité de Service, QdS1 et QdS2.

5

Ordonnancements évolués pour BGW

5.1 Introduction

Dans ce chapitre, notre objectif vise à proposer et évaluer de nouveaux ordonnanceurs afin d'obtenir une Qualité de Service meilleure que celle fournie par les ordonnanceurs exposés au chapitre précédent. En effet, l'on peut penser qu'en se basant sur la technique de la dernière chance plutôt que la technique de la première chance, nous minimiserons le temps processeur gaspillé et favoriserons l'exécution avant échéance des versions primaires. Nous décrirons plusieurs stratégies d'ordonnancement toutes ayant comme critère de priorité, l'urgence, avec EDF. Une étude de simulation permettra de donner des mesures de performance comparatives avec les stratégies vues au précédent chapitre.

5.2 Description des ordonnanceurs

Rappelons que la technique de la dernière chance initialement conçue pour le modèle du Mécanisme à Échéance, consiste à tenter d'exécuter en premier lieu la version primaire d'une tâche tout en garantissant l'exécution de la version secondaire de cette tâche avant échéance au cas où le primaire échouerait. En d'autres termes, on applique l'ordonnancement EDS aux primaires pour les exécuter au plus tôt et l'ordonnancement EDL aux secondaires pour les exécuter au plus tard. Cette technique est ainsi particulièrement adaptée aux systèmes surchargés (le primaire n'a parfois pas le temps de terminer son exécution). De plus, elle convient aux systèmes dits tolérants aux fautes où la version primaire peut faire l'objet d'une erreur de codage, d'un blocage permanent, etc. l'empêchant de terminer son exécution avec succès et nécessitant une redondance logicielle.

Sous le modèle BGW, la technique de la dernière chance peut s'utiliser de plusieurs manières selon les instances que l'on choisit d'exécuter au plus tard, les instances que l'on choisit d'exécuter au plus tôt puis l'ordre de priorités entre les listes d'instances.
Les instances Black devant impérativement s'exécuter, il apparaît logique de garantir leur exécution en toutes circonstances, ce qui nous amène à pré-réserver leur intervalle d'exécution par EDL. Concernant les instances Grey, nous devons garantir l'exécution de la version secondaire après avoir tenté l'exécution du primaire. Lorsque celui-ci se termine, alors son secondaire est supprimé, conduisant à la récupération de temps processeur. En résumé, les instances B et Ga seront exécutées au plus tard et ordonnées entre elles

avec la règle EDF. Rappelons qu'en procédant de la sorte, à tout moment le temps processeur disponible réservé à toutes les autres versions de tâches prêtes se trouve optimisé, c'est à dire des Gp, Wp et Wa.

Apparaissent ainsi plusieurs possibilités pour gérer conjointement ces listes de versions. Nous avons opté pour les deux suivantes :

- Choix 1 : exécuter d'abord les versions primaires des instances Grey, Gp, puis les versions primaires des instances White, Wp.
- Choix 2 : exécuter d'abord les versions primaires des instances Grey, Gp, puis les Wa suivies des Wp.

Pour ces deux choix, lorsqu'une instance Gp ou Wp est exécutée avec succès, la version secondaire Ga ou Wa correspondant à la même tâche est supprimée et son temps pré-alloué récupéré. Vu la plus grande importance des Gp par rapport aux Wp ou Wa, nous n'avons pas considéré une politique visant à exécuter les instances White prioritairement aux Grey.
La technique de la dernière chance étant utilisée, nous avons obligatoirement recours aux méthodes de calcul et de détermination des intervalles de temps creux EDL.

Les deux choix précédents correspondent respectivement aux stratégies que nous appellerons LCP (Last Chance technique for success of Primaries) et LCJ (Last Chance technique for success of Jobs). Nous allons montrer que LCP augmente la $QdS2$ et que LCJ augmente la $QdS1$.

5.3 Technique de dernière chance pour le modèle BGW

La technique de la dernière chance étant utilisée que ce soit avec LCP ou avec LCJ, nous avons obligatoirement recours aux méthodes de calcul et de détermination des intervalles de temps creux EDL développées dans [25] et [27]. Pour ces deux stratégies, nous appliquons la méthode de la dernière chance aux instances B et Ga dont l'exécution se doit d'être garantie en toutes circonstances.

Implémenter cette technique suppose que la séquence EDL sur les instances B et Ga soit mise à jour chaque fois que nécessaire. Ainsi, l'on doit connaître à tout instant la date à laquelle l'on doit commuter sur l'exécution d'une instance de type B ou Ga de façon à garantir le respect de leur échéance.
La séquence EDL sur l'ensemble des instances B et Ga de tout instant courant t jusqu'à la fin de l'hyper-période courante est précisément décrite par la donnée de deux vecteurs. Le premier contient la liste de toutes les échéances d'instances réveillées sur une hyper-période. A chacune de ces échéances, correspond une grandeur entière qui représente la longueur du temps creux processeur qui suit cette échéance. On dit alors que la séquence EDL est représentée par le *vecteur des échéances statique* et le *vecteur des temps creux statique*.

Montrons comment calculer le vecteur des temps creux statique.

Proposition 13 *Pour tout ordonnanceur qui garantit les instances B et Ga, le vecteur des temps creux statique $D = (\Delta_0, \Delta_1, ..., \Delta_j, \Delta_{j+1}, ..., \Delta_q)$*
d'un ensemble Γ de tâches BGW1, est déterminé par :
$\Delta_q = min\{T_i, 1 \leq i \leq n\}$,
$\Delta_j = sup(0, F_j)$, $i = q - 1$ *à* 0,
avec

$F_j = (P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a) - \sum_{k=j+1}^{q} \Delta_k$.
et $X = (P - k_j) - \lfloor \frac{P-k_j}{n_i.T_i} \rfloor.n_i.T_i + T_i$.

Preuve. Considérons pour chaque échéance k_j, $0 \leq j \leq q$, la durée du temps creux associée, notée Δ_j, pouvant être égale à zéro. Soit $p = ppcm(n_1.T_1, ..., n_i.T_i, ..., n_n.T_n)$. Pour chaque tâche τ_i, la dernière instance observée sur $[0, P]$ est blanche. Par conséquent, puisque k_q correspond à la dernière échéance Black ou Grey survenant sur $[0, H[$, Δ_q est donné par $P - k_q$ aussi égal à $min\{P_i, 1 \leq i \leq n\}$. Dans l'intervalle $[k_j, P]$, avec $0 \leq j \leq q$, le nombre total d'instances de toute nature (noire, grise et blanche), pour chaque tâche τ_i est égal à $\lceil \frac{P-k_j}{T_i} \rceil$.
Dans le cas où toutes ces instances terminent complètement leur exécution par les versions primaires dans $[k_j, P]$, le temps creux total sur $[k_j, P]$ sera égal à $(P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{T_i} \rceil.C_i^p$.
Dans le cas où chaque tâche accepte des dégradations en qualité selon la contrainte n_i du modèle BGW ou en quantité selon la contrainte l_i de ce modèle, c'est à dire chaque tâche doit exécuter nécessairement ses instances Black (B) et les versions secondaires de ses instances Grey (Ga), le nombre total d'exécutions se réduit à $\lceil \frac{P-k_j}{n_i.T_i} \rceil$ de versions primaires et de $\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor$ de versions secondaires,
d'où la durée totale de temps creux sur $[k_j, P]$ égale à
$(P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a)$.
avec $X = (P-k_j) - \lfloor \frac{P-k_j}{n_i.T_i} \rfloor.n_i.T_i + T_i$. Et comme la durée totale des temps creux sur $[k_{j+1}, P]$ est donnée par $\sum_{k=j+1}^{q} \Delta_k$, il vient que $\Delta_j = (P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a) - \sum_{k=j+1}^{q} \Delta_k$.
Sinon Δ_j est égale à 0. □

Le vecteur des temps creux statique est effectué hors-ligne. Montrons comment optimiser les calculs du vecteur des temps creux dynamique à n'importe quel instant t.

Proposition 14 *Le vecteur des temps creux dynamique* $D(t) = (\Delta_h(t), \Delta_{h+1}(t), ..., \Delta_j(t), \Delta_{j+1}(t), ..., \Delta_q(t))$ *se calcule comme suit :*
$\Delta_j(t) = \Delta_j$ *pour* $j = q$ *à* f
$\Delta_j(t) = sup(0, F_j(t))$ *pour* $j = f - 1h + 1$
avec :
$F_j = (P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a) + \sum_{i=1,d_j>k_j}^{n} A_i(t) - \sum_{k=j+1}^{q} \Delta_k(t)$.
et $X = (P - k_j) - \lfloor \frac{P-k_j}{n_i.T_i} \rfloor.n_i.T_i + T_i$
$\Delta_h(t) = (P - t) - \sum_{i=1}^{n}(\lceil \frac{P-t}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-t}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a - A_i(t)) - \sum_{k=h+1}^{q} \Delta_k(t)$ *avec* $X = (P - t) - \lfloor \frac{P-t}{n_i.T_i} \rfloor.n_i.T_i + T_i$.

Preuve. Calculons la durée du temps creux $\Delta_j(t)$ à l'instant k_j, avec $t \leq k_j \leq k_q$. Soit $A_i(t)$ la quantité d'exécution effectuée sur l'instance courante de la tâche τ_i et d_i l'échéance de cette instance. Nous définissons $M = sup\{d_i, \tau_i \in \Gamma(t)\}$ comme la plus grande échéance parmi celles des instances réveillées à l'instant t.
La séquence EDL calculée pour $\Gamma(t)$ sur $[k_f, P]$, avec $k_f = min\{k_j, k_j > M\}$, est identique à celle calculée hors ligne. Alors, $\Delta_j(t) = \Delta_j$, pour $j = q$ à f. Si $d_i < k$, la durée totale d'exécution demandée par τ_i sur $[k_j, P]$, en tenant compte de ses dégradations selon la contrainte n_i et de ses pertes selon la contrainte l_i, ou autrement dit pour l'exécution de toutes ses instances Black (B) et les versions secondaires des ses instances Grey (Ga), est donnée par :
$\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a$.
Sinon, elle est donnée par :
$\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a - A_i(t)$ vu que l'instance courante de τ_i a déjà exécuté une partie.

Par conséquent, la durée totale des temps creux sur l'intervalle $[k_j, P]$ est donnée par :
$(P - k_j) - \sum_{i=1}^{n}(\lceil \frac{P-k_j}{n_i.T_i} \rceil.C_i^p + (\lfloor \frac{P-k_j}{n_i.T_i} \rfloor.\lfloor \frac{n_i-1}{l_i} \rfloor + \lfloor \frac{X}{l_i.T_i} \rfloor).C_i^a) + \sum_{i=1,d_j>k_j}^{n} A_i(t)$. □

A chaque succès d'une instance Ga, la séquence EDL sera mise à jour par un re-calcul du vecteur des temps creux dynamique.

5.4 Stratégie d'ordonnancement LCP

La stratégie d'ordonnancement LCP se décrit comme suit :

- Chaque liste d'instances est ordonnancée par EDF.
- les instances Gp et les instances Wp sont exécutées au plus tôt.
- les instances B et les instances Ga sont exécutées au plus tard.

La figure 5.1 représente le pseudo-code de l'ordonnanceur LCP.

5.5 Stratégie d'ordonnancement LCJ

LCJ est introduite dans le but d'avoir un équilibrage entre le taux de respect d'échéances (QdS1) et le taux de respect d'échéances par les primaires (QdS2).
D'où, nous adoptons la définition suivante :

La stratégie d'ordonnancement LCJ se décrit comme suit :

- Chaque liste d'instances est ordonnancée par EDF.
- les instances Gp, Wa et Wp sont exécutées au plus tôt.
- les instances B et les instances Ga sont exécutées au plus tard.

La figure 5.2 représente le pseudo-code de l'ordonnanceur LCJ.

5.6 Stratégie d'ordonnancement $LCP - t$

La stratégie d'ordonnancement $LCP - t$ diffère uniquement de LCP sur le point suivant : avant d'admettre pour exécution toute instance Gp ou Wp, un test d'admission est mis en oeuvre. Celui-ci a pour objectif de vérifier que la durée d'exécution de l'instance sélectionnée est compatible avec une exécution faisable de celle-ci. En d'autres termes, l'instance ne sera lancée que si sa fin d'exécution intervient avant la date de démarrage de la séquence EDL, c'est à dire avant le démarrage au plus tard des instances B et Ga.

L'idée de ce test d'acceptation est de comparer le temps d'exécution de chaque instance Gp ou Wp dès son arrivée avec le temps processeur disponible dans l'intervalle $[t, d_j]$ après avoir soustrait le temps réservé aux instances B et Ga dans cet intervalle et le temps réservé aux autres instances Gp et Wp déjà acceptées.

Algorithm 4 Algorithme d'ordonnancement LCP

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
GpList, Liste des instances Gp prêtes ordonnancées entre elles selon EDF ;
WpList, Liste des instances Wp prêtes ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

begin
if Le temps creux à l'instant (t) != 0 **then**
/ Il y a un temps disponible à l'instant (t)*/*
if GpList.Head != NULL **then**
/ Présence des instances Gp prêtes pour s'exécuter */*
exécuter $GpList.Head$ et supprimer le, après exécution ;
supprimer l'instance Ga correspondante à l'instance Gp, de la liste $BGaList$;
else
/ Il n'y a pas des instances Gp prêtes pour s'exécuter */*
if WpList.Head != NULL **then**
/ Présence des instances Wp prêtes pour s'exécuter */*
exécuter $WpList.Head$ et supprimer le, après exécution ;
supprimer l'instance Wa correspondante à l'instance Wp, de la liste $WaList$;
else
if BGaList.Head != NULL **then**
/ Présence des instances B ou Ga prêtes pour s'exécuter */*
exécuter $BGaList.Head$ et supprimer le, après exécution ;
supprimer l'instance Gp correspondante à l'instance Ga, de la liste $GpList$;
end if
end if
end if
elseLe temps creux à l'instant (t) == 0
/ Il n'y a pas un temps disponible à l'instant (t)*/*
if BGaList.Head != NULL **then**
/ Présence des instances B ou Ga prêtes pour s'exécuter */*
exécuter $BGaList.Head$ et supprimer le, après exécution ;
supprimer l'instance Gp correspondante à l'instance Ga, de la liste $GpList$;
end if
end if
end

FIGURE 5.1 – Pseudo-code de LCP

Définition : La laxité $\omega_j(t)$ à l'instant t, d'une instance Gp ou Wp d'échéance d_j, est définie par :

$$\omega_j(t) = \Omega^{EDL}_{\Gamma(t)}(t, d_j) - \sum_{i=1}^{j} C_i^p(t) \tag{5.1}$$

$\Omega^{EDL}_{\Gamma(t)}(t, d_j)$ représente la somme totale des temps creux disponibles dans l'intervalle $[t, d_j]$.
La grandeur $\sum_{i=1}^{j} C_i^p(t)$ représente la somme des temps d'exécution restants à l'instant t sur l'ensemble des instances Gp et Wp déjà acceptées dont l'échéance est inférieure ou égale à d_j. Le test d'acceptation des instances Wp est identique à celui des Gp.

Algorithm 5 Algorithme d'ordonnancement LCJ

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
GpList, Liste des instances Gp prêtes ordonnancées entre elles selon EDF ;
WpList, Liste des instances Wp prêtes ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if Le temps creux à l'instant (t) != 0 then
    /* Il y a un temps disponible à l'instant (t)*/
    if GpList.Head != NULL then
        /* Présence des instances Gp prêtes pour s'exécuter */
        exécuter GpList.Head et supprimer le, après exécution ;
        supprimer l'instance Ga correspondante à l'instance Gp, de la liste BGaList ;
    else
        /* Il n'y a pas des instances Gp prêtes pour s'exécuter */
        if WaList.Head != NULL then
            /* Présence des instances Wa prêtes pour s'exécuter */
            exécuter WaList.Head et supprimer le, après exécution ;
        else
            /* Pas des instances Wa prêtes pour s'exécuter */
            if WpList.Head != NULL then
                /* Présence des instances Wp prêtes pour s'exécuter */
                exécuter WpList.Head et supprimer le, après exécution ;
            else
                /* Pas des instances Wp prêtes pour s'exécuter */
                if BGaList.Head != NULL then
                    /* Présence des instances B ou Ga prêtes pour s'exécuter */
                    exécuter BGaList.Head et supprimer le, après exécution ;
                    supprimer l'instance Gp correspondante à l'instance Ga, de la liste GpList ;
                end if
            end if
        end if
    end if
else Le temps creux à l'instant (t) == 0
    /* Il n'y a pas un temps disponible à l'instant (t)*/
    if BGaList.Head != NULL then
        /* Présence des instances B ou Ga prêtes pour s'exécuter */
        exécuter BGaList.Head et supprimer le, après exécution ;
        supprimer l'instance Gp correspondante à l'instance Ga, de la liste GpList ;
    end if
end if
end
```

FIGURE 5.2 – Pseudo-code de LCJ

5.6.1 Algorithme LCP-t

La figure 5.5 représente le pseudo-code de l'algorithme d'ordonnancement $LCP - t$.

Remarque :
Il est possible d'avoir des instances Wa qui s'exécutent alors que les instances correspondantes Wp ont été refusées par le test d'acceptation, dû à la différence possiblement très grande entre leur durée d'exécution.

Algorithm 6 Algorithme du test d'admission en ligne d'une instance Gp

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpWpList, Liste des instances Gp et Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, l'instant courant ;
Sorties : L'instance Gp est acceptée si $\omega_i(t)$ est supérieur à 0, non acceptée dans le cas contraire.

```
begin
while Une instance Gp arrive au temps t do
    acceptable = 1 ;
    chercher la plus grande échéance d_j de l'ensemble Gp_t ∪ AcceptedGpWpList ;
    calculer le temps creux total dans l'intervalle [t, d_j], f(t)^EDL(t, d_j) ;
    for toute instance τ_i de l'ensemble Gp_t ∪ AcceptedGpWpList telle que d_i ≥ d do
        calculer la laxité ω_i(t) de l'instance τ_i ;
        if ω_i(t) ≤ 0 then
            acceptable = 0 ;
        end if
    end for
    return admitted ;
end while
end
```

FIGURE 5.3 – Pseudo-code du test d'admission en ligne de Gp

Algorithm 7 Algorithme du test d'admission en ligne d'une instance Wp

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpWpList, Liste des instances Gp et Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;
Sorties : L'instance Wp_t est acceptée si $\omega_i(t)$ est supérieur à 0, non acceptée dans le cas contraire.

```
begin
while Une instance Wp_t arrive au temps (t) do
    acceptable = 1 ;
    chercher la plus grande échéance d_j de l'ensemble Wp ∪ AcceptedGpWpList ;
    caculer le temps creux dans l'intervalle [t, d_j], f(t)^EDL(t, d_j) ;
    for toute instance τ_i de l'ensemble Wp ∪ AcceptedGpWpList telle que d_i ≥ d do
        caculer la laxité ω_i(t) de l'instance τ_i ;
        if ω_i(t) ≤ 0 then
            acceptable = 0 ;
        end if
    end for
    return acceptable ;
end while
end
```

FIGURE 5.4 – Pseudo-code du test d'acceptation d'une instance Wp

5.7 Stratégie d'ordonnancement $LCJ-t$

Au lieu de tester l'acceptation des instances Gp et Wp (cas précédent), nous testons ici l'acceptation des instances Gp et Wa.

Algorithm 8 Algorithme d'ordonnancement $LCP - t$

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWpList, Liste des instances Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    exécuter AcceptedGpList.Head et supprimer le, après exécution ;
    supprimer l'instance Ga correspondante de l'instance Gp, de la liste BGaList ;
else
    /* Il n'y a pas des instances Gp acceptées pour être exécutée */
    if AcceptedWpList.Head != NULL then
        /* Présence des instances Wp acceptées pour être exécutée */
        exécuter AcceptedWpList.Head et supprimer le, après exécution ;
        supprimer l'instance Wa correspondante de l'instance Wp, de la liste WaList ;
    else
        /* Il n'y a pas d'instances Wp acceptées pour être exécutée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et supprimer le, après exécution ;
        else
            /* Pas d'instances B ou Ga prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                exécuter WaList.Head et supprimer le, après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 5.5 – Pseudo-code de $LCP - t$

5.7.1 Algorithme $LCJ - t$

La figure 5.6 représente le pseudo-code de l'ordonnanceur $LCJ - t$.
Remarque :
En pratique, s'il y a assez de temps pour exécuter une instance Wp après avoir exécuté l'instance Wa correspondante, évidement dans ce cas, le résultat considéré sera celui de la version donnant la meilleure QdS, soit Wp.

5.8 Simulations et évaluation de performances

L'environnement de simulation est le même que celui du chapitre précédent. L'ensemble de tâches généré se compose de 22 tâches avec des temps d'exécution pas nécessairement uniformes et avec des paramètres de contraintes de BGW fixes pour toutes les tâches, soient $n_i = 7$ et $l_i = 4$.

Algorithm 9 Algorithme d'ordonnancement $LCJ - t$

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWaList, Liste des instances Wa prêtes acceptées, ordonnancées entre elles selon EDF ;
WpList, Liste des instances Wp prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    exécuter AcceptedGpList.Head et supprimer le, après exécution ;
    supprimer l'instance Ga correspondante de l'instance Gp, de la liste BGaList ;
else
    /* Il n'y a pas d'instance Gp acceptée */
    if AcceptedWaList.Head != NULL then
        /* Présence des instances Wa acceptées pour être exécutée */
        exécuter AcceptedWaList.Head et le supprimer après exécution ;
    else
        /* Il n'y a pas d'instance Wa acceptée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et le supprimer après exécution ;
        else
            /* Pas d'instance B ou Ga prête pour s'exécuter */
            if WpList.Head != NULL then
                /* Présence d'instances Wp prêtes pour s'exécuter */
                exécuter WpList.Head et le supprimer après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 5.6 – Pseudo-code de $LCJ - t$

5.8.1 Analyse des systèmes légèrement contraints

Nous évaluons les performances en $QdS1$, $QdS2$, taux de préemption (PR) et gaspillage de temps processeur (WR) pour les ordonnanceurs $LCP - t$, $LCJ - t$, LCP et LCJ pour des systèmes temps réel légèrement chargés.
La figure 5.7 présente la $QdS1$. Celle-ci est toujours supérieure de 25% pour tous les algorithmes, même dans des conditions de surcharge. À rappeler que cette valeur correspond à la valeur minimum acceptée par le modèle BGW via sa contrainte de l_i.
Les stratégies d'ordonnancement $LCJ-t$ et LCJ donnent une $QdS1$, presque toujours égale à 100%. Ceci provient de la légèreté des charges secondaires et aux priorités élevées attribuées à ces versions dans ces stratégies.
Par comparaison aux autres stratégies d'ordonnancement, nous remarquons clairement que $LCP-t$ donne des résultats meilleurs en termes de $QdS1$ que LCP, l'EDF classique et l'ordonnanceur $GbWp$ basé sur la technique de première chance.
Et LCP est toujours meilleur que EDF et $GbWp$.

Sous $LCP - t$, $QdS1$ reste supérieur à 90% jusqu'à $U_p = 1.4$. Et, même dans des conditions de très

grande surcharge, $QdS1$ est très élevé, soit 74.35% pour $U_p = 2.2$ alors que pour LCP, $QdS1$ descend jusqu'à 50.88%
Sous LCP, $QdS1$ descend à 93.22% pour $U_p = 1.1$ et jusqu'à 74.63% pour $U_p = 1.4$. Nous avons une différence de 15.95% entre $LCP-t$ et LCP, une différence de 18.35% entre $LCP-t$ et l'EDF classique et une différence de 21.64% avec l'algorithme $GbWp$.
Pour $Up = 2.2$, nous avons une différence de 23.47% en faveur de $LCP-t$ par rapport à LCP, une différence de 29.93% par rapport à EDF et une différence de 39.52% avec $GbWp$.
Nous en déduisons l'intérêt évident du test d'acceptation en ligne dans le but d'augmenter la Qualité de Service du système lorsque celle-ci correspond au ratio d'échéances respectées (par primaire ou secondaire).

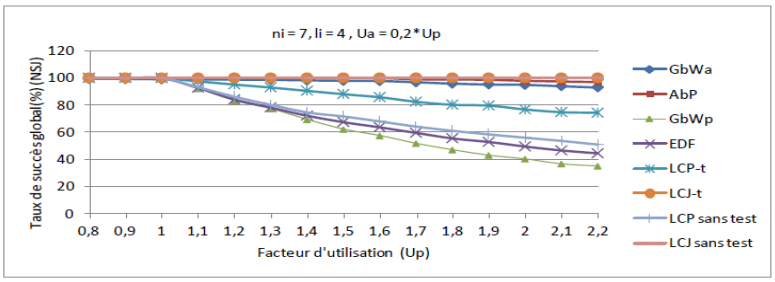


FIGURE 5.7 – $QdS1$ pour des systèmes légers.

La figure 5.8 présente la $(QdS2)$. Nous remarquons ainsi que $QdS2$ est toujours supérieure, à la valeur de 14.28% pour tous les ordonnanceurs, même dans des conditions de surcharge. À rappeler que cette valeur correspond à la valeur minimum acceptée par le modèle BGW via sa contrainte de n_i.
La stratégie d'ordonnancement $LCP-t$ a de loin les meilleures performances en termes de $QdS2$ par rapport à toutes les autres stratégies que celles-ci appliquent la technique de la dernière chance et par voie de conséquence celle de la première chance.

Comparons LCP et LCJ. La stratégie d'ordonnancement LCP donne des résultats meilleurs par rapport aux stratégies $LCJ-t$ et LCJ et par rapport à toutes les stratégies basées sur la première chance.
En effet, pour $Up = 1.1$, nous avons une $QdS2$ de 96.39% pour $LCP-t$, de 92.47% pour LCP, de 92.75% pour EDF, de 91.18% pour $GbWp$, de 65.53% pour $LCJ-t$ et de 65.19% pour LCJ.
Pour $Up = 1.5$, nous avons une $QdS2$ de 85.24% pour $LCP-t$, 67.47% pour EDF, 66.31% pour LCP, 66.61% pour $GbWp$, 37.98% pour $LCJ-t$ et 37.76% pour LCJ.
Ainsi, pour $Up = 2.2$, nous avons une $QdS2$ de 66.29% pour $LCP-t$, 44.42% pour EDF, 35.47% pour LCP sans test, 34.49% pour $GbWp$, 20.15% pour $LCJ-t$ et 20.56% pour LCJ sans test.
En regroupant les observations concernant $QdS1$ et $QdS2$ des différentes stratégies d'ordonnancement étudiées, nous pouvons constater que l'ordonnanceur $LCP-t$ présente la meilleure performance.

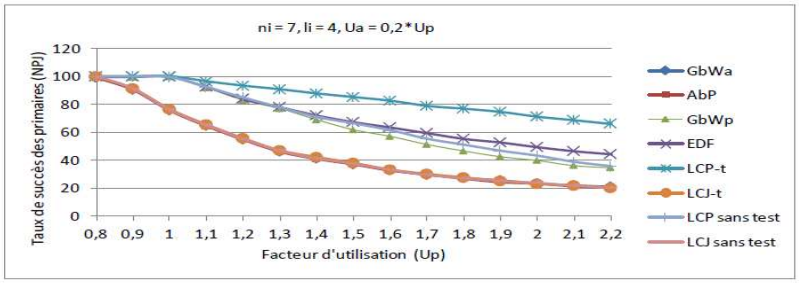


FIGURE 5.8 – $QdS2$ pour des systèmes légèrement chargés.

Les figures 5.9 présentent le taux de préemption. Il s'agit ici d'évaluer le nombre de changements de contexte et par conséquent les surcoûts qu'ils engendrent.

Pour $LCP - t$, le taux de préemption PR est assez élevé pour des charges petites, mais il commence à diminuer dès que la charge devient lourde. C'est logique vu que le système commence à consacrer beaucoup de temps aux versions primaires.
Pour la stratégie LCP, le PR est meilleur comparé à $LCP - t$: sous LCP, le système continue à exécuter des tâches tant qu'il n' y a aucune instance B ou Ga qui demande l'exécution, sans être obligée d'engendrer une préemption.
Dans les conditions de surcharge, la stratégie $LCP - t$ commence à donner des performances en termes de PR, meilleurs par rapport à la stratégie LCJ à partir de $U_p = 1.6$ et par rapport à la stratégie $LCJ - t$ à partir de $U_p = 1.8$. Lorsque la charge devient très lourde, c'est à dire vers $U_p = 2.2$, nous remarquons que les performances des deux stratégies $LCP - t$ et LCP en termes de taux de préemption tendent à être identiques.

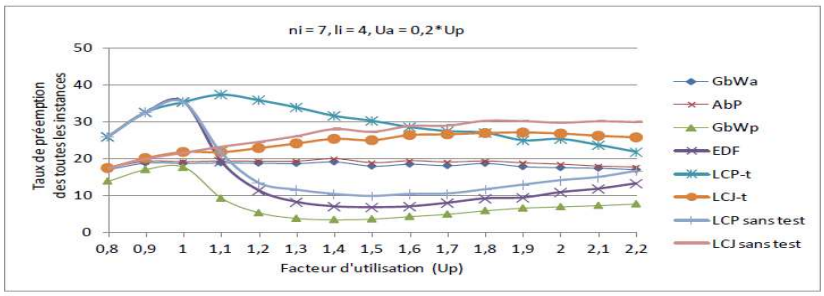


FIGURE 5.9 – Taux de préemption PR pour systèmes légèrement chargés

La figure 5.10 concerne le gaspillage de temps processeur, WR.
Nous remarquons clairement que sous la stratégie $LCP-t$, le système gaspille peu la ressource processeur. Le très petit temps gaspillé vient que le système exécute toutes les versions primaires acceptées puis essaye d'exécuter certaines versions secondaires non testées.
La stratégie LCP gaspille moins la ressource processeur que toutes les autres stratégies, à l'exception de $LCP - t$. Les stratégies $LCJ - t$ et LCJ donnent un WR très élevé : Avec ces stratégies, les versions primaires qui sont plus lourdes, sont exécutées en dernier et ne font pas l'objet d'un test d'acceptation avant

exécution.

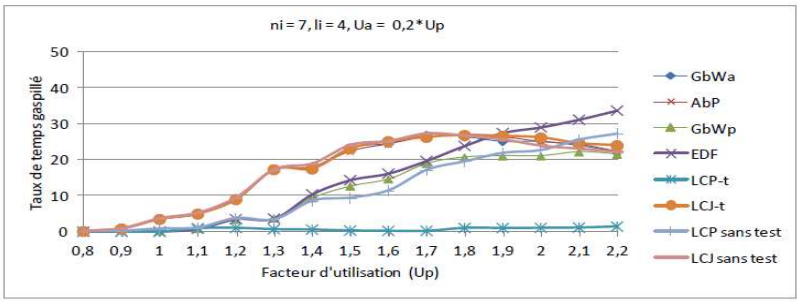


FIGURE 5.10 – WR pour systèmes légèrement chargés.

La figure 5.11 présente le temps d'oisiveté du processeur.
Nous remarquons que pour presque toutes les stratégies étudiées, le temps non occupé du système est quasiment nul dès que la charge des primaires devient supérieure à 1.0.

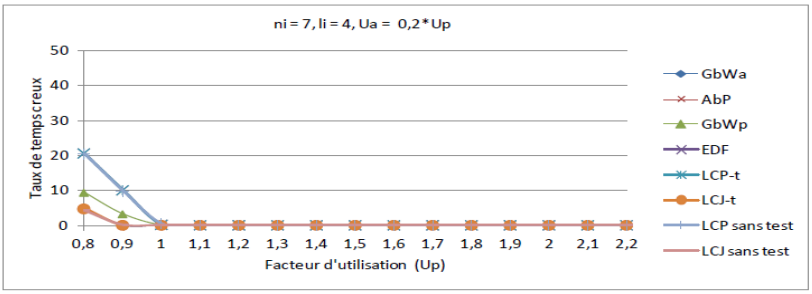


FIGURE 5.11 – Temps d'inactivité de processeur pour systèmes légèrement chargés.

5.8.2 Analyse des systèmes lourdement contraints

La figure 5.12 présente la $QdS1$ qui est toujours supérieure à la valeur de 25% pour tous les algorithmes, même pour des conditions de surcharge.
Les stratégies $LCJ - t$ et LCJ ne donnent plus des pourcentages en $QdS1$ de 100% et c'est à cause que les charges secondaires deviennent importantes ($U_a = 0.6 * U_p$).
La stratégie $LCJ - t$ donne toujours la meilleure performance par rapport à toutes autres stratégies.
À partir de $U_p = 1.7$, $LCP - t$ commence à donner des résultats meilleurs que toutes les stratégies sauf $LCJ - t$.
La stratégie LCJ et les deux stratégies AbP et $GbWa$ restent meilleurs jusqu'à $U_p = 1.5$ où leur performance commence à diminuer considérablement. Nous constatons ainsi que LCP se comporte presque identiquement à EDF en termes de $QdS1$, pour des systèmes à versions secondaires lourdes. LCP reste toujours meilleur que $GbWp$ basé sur la première chance.
La figure 5.13 présente $QdS2$. $QdS2$ est toujours supérieure à 14.28%. $LCP - t$ reste toujours la meilleure avec un taux supérieur à 60% même dans des conditions de surcharge. Ce qui représente un pourcentage très élevé des versions primaires réussies.

La stratégie LCP donne des performances meilleures par rapport à toutes les stratégies de la première chance.
$LCJ-t$ et LCJ donnent les résultats les moins bons : l'attribution des priorités les moins élevées vont aux versions primaires dans ces stratégies.

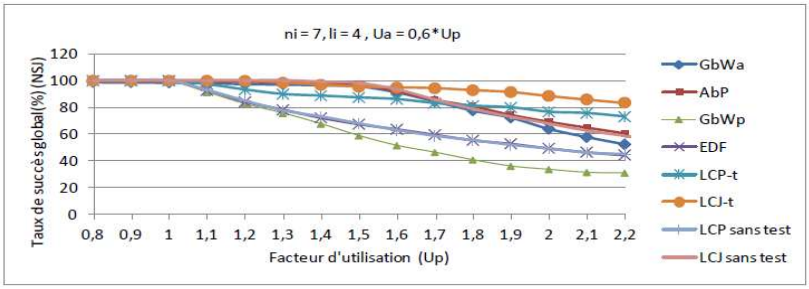


FIGURE 5.12 – $QdS1$ pour systèmes lourdement chargés.

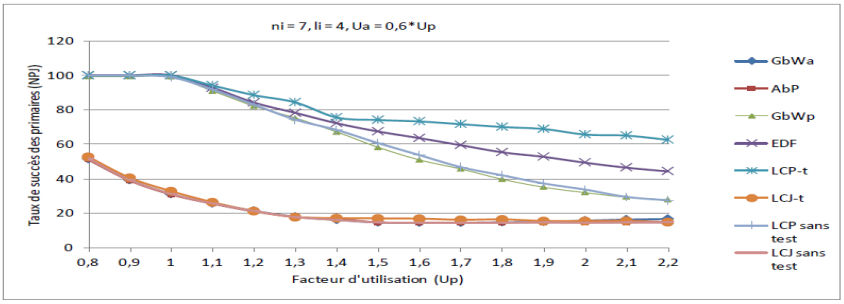


FIGURE 5.13 – $QdS2$ pour systèmes lourdement chargés.

La figure 5.14 donne le taux de préemption, PR. Nous constatons qu'avec des versions secondaires lourdes, la stratégie $LCJ-t$ devient la plus mauvaise en termes de PR.
La stratégie d'ordonnancement $LCP-t$ devient bonne, comparablement avec $LCJ-t$.
LCP reste aussi meilleure entre $U_p = 1.1$ et $U_p = 1.9$ par rapport à toutes les autres stratégies à l'exception de $GbWp$ et de EDF.
Lorsque la charge tend à être très lourde, les comportements des stratégies LCP et EDF se rapprochent de celui de $LCP-t$.
Ainsi, la stratégie $GbWp$ présente les meilleures performances dans toutes les conditions de charge engendrant donc le moins de surcoûts de préemptions.

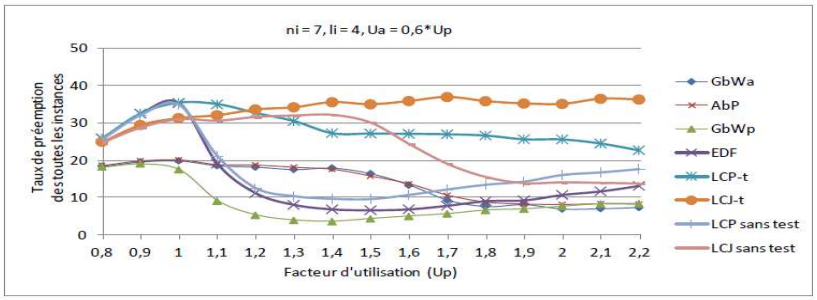


FIGURE 5.14 – PR pour des systèmes lourds.

La figure 5.15 présente le gaspillage de la ressource processeur : dans le cas des systèmes globalement lourds ou avec des versions secondaires lourdes, le plus remarquable est la petite augmentation en termes de WR pour la stratégie $LCP-t$. Ceci s'explique vu qu'on ne teste pas l'acceptation des instances secondaires Wa provoquant ainsi le gaspillage.
EDF donne les performances les moins bonnes par rapport à toutes les autres stratégies et ceci à partir de $U_p = 1.4$. LCP donne les résultats les moins bons par rapport à toute autre stratégie à l'exception de EDF.
La figure 5.16 présente le temps d'oisiveté où nous faisons les même constat que pour les systèmes légers.

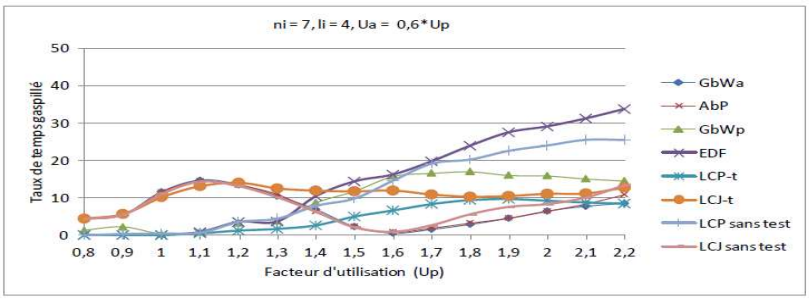


FIGURE 5.15 – WR pour des systèmes lourdement chargés.

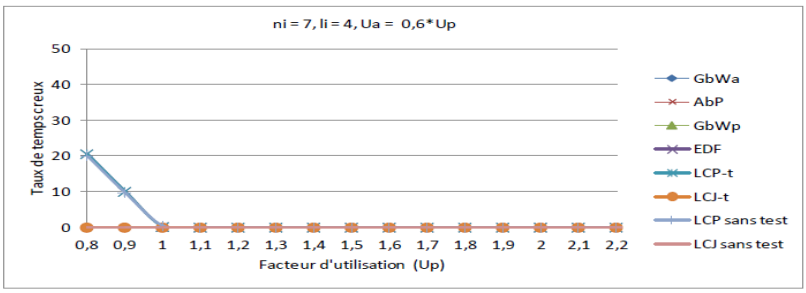


FIGURE 5.16 – Oisiveté des systèmes lourdement chargés.

5.8.3 Synthèse de l'étude

Les tableaux suivants résument les performances relatives des différentes stratégies d'ordonnancement proposées dans ce chapitre. * signifie une performance acceptable, ** une bonne performance et *** une très bonne performance.

Algorithme	$QdS1$	$QdS2$	Préemptions	Gaspillage
$GbWa$	**	**	**	*
AbP	***	*	**	*
$GbWp$	*	***	***	**
LCP	*	**	***	**
LCJ	*	**	*	*
$LCP-t$	*	**	**	***
$LCJ-t$	*	**	*	*
EDF	*	**	**	*

TABLE 5.1 – Synthèse des performances pour systèmes légèrement chargés.

Algorithme	$QdS1$	$QdS2$	Préemptions	Gaspillage
$GbWa$	**	**	**	***
AbP	***	*	**	***
$GbWp$	*	***	***	**
LCP	*	**	**	*
LCJ	*	**	**	**
$LCP-t$	*	**	**	***
$LCJ-t$	*	**	*	**
EDF	*	**	**	*

TABLE 5.2 – Synthèse des performances pour systèmes lourdement chargés.

Ces tableaux récapitulatifs montrent que les différents ordonnanceurs applicables au modèle BGW diffèrent quant à leurs performances. Choisir un ordonnanceur doit d'abord s'effectuer par rapport aux spécificités de l'application. Le système est-il lourdement ou légèrement chargé ? Cherche t-on à optimiser le taux d'instances réussies ou le taux de primaires réussis ? Une préemption engendre un surcoût temporel dans le système d'exploitation. Lorsque celui-ci présente des overheads extrêmement faibles, le taux de préemption a peu d'impact sur la performance. Dans le cas contraire, il s'agit d'une mesure à prendre en compte.

5.9 Conclusion

Dans ce chapitre, nous avons proposé quatre nouvelles stratégies d'ordonnancement pour le modèle BGW par l'application de la technique de la dernière chance. Plus sophistiquées que les stratégies présentées au chapitre précèdent, nous les avons évaluées de façon comparative. Cette étude a été menée sur deux profils d'applications : systèmes légèrement chargés et systèmes lourdement chargés.

Si l'on fait abstraction des overheads, $LCP - t$ permet d'obtenir le taux d'échéances respectées le plus élevé, ce qui est généralement la mesure de Qualité de service utilisée. Si l'on veut plutôt privilégier la qualité des résultats fournis outre leur nombre, nous avons mis en évidence la supériorité de la stratégie $LCJ - t$.

L'étude que nous avons menée dans ce chapitre porte sur une analyse du comportement du système au niveau de l'ensemble les tâches qui le composent. Nous proposons de pousser plus loin cette analyse en la portant au niveau individuel de chaque tâche. La question que nous posons s'exprime dans les termes suivant : La Qualité du Service est elle équitablement répartie sur l'ensemble des tâches ? N'est-il pas possible de maintenir une très bonne Qualité de Service global avec une meilleure équité de service ?

6

Ordonnancements équitables pour BGW

6.1 Introduction

Dans ce chapitre, nous proposons des stratégies d'ordonnancement qui non seulement, visent à optimiser la Qualité de Service mesurée au niveau du système mais font en sorte que les tâches d'une part soient traitées de façon équitable et que ces tâches fournissent individuellement un service qualifié d'uniforme. Nous commencerons par introduire ces nouvelles notions d'équité et d'uniformité de service. Ensuite, nous décrirons deux stratégies visant à améliorer l'équité de service et deux autres visant à améliorer l'uniformité de service. Chacune d'elles peut alors être combinée à un des ordonnanceurs présentés dans les chapitres précédents en remplacement de la règle classique d'ordonnancement, EDF. Une étude de simulation rapportera les gains ainsi obtenus en matière d'équité et d'uniformité de service.

6.2 Équité de service et uniformité de service

Même si la QdS tel que vue au chapitre précédent, reflète la performance d'un système temps réel, cette mesure ne nous renseigne pas sur la QdS individuelle de chaque tâche. En effet, une QdS élevée du système peut cacher des performances individuelles médiocres de quelques tâches. D'où la nécessité d'approfondir l'analyse au niveau de la tâche en mesurant sa propre QdS.

Nous illustrons cela par des résultats obtenus sous la stratégie $LCP - t$:
Pour une charge des primaires $U_p = 1.7$, nous avons les $QdS1$ suivantes pour les 10 tâches : 100%, 72.22%, 88%, 61.29%, 75%, 72.73%, 69.81%, 69.23%, 81.25%, 67.74%. Nous remarquons une inéquité de service puisque la différence maximale de $QdS1$ s'élève à 38.71%.
Pour une charge des primaires $U_p = 1.8$, nous avons les $QdS1$ suivantes : 92.31%, 61.11%, 79.17%, 61.29%, 75%, 63.64%, 69.81%, 70.15%, 77.5%, 69.89%. Nous remarquons encore une inéquité de service puisque la différence maximale de $QdS1$ s'élève à 31.2%.

Nous montrons dans ce chapitre comment pallier à ce problème. Donnons d'abord les définitions suivantes :

Définition 1 : On appelle $QdS1$ *individuelle* d'une tâche, notée $QdS1$, le ratio entre le nombre total d'échéances respectées par cette tâche et le nombre d'instances générées par cette tâche sur un intervalle de référence donné.

Définition 2 : On appelle $QdS2$ *individuelle* d'une tâche, notée $QdS2$ le ratio entre le nombre total d'échéances respectées par les primaires de cette tâche et le nombre d'instances générées par cette tâche sur un intervalle de référence.

Définition 3 : Un algorithme d'ordonnancement X est dit plus *équitable* qu'un algorithme d'ordonnancement Y *au sens de la* $QdS1$, si la plus grande différence entre les $QdS1$ individuelles des tâches avec X est inférieure à la plus grande différence entre les $QdS1$ individuelles des tâches avec Y.

Définition 4 : Un algorithme d'ordonnancement X est dit plus *équitable* qu'un algorithme d'ordonnancement Y *au sens de la* $QdS2$, si la plus grande différence entre les $QdS2$ individuelles des tâches avec X est inférieure à la plus grande différence entre les $QdS2$ individuelles des tâches avec Y.

Nous proposons ci-dessous d'autres critères de qualité de service comme la distance maximale séparant deux exécutions réussies d'une tâche et la distance maximale séparant deux exécutions réussies des versions primaires d'une tâche.

Définition 5 : On appelle *distance de réussite* d'une tâche, notée $DMax1$, le plus grand nombre d'échéances successives non respectées par cette tâche sur un intervalle de référence.

Définition 6 : On appelle *distance de réussite primaire* d'une tâche, notée $DMax2$, le plus grand nombre d'échéances successives non respectées par la version primaire de cette tâche sur un intervalle de référence.

Ces deux nouvelles mesures nous amènent donc à donner les deux propriétés suivantes d'un algorithme d'ordonnancement.

Définition 7 : Un algorithme d'ordonnancement X est dit plus *uniforme* qu'un algorithme d'ordonnancement Y *au sens de la* $QdS1$, si la plus grande différence entre les $DMax1$ individuelles des tâches avec X est inférieure à la plus grande différence entre les $DMax1$ individuelles des tâches avec Y.

Définition 8 : Un algorithme d'ordonnancement X est dit plus *uniforme* qu'un algorithme d'ordonnancement Y *au sens de la* $QdS2$, si la plus grande différence entre les $DMax2$ individuelles des tâches avec X est inférieure à la plus grande différence entre les $DMax2$ individuelles des tâches avec Y.

Dans les paragraphes suivants, nous proposons quatre algorithmes appelés MSJ, MSP, MDJ et MDP qui permettent l'amélioration de l'équité et de l'uniformité de service au sens de la $QdS1$ et de la $QdS2$.

6.3 Stratégie d'ordonnancement MSJ

MSJ(Minimum Success of Jobs) consiste à utiliser une autre priorité que l'urgence pour attribuer le processeur à une tâche. A tout instant, la tâche qui est sélectionnée au sein d'une liste pour s'exécuter est celle qui a la plus petite $QdS1$ individuelle calculée depuis l'initialisation du système. Ce nouveau critère de sélection s'applique aux listes des instances des Gp, Wp ou des Wa. Dans le cas de conflit des priorités

selon MSJ (si deux ou plus des tâches ont la même priorité selon MSJ, le problème est résolu en faveur de la tâche qui a l'échéance la plus proche. Pour les listes des instances B ou Ga, la politique d'attribution des priorités reste sans changement.

Compte tenu des résultats prouvés dans le chapitre précédent, nous utiliserons la technique de la dernière chance avec test combinée à cette nouvelle règle d'affectation des priorités.

La figure 6.1 représente le pseudo code de l'ordonnanceur MSJ.

Algorithm 10 Algorithme d'ordonnancement MSJ

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWpList, Liste des instances Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    chercher l'instance Gp_i de la tâche qui a la moindre QdS1 individuelle calculée à partir de l'initialisation
    du système, de la liste AcceptedGpList ;
    exécuter Gp_i et le supprimer après exécution ;
    supprimer l'instance Ga_i correspondante de l'instance Gp_i de la liste BGaList ;
else
    /* Il n'y a pas d'instances Gp acceptées pour être exécutées */
    if AcceptedWpList.Head != NULL then
        /* Présence des instances Wp acceptées pour être exécutée */
        chercher l'instance Wp_i de la tâche qui a la plus petite QdS1 individuelle calculée à partir de l'initialisation
        du système, de la liste AcceptedWpList ;
        exécuter Wp_i et le supprimer, après exécution ;
        supprimer l'instance Wa_i correspondante de l'instance Wp_i, de la liste WaList ;
    else
        /* Il n'y a pas d'instances Wp acceptées pour être exécutée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et supprimer le, après exécution ;
        else
            /* Pas d'instances B ou Ga prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                chercher l'instance Wa_i de la tâche qui a la moindre QdS1 individuelle calculée à partir de l'initialisation
                du système, de la liste WaList ;
                exécuter Wa_i et supprimer le, après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 6.1 – Pseudo-code de la stratégie MSJ combinée à $LCP - t$

6.4 Stratégie d'ordonnancement MSP

MSP (Minimum Success of Primary versions) consiste à sélectionner la tâche qui a la plus petite $QdS2$ individuelle calculée à partir de l'initialisation du système. Cette politique s'applique aux listes Gp ou des Wp. Pour les listes d'instances B et Ga, la politique d'attribution des priorités reste sans changement. Là aussi, nous combinerons cette règle d'affectation de la priorité à une stratégie d'ordonnancement basée sur la technique de la dernière chance, LCP.

La figure 6.2 représente le pseudo code de l'ordonnanceur MSP.

Algorithm 11 Algorithme d'ordonnancement MSP

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWpList, Liste des instances Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    chercher l'instance Gp_i de la tâche qui a la moindre (QdS2) individuelle calculée à partir de l'initialisation
    du système, de la liste AcceptedGpList ;
    exécuter Gp_i et supprimer le, après exécution ;
    supprimer l'instance Ga_i correspondante de l'instance Gp_i, de la liste BGaList ;
else
    /* Il n'y a pas des instances Gp acceptées pour être exécutée */
    if AcceptedWpList.Head != NULL then
        /* Présence des instances Wp acceptées pour être exécutée */
        chercher l'instance Wp_i de la tâche qui a la moindre (QdS2) individuelle calculée à partir de l'initialisation
        du système, de la liste AcceptedWpList ;
        exécuter Wp_i et le supprimer après exécution ;
        supprimer l'instance Wa_i correspondante de l'instance Wp_i, de la liste WaList ;
    else
        /* Il n'y a pas d'instances Wp acceptées pour être exécutée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et le supprimer après exécution ;
        else
            /* Pas d'instances B ou Ga prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                exécuter WaList.Head et le supprimer après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 6.2 – Pseudo-code de la stratégie MSP combinée à $LCP - t$

6.5 Ordonnanceur MDJ

MDJ (Maximal Distance between successful Jobs) consiste à sélectionner la tâche qui a la plus grande Distance de Réussite individuelle ($DMax1$) à partir de l'initialisation du système. Dans le cas de conflit des priorités selon MDJ (si deux ou plus des tâches ont la même priorité selon MDJ), le problème est résolu en faveur de la tâche qui a l'échéance la plus proche. Autrement dit, on privilégie l'exécution de l'instance dont la dernière exécution réussie est la plus grande. Cette politique s'applique aux listes Gp, Wp et Wa. Pour les listes d'instances B et Ga, la politique d'attribution des priorités reste sans changement (EDF). Là aussi, nous combinerons cette règle d'affectation de la priorité à une stratégie d'ordonnancement basée sur la technique de la dernière chance telle que LCP ou $LCP - t$.

La figure 6.3 représente le pseudo code de l'ordonnancemeur MDJ.

6.6 Ordonnanceur MDP

MDP (Maximal Distance between successful Primaries) consiste à sélectionner la tâche qui a la plus grande Distance de Réussite primaire individuelle ($DMax2$) calculée à partir de l'initialisation du système. Dans le cas de conflit des priorités selon MDP (si deux ou plus des tâches ont la même priorité selon MDJ), le problème est résolu en faveur de la tâche qui a l'échéance la plus proche. Autrement dit, on privilégie l'exécution de l'instance dont la dernière exécution réussie par un primaire est la plus grande. Cette politique s'applique aux listes Gp, Wp et Wa. Pour les listes d'instances B et Ga, la politique d'attribution des priorités reste sans changement (EDF). Là aussi, nous combinerons cette règle d'affectation de la priorité à un ordonnanceur tel que LCP.

La figure 6.4 représente le pseudo code de l'ordonnanceur MDP.

6.7 Simulations et évaluation des performances

L'environnement des simulations dans ce chapitre est un peu différent de ceux des chapitres précédents, en particulier en termes de nombre de tâches générées. Nous nous limitons dans ce chapitre au nombre de 10 tâches, afin de pouvoir afficher les résultats des performances au niveau individuel.

6.7.1 Équité de service au sens de la $QdS1$

La figure 6.5 représente les mesures de la $QdS1$ individuelle pour un ensemble de 10 tâches, sous la stratégie $LCP - t$ utilisant EDF et donc sans algorithme de gestion de l'équité de service.
Nous remarquons d'abord que la $QdS1$ observée est de loin supérieure à la valeur de 25% pour chaque tâche, dans toutes les conditions de charge. Rappelons qu'il s'agit de la valeur minimale acceptée par le modèle BGW via sa contrainte de l_i. Nous constatons ainsi qu'il existe des écarts importants entre les $QdS1$ individuelles des tâches d'une même application.

Par exemple, pour $U_p = 1.4$, nous avons les $QdS1$ suivantes pour 10 tâches : 100%, 100%, 80%, 90.91%, 65%, 78.72%, 83.02%, 86.76%, 79.27%, 88.66%. D'où un écart de 35% entre la plus petite et la plus grande $QdS1$.
Pour $U_p = 1.5$, nous avons une $QdS1$ de 100% pour la tâche τ_1, alors que pour la tâche τ_3 nous avons une $QdS1$ de 64%, ce qui représente un écart maximal de 36%.
Pour $U_p = 2$, nous avons les $QdS1$ suivantes 10 tâches : 83.33%, 78.95%, 70.83%, 84.38%, 57.89%,

Algorithm 12 Algorithme d'ordonnancement MDJ

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWpList, Liste des instances Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    chercher dans la liste AcceptedGpList, l'instance Gp_i de la tâche qui a la plus grande DMax1 individuelle,
    calculée à partir de l'initialisation du système ;
    exécuter Gp_i et supprimer le, après exécution ;
    supprimer l'instance Ga_i correspondante de l'instance Gp_i, de la liste BGaList ;
else
    /* Il n'y a pas des instances Gp acceptées pour être exécutée */
    if AcceptedWpList.Head != NULL then
        /* Présence des instances Wp acceptées pour être exécutée */
        chercher dans la liste AcceptedWaList, l'instance Wp_i de la tâche qui a la plus grande DMax1
        individuelle, calculée à partir de l'initialisation du système ;
        exécuter Wp_i et le supprimer après exécution ;
        supprimer l'instance Wa_i correspondante de l'instance Wp_i, de la liste WaList ;
    else
        /* Il n'y a pas d'instances Wp acceptées pour être exécutée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et le supprimer après exécution ;
        else
            /* Pas d'instances B ou Ga prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                chercher dans la liste AcceptedWaList, l'instance Wa_i de la tâche qui a la plus grande DMax1
                individuelle, calculée à partir de l'initialisation du système ;
                exécuter Wa_i et le supprimer après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 6.3 – Pseudo-code de la stratégie MDJ combinée à $LCP - t$

72.34%, 73.08%, 78.79%, 67.50%, 67.02%. D'où un écart de 26.5% entre la plus petite et la plus grande $QdS1$.

Ces mesures soulignent la nécessité de faire appel à des ordonnanceurs avec un meilleur comportement au regard de l'équité de service.

La figure 6.6 représente la $QdS1$ individuelle pour un ensemble de dix tâches, sous la stratégie $LCP-t$ qui substitue EDF par une règle plus adaptée d'assignation de la priorité. Nous remarquons ainsi que la $QdS1$ individuelle est de loin supérieure à la valeur de 25% et ce pour chaque tâche du système, dans toutes les conditions de charge, même dans celles de surcharge.

Algorithm 13 Algorithme d'ordonnancement MDP

Entrées : $\Gamma = \{\tau_1, ..., \tau_n\}$
BGaList, Liste des instances B et Ga prêtes ordonnancées entre elles selon EDF ;
AcceptedGpList, Liste des instances Gp prêtes acceptées, ordonnancées entre elles selon EDF ;
AcceptedWpList, Liste des instances Wp prêtes acceptées, ordonnancées entre elles selon EDF ;
WaList, Liste des instances Wa prêtes ordonnancées entre elles selon EDF ;
t, L'instant courant ;

```
begin
if AcceptedGpList.Head != NULL then
    /* Présence des instances Gp acceptées pour être exécutée */
    chercher dans la liste AcceptedGpList, l'instance Gp_i de la tâche qui a la plus grande DMax2
    individuelle, calculée à partir de l'initialisation du système ;
    exécuter Gp_i et le supprimer après exécution ;
    supprimer l'instance Ga_i correspondante de l'instance Gp_i, de la liste BGaList ;
else
    /* Il n'y a pas des instances Gp acceptées pour être exécutée */
    if AcceptedWpList.Head != NULL then
        /* Présence des instances Wp acceptées pour être exécutée */
        chercher dans la liste AcceptedWpList, l'instance Wp_i de la tâche qui a la plus grande DMax2
        individuelle, calculée à partir de l'initialisation du système ;
        exécuter Wp_i et le supprimer après exécution ;
        supprimer l'instance Wa_i correspondante de l'instance Wp_i, de la liste WaList ;
    else
        /* Il n'y a pas d'instances Wp acceptées pour être exécutée */
        if BGaList.Head != NULL then
            /* Présence des instances B ou Ga prêtes pour s'exécuter */
            exécuter BGaList.Head et le supprimer après exécution ;
        else
            /* Pas d'instances B ou Ga prêtes pour s'exécuter */
            if WaList.Head != NULL then
                /* Présence des instances Wa prêtes pour s'exécuter */
                exécuter WaList.Head et le supprimer après exécution ;
            end if
        end if
    end if
end if
end
```

FIGURE 6.4 – Pseudo-code de la stratégie MDP combinée à $LCP - t$

Nous faisons les observations suivantes :
Pour $U_p = 1.4$, les $QdS1$ individuelles sont : 84.62%, 78.95%, 62.96%, 78.79%, 78.05%, 78.72%, 77.78%, 78.26%, 78.31%, 78.35%. Il existe un écart maximal en $QdS1$ qui est passé de 35% (sans politique de gestion de l'équité de service) à 21.66%.
Pour $U_p = 1.5$, nous remarquons que l'écart maximal en $QdS1$ est passé à 7.22% au lieu de 36%. Pour $U_p = 2$, les $QdS1$ individuelles sont : 76.92%, 72.22%, 72%, 71.88%, 71.79%, 71.11%, 71.70%, 71.64%, 72.15%, 71.74%. Il existe un écart maximal en $QdS1$ qui est passé de 26.49% à 5.81%.
Nous remarquons aussi une nette augmentation de la QdS globale du système, $QdS1$, avec MSJ en comparaison de EDF.

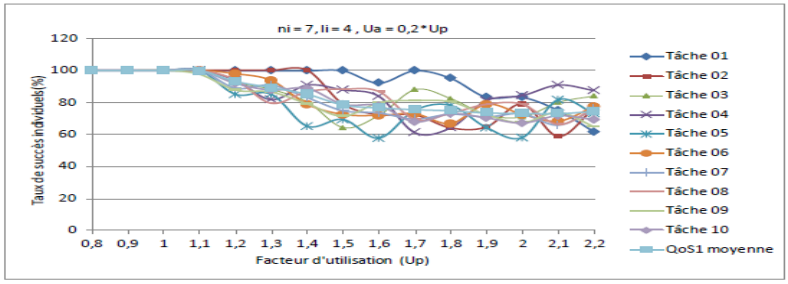


FIGURE 6.5 – $QdS1$ individuelle sans équité de service.

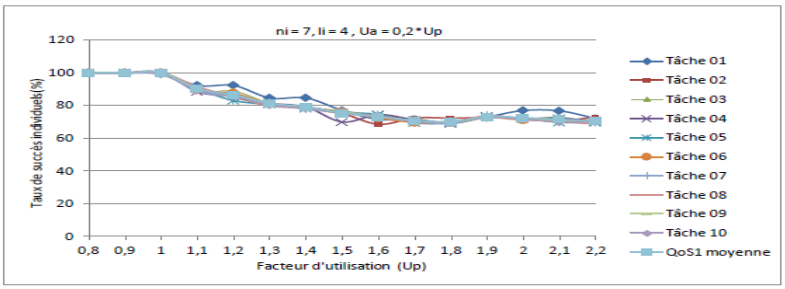


FIGURE 6.6 – $QdS1$ individuelle avec MSJ.

6.7.2 Équité de service au sens de la $QdS2$

La figure 6.7 représente les mesures de la $QdS2$ individuelle pour un ensemble de dix tâches, sous la stratégie $LCP - t$ en l'absence de gestion de l'équité de service.
Nous remarquons que la $QdS2$ est de loin supérieure à 14.28% pour chaque tâche, dans toutes les conditions de charge. Rappelons que cette valeur correspond à la valeur minimale acceptée par le modèle BGW à travers sa contrainte de n_i.
Nous constatons des écarts importants de $QdS2$ entre les tâches.
Par exemple, pour $U_p = 1.4$, les $QdS2$ individuelles des 10 tâches sont : 85.71%, 68.42%, 56.00%, 69.70%, 47.50%, 72.34%, 73.58%, 76.47%, 68.29%, 78.35% d'où un écart de plus de 38%.
Pour $U_p = 1.5$, nous avons une $QdS2$ de 85.7% pour τ_1, alors que τ_3 présente une $QdS2$ de 44.0%, d'où un écart de plus de 40%.
Pour $U_p = 2.00$, les $QdS2$ individuelles des 10 tâches sont : 33.33%, 36.84%, 33.33%, 53.13%, 31.58%, 48.94%, 48.08%, 54.55%, 45.00%, 46.81% d'où un écart en $QdS2$ de près de 23%.

La figure 6.8 représente les mesures de la $QdS2$ individuelle pour un ensemble de dix tâches, sous la stratégie $LCP - t$ avec gestion de l'équité de service par l'algorithme MSP. La $QdS2$ individuelle est de loin supérieure à 14.28% pour chaque tâche, dans toutes les conditions de charge, même dans celles de surcharge. Nous constatons que l'algorithme d'équité de service en $QdS2$ appliqué à la stratégie $LCP - t$ a pu assurer une bonne équité de service par rapport à ce critère, entre les différentes tâches.

Pour $U_p = 1.4$, les $QdS2$ individuelles des 10 tâches sont : 69.23%, 68.42%, 66.67%, 64.71%, 65.85%, 65.96%, 64.81%, 65.22%, 65.43%, 65.26%. D'où un écart maximal en $QdS2$ qui est passé de 38.21% à 4.52% avec la politique MSP.
Pour $U_p = 1.5$, l'écart maximal en $QdS2$ est passé à 3.74% (avec MSP) au lieu de 41.71% (avec EDF).
Pour $U_p = 2.00$, la plus petite valeur de $QdS2$, 39.47% et la plus grande valeur de $QdS2$, 42.11%, conduisent à un écart maximal de 2.64% au lieu des 22.97% en l'absence d'un mécanisme de gestion d'équité de service.
Finalement, nous remarquons qu'il y a aussi une augmentation notoire de la $QdS2$ globale du système.

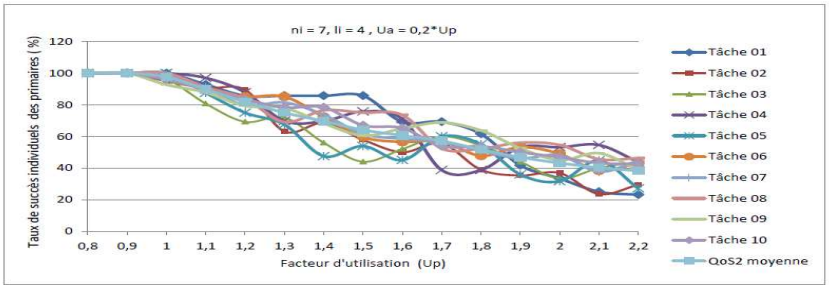


FIGURE 6.7 – $QdS2$ individuelle sans équité de service.

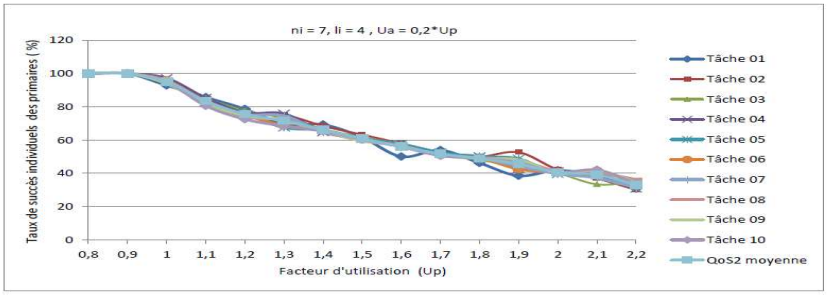


FIGURE 6.8 – $QdS2$ individuelle avec MSP.

6.7.3 Uniformité de service au sens de la $QdS1$

La figure 6.9 montre les mesures de la $DMax1$ individuelle pour un ensemble de dix tâches, sous la stratégie LCP sans application d'aucun mécanisme d'équité de service. Cette mesure nous donne pour chaque tâche la répartition du service effectué sur cette tâche. Plus cette grandeur est petite, plus le service produit par cette tâche (avec son primaire ou son secondaire) est uniformément réparti dans le temps.
Notons que la $DMax1$ ne peut pas dépasser la valeur 3 pour chaque tâche, dans toutes les conditions de charge, y compris celles de surcharge. Il s'agit de la valeur maximale acceptée par le modèle BGW à travers sa contrainte sur l_i avec les paramètres adoptés dans notre étude en simulation qui sont $n_i = 7$ et $l_i = 4$.
Pour $U_p = 1.4$, nous avons une $DMax1$ de 0 pour τ_1, 1 pour les tâches τ_2 et τ_4, 2 pour les cinq tâches τ_3, τ_6, τ_8, τ_9 et τ_{10} et une $DMax1$ de 3 pour les autres tâches. Pour $U_p = 1.6$, nous avons une $DMax1$ de 1 pour les deux tâches τ_1 et τ_8, 2 pour les trois tâches τ_2, τ_3, τ_6 et 3 pour les autres tâches.
Cet exemple met donc en évidence un besoin fort d'uniformité de service non fourni par l'utilisation de la

règle de priorité de EDF qui ne considère que l'urgence des tâches.
La figure 6.10 montre les mesures de la $DMax1$ individuelle pour un ensemble de dix tâches, sous la stratégie $LCP - t$ combiné à MDJ qui permet d'améliorer l'uniformité de service par rapport à EDF.

Notons ici que la $DMax1$ ne dépasse pas 3 pour chaque tâche de par la contrainte l_i du modèle BGW. Nous observons que :
Pour $U_p = 1.4$, $DMax1$ vaut 2 pour toutes les tâches sauf pour τ_5 avec une $DMax1$ de 3. Pour $U_p = 1.6$, toutes les tâches ont une $DMax1$ de 2, à l'exception des deux tâches τ_9 et τ_{10} avec une $DMax1$ de 3. Quelque soit la charge processeur U_p, nous avons deux valeurs différentes de la $DMax1$ avec un écart de 1 entre ces deux valeurs.

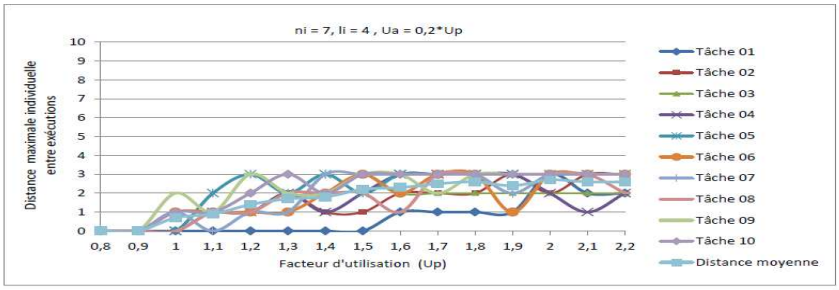


FIGURE 6.9 – Distance de réussite individuelle ($DMax1$) sans uniformité de service.

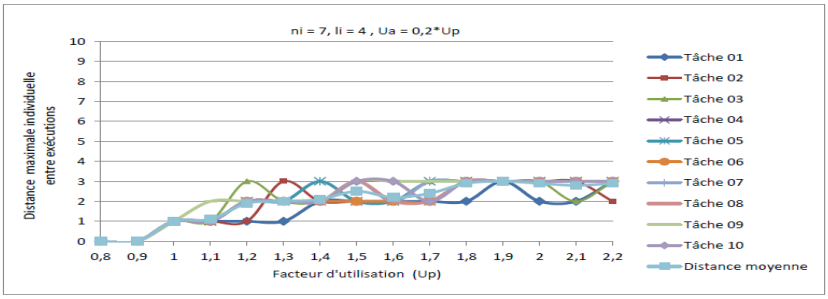


FIGURE 6.10 – Distance de réussite individuelle ($DMax1$) avec MDJ.

6.7.4 Uniformité de service au sens de la $QdS2$

La figure 6.11 montre les mesures de la $DMax2$ individuelle sous la stratégie $LCP - t$ utilisant la règleEDF. Nous remarquons que $DMax1$, la distance de réussite entre deux primaires, ne peut pas dépasser 6 pour chaque tâche, dans toutes les conditions de charge. Rappelons que cette valeur correspond à la valeur maximale acceptée par le modèle BGW à travers sa contrainte de n_i.
Nous constatons que les tâches présentent des $DMax2$ avec des écarts importants, signifiant ainsi un service rendu par les versions primaires très irrégulier d'une tâche à l'autre. Par exemple, pour $U_p = 1.4$, τ_1 a une $DMax2$ de 1 alors que pour τ_5 sa $DMax2$ vaut 6 et pour trois autres tâches, τ_7, τ_8 et τ_{10} nous avons une $DMax2$ de 3. Pour les autres tâches, $DMax2$ vaut 2. On peut dire que τ_1 est la plus uniformément traitée au cours du temps (réussite des primaires à périodicité quasi-constante) alors qu'au contraire τ_5 est

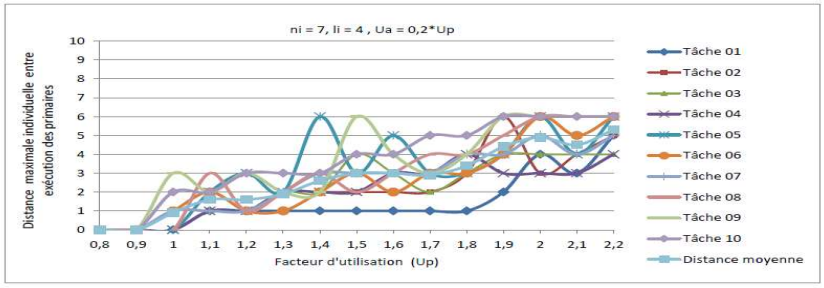


FIGURE 6.11 – Distance de réussite ($DMax2$) sans uniformité de service.

la moins uniformément traitée au cours du temps (réussite des primaires à périodicité très variable).
Pour $U_p = 1.6$, nous avons une $DMax2$ de 1 pour τ_1 alors que pour τ_2 et τ_6 nous avons une $DMax2$ de 2, pour deux autres tâches, τ_9 et τ_{10} nous avons une $DMax2$ de 4, pour la tâche τ_5 la $DMax2$ vaut 5 et pour les autres tâches nous avons des $DMax2$ de 3.
Pour $U_p = 2$, nous avons pour les deux tâches τ_1 et τ_3 une $DMax2$ de 4, pour les deux tâches τ_2 et τ_4 une $DMax2$ de 3 alors que pour τ_7 nous avons une $DMax2$ de 5 et pour les autres tâches, $DMax2$ vaut 6.

En résumé, quelque soit la charge du processeur, il existe une différence importante entre les tâches en matière de régularité d'exécution de leurs primaires. D'où la nécessité la-aussi de substituer le classique EDF par une règle plus adaptée, en particulier MDP.

La figure 6.12 montre les mesures de la $DMax2$ individuelle sous la stratégie $LCP - t$ combinée à un ordonnanceur spécifique, MDP, présenté avant.
Nous constatons que :
Pour $U_p = 1.4$, nous avons une $DMax2$ de 3 pour toutes les tâches sauf pour τ_2 qui a une $DMax2$ de 2. D'où un écart de 1 alors que sans mécanisme d'uniformité de service, cet écart était de 5.
Pour $U_p = 1.6$, nous avons une $DMax2$ de 4 pour trois tâches et pour toutes les autres, $DMax2$ vaut 3. Cet écart de 1 remplace l'écart de 4 que nous avions avec EDF.
Pour $U_p = 2$, nous avons une $DMax2$ de 4 pour quatre tâches et pour toutes les autres, $DMax2$ vaut 5. Cet écart de 1 remplace l'écart de 3 que nous avions avec EDF.
Quelque soit la charge processeur, l'uniformité de traitement des primaires sur le temps est acquise, d'une part au niveau de chaque tâche et d'autre part au niveau du système global.

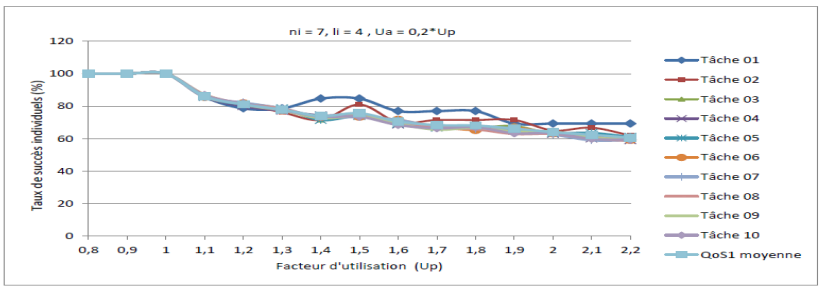


FIGURE 6.14 – $QdS1$ individuelle avec MSJ.

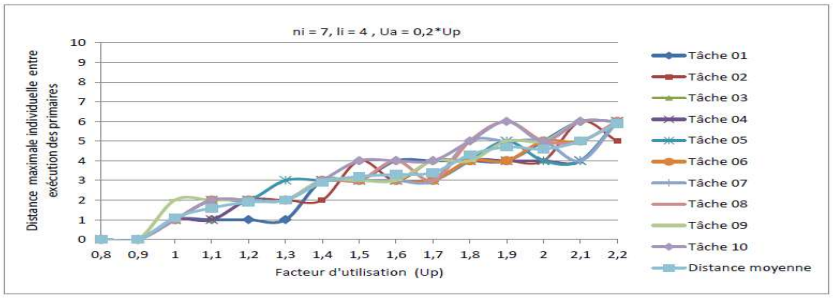


FIGURE 6.12 – Distance de réussite ($DMax2$) avec MDP.

6.7.5 Autres mesures

L'ensemble des résultats rapportés dans la section précédente concernent l'ordonnanceur $LCP - t$. Nous avons reproduit la même expérience mais en considérant l'ordonnanceur LCP et ce, pour savoir si les politiques MSJ, MSP, MDJ et MDP avaient le même impact sur l'équité et l'uniformité de service résultantes. Les figures 6.13 à 6.16 illustrent un gain similaire tant du point de vue de l'équité de service que de l'uniformité de service.

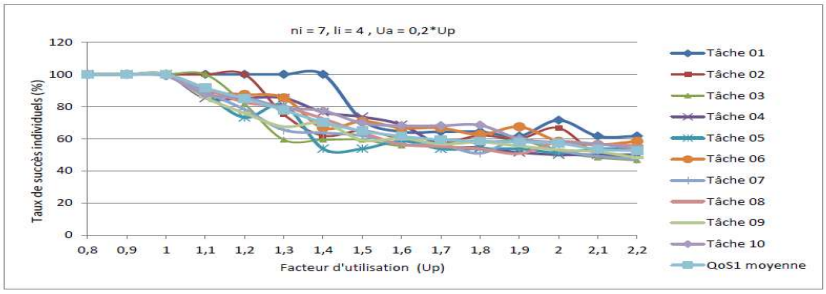


FIGURE 6.13 – $QdS1$ individuelle sans équité de service.

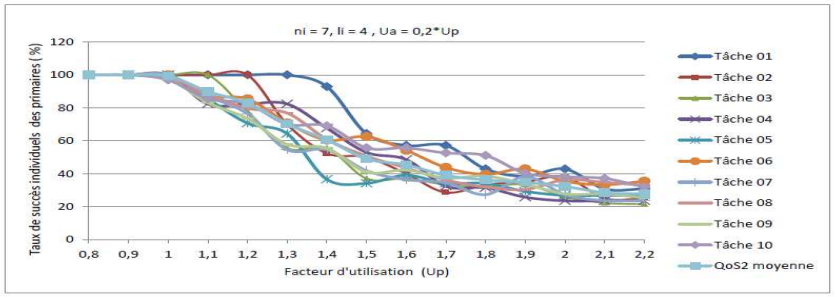


FIGURE 6.15 – $QdS2$ individuelle sans équité de service.

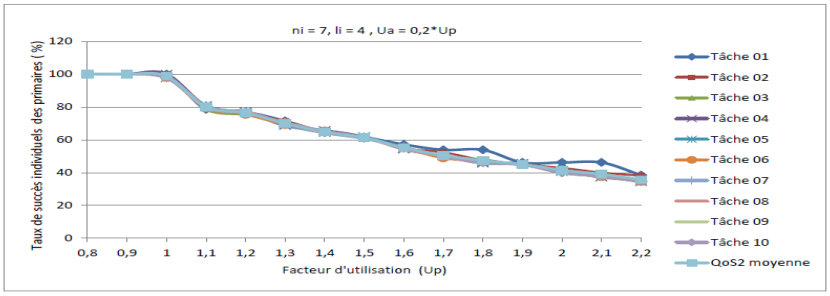


FIGURE 6.16 – $QdS2$ individuelle avec MSP.

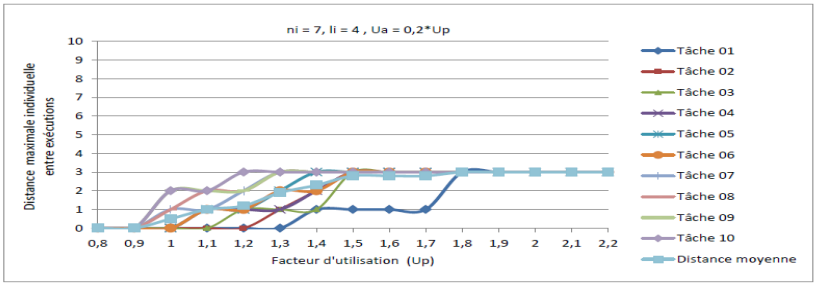


FIGURE 6.17 – Distance de réussite ($DMax1$) sans équité de service.

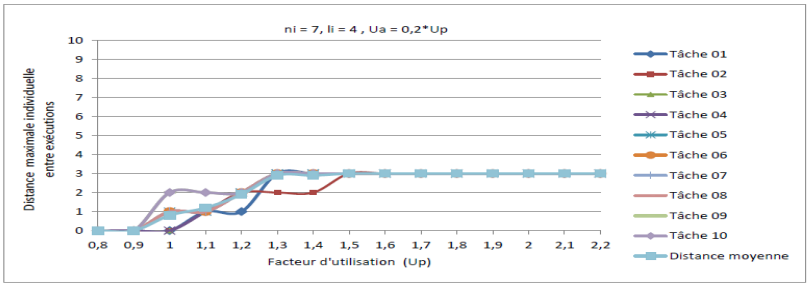


FIGURE 6.18 – Distance de réussite ($DMax1$) avec MDJ.

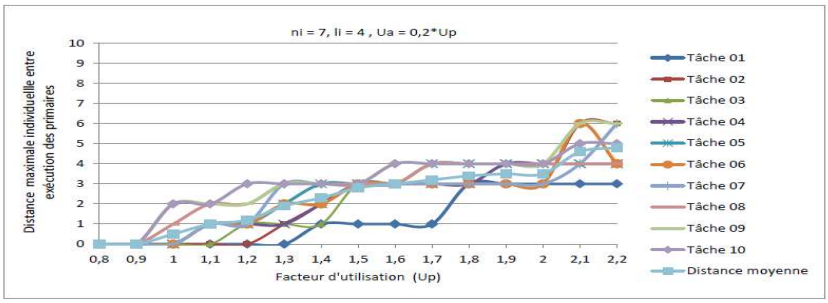


FIGURE 6.19 – Distance de réussite ($DMax2$) sans équité de service.

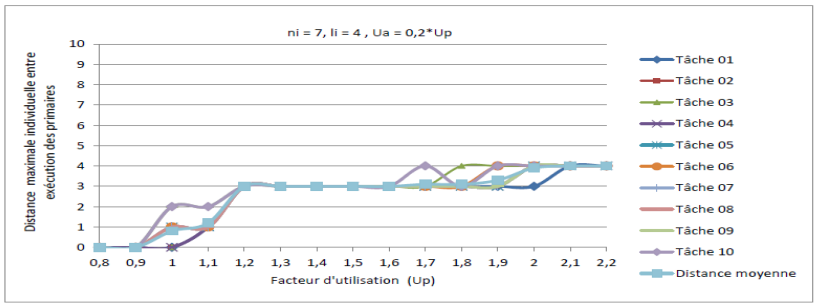


FIGURE 6.20 – Distance de réussite ($DMax2$) avec MDP.

Pour montrer l'influence de l'équité de service sur les $QoS1$ et $QoS2$ globales du système, nous présentons dans les figures suivantes, les résultats donnés par la stratégie $LCP - t$ sans et avec l'application d'algorithmes d'équité de service.
La figure 6.21, compare les résultats en $QoS1$ globale de la stratégie $LCP - t$ sans l'application d'algorithmes d'équité de service et $LCP - t$ avec l'application d'algorithmes d'équité de service respectivement MSJ, MSP, MDJ et MDP.
Nous rappelons que en l'absence d'algorithme d'équité de service, la politique d'attribution des priorités est celle de l'EDF.

La figure 6.22, compare les résultats en $QoS2$ globale de la stratégie $LCP-t$ sans l'application d'algorithmes d'équité de service et $LCP-t$ avec l'application d'algorithmes d'équité de service respectivement MSJ, MSP, MDJ et MDP.
A titre illustratif, pour $Up = 1.3$, l'écart de performance en terme de $QoS1$ globale des tâches, observé entre la stratégie $LCP-t$ avec EDF et les autres variantes est de l'ordre de 8.084%, 5.461%, 2.03%, et 8.756%, sous MSJ, MSP, MDJ et MDP respectivement.
pour $Up = 1.4$, l'écart de performance en terme de $QoS1$ globale des tâches, observé entre la stratégie $LCP-t$ avec EDF et les autres variantes est de l'ordre de 7.763%, 5.922%, 1.702%, et 5.743%, sous MSJ, MSP, MDJ et MDP respectivement.
Pour $Up = 1.3$, l'écart de performance en terme de $QoS2$ globale des tâches, observé entre la stratégie $LCP-t$ avec EDF et les autres variantes est de l'ordre de 6.881%, 3.428%, 4.298%, et 1.83%, sous MSJ, MSP, MDJ et MDP respectivement.
Pour $Up = 1.6$, l'écart de performance en terme de $QoS2$ globale des tâches, observé entre la stratégie $LCP-t$ avec EDF et les autres variantes est de l'ordre de 4.661%, 4.545%, 5.215%, et 1.761%, sous MSJ, MSP, MDJ et MDP respectivement.
En résumé, nous pouvons dire que le gain d'équité et d'uniformité de services au sens de la $QoS1$, acquis sous les variantes MSJ, MSP, MDJ et MDP s'assortit d'une perte de $QoS1$ globale du système.
Nous constatons aussi qu'il y a une diminution en $QoS2$ globale du système en contre partie de l'équité et d'uniformité de service au sens de la $QoS2$ pour les différentes stratégies en comparaison avec $LCP-t$ avec EDF.

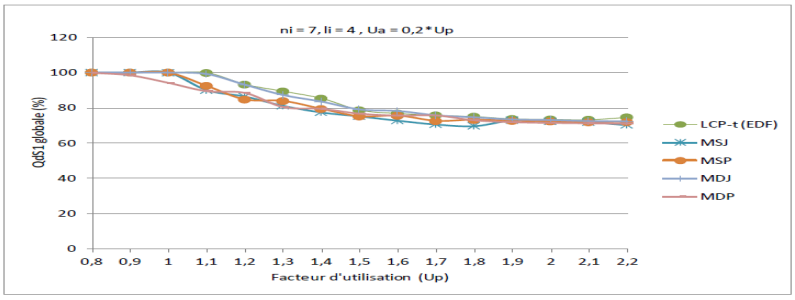


FIGURE 6.21 – $QdS1$ globale sous $LCP-t$, sans et avec équité de service.

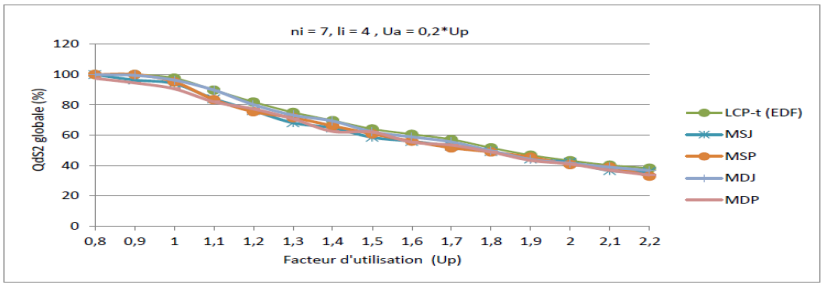


FIGURE 6.22 – $QdS2$ globale sous $LCP-t$, sans et avec équité de service.

6.8 Commentaires

Nous résumons dans le tableau suivant, les résultats de performances des quatre algorithmes proposées dans ce chapitre. On note par * une performance faible, ** une performance correcte, *** une très bonne performance. Notre commentaire synthétique concernant le choix d'une politique d'assignation des priorités se

Algorithme	$QdS1$	$QdS2$	Équité $QdS1$	Équité $QdS2$	Uniformité $QdS1$	Uniformité $QdS2$
MSJ	***	***	***	**	**	**
MSP	***	***	**	***	**	**
MDJ	***	***	**	**	***	**
MDP	***	***	**	**	**	***
EDF	***	***	*	*	*	*

TABLE 6.1 – Comparaison des politiques d'assignation des priorités

résume simplement comme suit :
Pour les applications temps réel où l'équité de service au sens de la $QdS1$ individuelle entre tâches est primordiale, on propose de choisir la stratégie $LCP - t$ avec l'application de la variante MSJ.
Pour les applications où l'équité de service au sens de la $QdS2$ individuelle entre tâches est prioritaire, on propose de choisir la stratégie $LCP - t$ avec l'application de la variante MSP.
Pour les applications temps réel où l'équité de service au sens de la $QdS1$ et de la $QdS2$ individuelles entre tâches est moyennent importante, on propose de choisir la stratégie $LCP - t$ avec EDF, compte tenu de l'augmentation avantageuse en termes de QdS globale du système.

6.9 Conclusion

Dans ce chapitre, nous avons étudié l'équité et l'uniformité de services et des stratégies d'ordonnancement $LCP-t$ et LCP. Après avoir souligné leur manque d'équité et d'uniformité de services, au sens de l'équilibre des $QdS1$ et $QdS2$ individuelles des tâches, nous avons proposé quatre variantes d'ordonnancement des instances de tâches Gp, Wa et Wp, à savoir MSJ, MSJ, MSJ, et MSJ et ce, en remplacement de Earliest Deadline First (EDF).
L'évaluation menée a permis de vérifier le gain d'équité et d'uniformité de services, procurés par ces nouvelles versions pour l'ensemble des ordonnanceurs. Ensuite, dans un souci de cohérence de ces nouvelles versions, nous avons étudié l'incidence du gain de d'équité de service sur la QoS globale observée au niveau du système.
En conclusion, nous pouvons dire que l'optimisation de l'équité de service s'accompagne d'une petite diminution de la QoS globale pour les différents ordonnanceurs.
Par conséquent, le choix de l'algorithme d'ordonnancement des instances de tâches Gp, Wa et Wp incombe à l'utilisateur qui doit sélectionner l'algorithme qui convient le mieux à l'application physique contrôlée, en fonction de ses exigences d'équité et d'uniformité de services au sens des $QdS1$ et $QdS2$ individuelles des tâches d'une part et de la QoS globale du système, d'une autre part.

Conclusion générale et perspectives

Conclusion générale

Notre travail de thèse a porté sur la problématique des systèmes temps réel soumis à des surcharges de traitement et donc incapables de satisfaire toutes les contraintes temporelles spécifiées. Quelle méthode peut-on utiliser pour faire en sorte que la surcharge de traitement ait le moins d'impact négatif au niveau applicatif ? En effet, un système informatique temps réel peut par exemple servir à contrôler un ou plusieurs processus. De la gestion des cas de surcharge dépend la qualité du contrôle/commande de l'environnement qui doit rester stable.

Ce manuscrit contient les différents travaux entrepris pour tenter de répondre à la question précédente posée. Nous avons d'abord synthétisé notre travail de recherche bibliographique au travers des deux premiers chapitres, mettant ainsi en évidence les différentes méthodes existantes pour la gestion de surcharge. En particulier, nous nous sommes arrêtés sur le modèle skip-over et le mécanisme à échéance.

Cela nous a amené à proposer une nouvelle manière, plus complète que celles existantes, pour modéliser une application temps réel. Le modèle de tâche appelé BGW a ainsi constitué notre première contribution. Dans un premier temps, nous avons observé le comportement d'ordonnanceurs simples à mettre en oeuvre car basés sur la technique de la première chance. En constatant la Qualité de service relativement modeste que ces ordonnanceurs délivrent, mesurée en termes d'échéances respectées, nous avons proposé d'autres ordonnanceurs plus complexes mais cependant plus performants.
L'étude de simulation réalisée pour des profils d'application différents a mis en relief le gain de performance réalisé par les ordonnanceurs appelés respectivement LCP et $LCP - t$. Ces deux ordonnanceurs se basent sur EDF pour assigner des priorités aux instances de tâches et sur la technique dite de la dernière chance qui augmente la chance d'exécuter avec succès les versions primaires de tâches. La stratégie $LCP - t$, au prix d'une plus grande complexité d'implémentation (en mémoire et overheads d'exécution) que LCP vient encore améliorer la Qualité de Service du système par l'utilisation d'un test d'admission en ligne avant toute exécution de tâche, conduisant à moins de gaspillage de temps processeur.

Bien que fondamental, le comportement d'un système temps réel ne peut se caractériser uniquement par les deux seules grandeurs de QdS appelés dans ce manuscrit QdS1 et QdS2. Ces deux grandeurs reflètent la performance en termes de ratio d'échéances satisfaites, quelque soient les versions de tâches exécutées (QdS1) ou par les seuls primaires exécutés (QdS2). Nous avons ainsi une évaluation de performance du système dans sa globalité. Toutefois, pour certaines applications de contrôle/commande en particulier, une observation du taux de respect des échéances au niveau individuel de chaque tâche peut s'avérer indispensable. C'est pourquoi, notre contribution a ensuite consisté à introduire deux nouvelles propriétés d'un ordonnanceur que nous avons appelées l'équité de service et l'uniformité de service.
Un ordonnanceur jugé équitable sera celui qui tendra à générer une qualité de service individuelle identique pour chaque tâche de l'application. Et un ordonnanceur sera dit uniforme si pour chaque tâche, il répartit

équitablement au cours du temps ses services aux différentes instances.
Notre contribution a donc consisté à proposer quatre nouvelles stratégies d'assignation des priorités en remplacement de EDF qui lui, se base uniquement sur l'urgence des tâches et donc en ignorant les traitements passés. Ainsi, nous préconisons d'utiliser les stratégies appelées MSJ et MSP pour une meilleure équité de service et les stratégies MDJ et MDP pour une meilleure uniformité de service. Chacune de ces stratégie peut s'employer alors conjointement à un ordonnanceur, en particulier LCP ou $LCP-t$ en raison de leur performance.

Dans cette thèse, nous nous sommes focalisés sur l'ordonnancement pour des architectures monoprocesseur sous-entendu mono-coeur. Or, de plus en plus, la complexité croissante des logiciels d'application conduit à se tourner vers des architectures multi-coeur, répondant à des besoins plus forts en capacité de traitement. Malheureusement, le passage de la théorie de l'ordonnancement monoprocesseur à celle de l'ordonnancement multiprocesseur n'est pas immédiate. Plusieurs approches existent, connus en tant qu'ordonnancement partitionné, ordonnancement semi-partitionné ou ordonnancement global. Nous suggérons d'étendre l'utilisation du modèle BGW à une architecture multiprocesseur en considérant successivement ces différentes approches.

Perspectives

Nous allons proposer à la fin de ce travail d'autres approches possibles d'ordonnancement temps réel pour gérer en ligne et de façon adaptative les surcharges.

La première proposition repose sur l'adaptation des paramètres du modèle BGW (paramètres de perte et de dégradation) pour la modification dynamique de la charge infligée au processeur. Nous parlerons alors de *modèle BGW dynamique*. La seconde proposition consiste à utiliser un bloc de contrôle d'admission des instances Grey ou White. Pour ces deux propositions, nous faisons appel à des techniques issues de l'automatique visant la régulation de la charge processeur.

Modèle BGW dynamique

Dans ce modèle, nous supposons ces paramètres variables afin de les rendre flexibles et adaptables en ligne en fonction de la disponibilité du processeur. En d'autres termes, nous visons à effectuer la régulation de la charge processeur grâce au changement dynamique des paramètres (distance entre primaires et facteur de pertes) du modèle BGW, représentés par les grandeurs n_i et l_i pour BGW1, n_i et s_i pour BGW2 et n_i et m_i, k_i pour BGW3.
Nous considérons démarrer avec des paramètres statiques ni_{max} et li_{min} proposés par le concepteur du système auxquels s'adjoignent les paramètres dynamiques ni_{dyn} et li_{dyn} satisfaisant les conditions suivantes :

$$ni_{min} \leq ni_{dyn} \leq ni_{max}$$

$$li_{min} \leq li_{dyn} \leq li_{max}$$

Par conséquent, chaque tâche τ_i du modèle BGW1 dynamique est caractérisée à tout instant par :
$(r_i, C_i^a, C_i^p, D_i, T_i, ni_{min}, ni_{dyn}, ni_{max}, li_{min}, li_{dyn}, li_{max})$ avec :

- r_i, D_i, T_i les paramètres temporels comme définis dans les chapitres précédents.
- C_i^a, C_i^p la durée d'exécution pire cas de la version secondaire et celle de la version primaire avec $C_i^a \leq C_i^p$.
- ni_{dyn}, li_{dyn} les deux paramètres dynamiques de QdS définis ci-dessus avec $0 \leq li_{dyn} \leq ni_{dyn}$.

Avec le BGW dynamique, nous proposons, deux façons de régulation automatique de la charge totale du processeur, soient :

- Adaptation des paramètres BGW par régulation PID : Dans l'objectif d'avoir un niveau souhaité de charge processeur, nous pouvons appliquer les méthodes de la régulation automatique (Feedback Control Scheduling ou Ordonnancement Régulé) de la charge [2], par le biais d'utilisation de l'un de régulateurs classiques de l'automatique de type P, PD, PI ou PID sur le Modèle BGW dynamique.
- Adaptation des paramètres BGW par la logique floue : La régulation par la logique floue est une technique bien adaptée aux modèles, non linéaires, complexes ou inconnus [1]. Elle ne dépend pas du modèle mathématique et ne prend en considération qu'un ensemble de règles qui gouvernent le système.

Classiquement, l'ordonnancement par rétroaction (feedback scheduling) sert à ajuster dynamiquement les paramètres des tâches temps réel (la période par exemple). L'ordonnanceur par rétroaction peut être implémenté comme une tâche périodique dont la priorité est la plus élevée. Du point de vue de la théorie de l'automatique, l'ordonnanceur représente ainsi le procédé à contrôler. Ses sorties incluent naturellement le taux d'utilisation du processeur par les tâches. Le contrôleur compare cette valeur de sortie à la valeur désirée. Il décide alors de la manière de rééchelonner les paramètres des tâches à appliquer.

Régulation de charge par contrôle d'admission

Nous donnons ici un aperçu d'une autre technique pour anticiper la surcharge via une régulation automatique du taux d'utilisation du processeur. Cette régulation se base sur l'utilisation d'un bloc de contrôle d'admission dont le rôle consiste à décider s'il faut sauter :

- une ou des versions primaires d'instances Grey (Wp).
- une ou des versions primaires d'instances White (Wp).
- une ou des versions secondaires d'instances White (Wa).

Cette décision se prend au regard des entrées et des sorties du système suivantes :

- niveau réel mesuré de la charge du processeur.
- niveau désiré de la charge du processeur.
- niveau mesuré de la Qualité de Service globale du système ($QdS1$).
- niveau désiré de la Qualité de Service globale du système ($QdS1$).
- niveaux mesurés des Qualités de Service individuelles des tâches.
- niveaux mesurés des Qualités de Service individuelles des tâches.

Le contrôleur d'admission s'appuie sur les techniques classiques de régulation d'automatique comme les régulateurs PID ou d'autres méthodes de régulation.

Bibliographie

[1] B. Bouchon-Meunier. *La logique floue*, 4e éd., Paris, Presses Universitaires de France « Que sais-je ? », 128 pages, 2007. 137

[2] D. Simon, Y.-Q.Song and O. Sename, *Control and Scheduling Joint Design*, in Real-time Systems Scheduling vol. 2 "Focuses", Maryline Chetto ed., ISBN-13 : 9781848217898, ISTE Wiley, 2014. 137

[3] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. *QoS negotiation in real-time systems and its application to automatic flight control*. IEEE Real-Time Technology and Applications Symposium, June 1997.

[4] A. K. Atlas and A. Bestavros. *Statistical rate monotonic scheduling*. In Proceedings of IEEE Real-Time Systems Symposium, December 1998.

[5] T. F. Abdelzaher. *QOS adaptaion in rea-time systems*. PHD thesis, University of Michigan 1999.

[6] T. F. Abdelzaher. *An Automated Profiling Subsystem for QoS-Aware Services*. IEEE Real-Time Technology and Applications Symposium, Washington D.C., June 2000.

[7] A. A. Aburas, V. Miho. *Fuzzy Logic Based Algorithm for Uniprocessor Scheduling*. Proceedings of the International Conference on Computer and Communication Engineering 2008 May 13-15, 2008 Kuala Lumpur, Malaysia.

[8] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. *A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers*. IEEE Real-Time Technology and Applications Symposium, Taipei, Taiwan, June 2001.

[9] G. C. Buttazzo, L. Abeni, and G. Lipari. *Elastic task model for adaptive rate control*. In IEEE Real Time System Symposium, Madrid, Spain, December 1998. 44

[10] R. Barbosa. Layered *Fault Tolerance for Distributed Embedded Systems*. PhD thesis, Department of Computer Science and Engineering Chalmers University of Technology. Göteborg, Sweden 2008.

[11] Bini, E., Buttazzo, G.-C., Buttazzo, G.-M., *A hyperbolic bound for the rate monotonic algorithm*. In Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, pp.59-66, 2001.

[12] G. Bernat and A. Burns and A. Llamosi, *Weakly-hard real-time systems*. IEEE Transactions on Computers, vol 50, pp.308–321, 2001. 46

[13] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli, *Rate modulation of soft real-time tasks in autonomous robot control system*. In IEEE Proceedings of the 11th Euromicro Conference on Real-Time Systems, June 1999.

[14] A. Bestavros and S. Nagy. *Value-cognizant Admission Control for RTDB Systems*. In proceedings of RTSS'96 : The 16th IEEE Real-Time Systems Symposium, 1996.

[15] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. *Elastic scheduling for flexible workload management*. IEEE Transactions on Computers, 51(3) :289-302, March 2002. 44

[16] Buttazzo, G., Stankovic, J. *RED Robust Earliest Deadline Scheduling*. In Proceedings of The 3rd International Workshop on Responsive Computing Systems, Austin, USA, 1993. 41

[17] Buttazzo, G., Stankovic, J., *Adding Robustness in Dynamic Preemptive Scheduling*. In Responsive Computer Systems : Towards integration of Fault-Tolerance and Real-Time. Kluwer Press, 1994.

[18] Buttazzo, G.-C., Sensini, F., *Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments*. IEEE Transactions on Software Engineering, 25(1) : 22-32, 1999.

[19] G. Buttazzo, M. Spuri, F. Sensini. *Value vs. Deadline Scheduling in Overload Conditions*. In Proceedings of the 15th Real-Time Systems Symposium, Pisa, Italy, 1995.

[20] G. C. Buttazzo, F. Sensini. *Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments*. IEEE Transactions on Software Engineering, 25(1) : 22-32, 1999.

[21] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer academic, 1997. 22, 24, 29

[22] M. Caccamo, G. C. Buttazzo. *Exploiting skips in periodic tasks for enhancing aperiodic responsivess*. In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97), 1997. 51, 52

[23] M. Caccamo, G. Buttazzo. *Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems*.In Proceedings of the IEEE Real-Time Computing Systems and Applications, 1998.

[24] M. Caccamo, G. Buttazzo, and L. Sha, *Elastic feedback control*. IEEE Proceedings of the Euromicro Conference on Real-Time Systems, Stockholm, Sweden, pp.121-128, 2000.

[25] H. Chetto, M. Chetto. *Some results of the earliest deadline scheduling algorithm*. IEEE Transactions on Software Engineering, vol.15.n0.10. : 1261-1269, October 1989. 32, 33, 34, 47, 49, 54, 104

[26] H. Chetto, M. Chetto. *A feasibility test for scheduling tasks in a distributed hard real-time system*. APII, 239-252, 1990. 54

[27] H. Chetto and M. Chetto. *An adaptive scheduling algorithm for fault-tolerant real-time system*. Software engineering journal May 1991. 47, 49, 104

[28] F. Cottet, J. Delacroix. C. Kaiser, Z. Mammeri. *Ordonnancement temps réel*. Paris, Hermes Sciences Publications, 2000. 24, 30

[29] F. Cottet, J. Delacroix. C. Kaiser, Z. Mammeri. *Scheduling in real-time systems*. John Wiley and Sons ltd, 2002. 45

[30] A. Cervin and J. Eker. *Feedback scheduling of contol tasks*. In Proceedings of the 39th IEEE Conference on Decision and Control, Sydney, Australia, Dec. 2000.

[31] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Arzén. *Feedback-feedforward scheduling of control tasks*. Real-Time Systems, vol. 23, no. 1, 2002.

[32] S. Cen. *A Software Feedback Toolkit and its Application In Adaptive Multimedia Systems*. Ph.D. Thesis, Oregon Graduate Institute, October 1997.

[33] M. Chrobak, L. Epstein, J. Noga, J. Sgall, R. van Stee, T. Tichy, N. Vakhania. *Preemptive Scheduling in Overloaded Systems*. In Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP2002), pp 800-811, Malaga, Spain, 2002.

[34] A. Cervin. *Towards the integration of control and real-time scheduling design*. Department of Automatic Control, Lund Institute of Technology, Sweden, Tech. Rep. Licentiate thesis, May 2000.

[35] A. Cervin. *Integrated Control and Real-Time Scheduling*. Department of Automatic Control Lund Institute of Technology Lund, April 2003.

[36] R.-H. Campbell, K.-H. Horton, G.-G. Belford, *Simulations of a fault-tolerant deadline mechanism*. Digest of papers FTCS-9, pp 95-101, 1979. 47, 49

[37] C-C. Han, K. G. Chih, W. Jian. *fault-tolerant scheduling algorithm for real-time periodic tasks with software faults*. IEEE Transactions on computers, vol.52. n0.3. March 2003.

[38] M. Caccamo, G. Lipari, G. Butazzo. *Sharing resources among periodic and aperiodic tasks with dynamic deadlines*. In Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.

[39] J.-Y. Chung, J.-W.-S. Liu, K. Lin. *Scheduling periodic jobs that allow imprecise results*. In IEEE Transactions on Computers, 39(9) : 1156-1174, 1990. 43, 46

[40] Campbell, A., McDonald, P., and Ray, K. *Single event upset rates in space*. IEEE Trans. on Nuclear Science 39(6) :1828-1835. 1992. 38

[41] Castillo, X., McConnel, S. R., and Siewiorek, D. P. *Derivation and caliberation of a transient error reliability model*. IEEE Trans. on Computers C-31(7) : 658 ?671. 1982.

[42] F. Dorin. *Contributions à l'ordonnancement et l'analyse des systèmes temps réel critiques*. Thèse de doctorat, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique. 30 septembre 2010.

[43] R. Davis and A. Wellings. *Dual priority scheduling*. the 16th IEEE Real-Time Systems Symposium (RTSS'95). 1995. 46

[44] J. Eker, P. Hagander, K. E. Årzén . *A Feedback Scheduler for Real-time Controller Tasks*. In IFAC Control Engineering Practice 2000.

[45] J. Eker. *Flexible Embedded Control Systems-Design and Implementation*. PhD-thesis, Lund Institute of Technology, December 1999.

[46] T. Garcia-Fernandez. *Conception et développement de composants pour logiciels temps-réel embarqués*. Thèse de Doctorat, Université de Nantes, 2005.

[47] R. HU, Z. HU. *A Scheduling Algorithm Aimed at Time and Cost for Meta-tasks in Grid Computing Using Fuzzy Applicability*. Proceedings of the Eighth International Conference on High Performance Computing in Asia-Pacific Region (HPCASIA'05) 0-7695-2486-9/05 20.00 © 2005 IEEE.

[48] H.-S. Hu, D. Liu, M.-D. Lemmon, Q. Ling. *Firm real-time system scheduling based on a novel QoS constraint*. In Proceedings of the 24th Real-Time Systems Symposium (RTSS'03), Cancun, Mexico, 2003.

[49] K.-I.-J. Ho, J.-Y.-T. Leung, W.-D. Wei. *Minimizing maximum weighted error of imprecise computation tasks*. Technical Report, Dep. Of Computer Science and Engineering. Univ. of Nebraska, USA, 1992.

[50] M. Hamdaoui, P. Ramanathan. *A dynamic priority assignment technique for streams with (m,k)-firm deadlines*. IEEE Transactions on Computers, 44(4) : 1443- 1451, 1995. 46

[51] C. C. Han, K. G. Shin, and J. Wu, *A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults*. IEEE Transaction on Computers, vol. 52, no. 3, pp. 362-372, March 2003.

[52] X.B. He, G.Y. Zhang, M. Liu, Y.P. Zhao, W. Li. *A Fuzzy EDF Scheduling Algorithm Being Suitable for Embedded Soft Real-time Systems in the Uncertain Environments*. 978-1-4244-5848-6/10/ 26.00 ©2010 IEEE.

[53] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. *Measurement and Modeling of Computer Reliability as Affected by System Activity*. ACM Trans. On Computer System, 4(3) :214-237, Aug. 1986.

[54] K. Jeffay and D. L. Stone. *Accounting for interrupt handling costs in dynamic priority task systems*. In Proceedings of the 14th IEEE Real-Time Systems Symposium, pages 212.221, December 1993. 32

[55] A. Koubaa. *Gestion de la Qualité de Service temporelle selon la contrainte (m,k)-firm dans les réseaux à commutation de paquets*. Thèse de doctorat, Institut National Polytechnique de Lorraine, 2004.

[56] L.Krishnamurthy. *AQUA : An Adaptive Quality of Service Architecture for Distributed Multimedia Applications*. PhD thesis, University of Kentucky, 1997.

[57] G. Koren and D. Shasha. *D-over : An optimal on-line scheduling algorithm for overloaded real-time systems*. In Proceedings of the IEEE Real-Time Systems Symposium, Phoenix, Arizona, USA, 1992. 40

[58] G. Koren and D. Shasha. *Skip-over : Algorithms and complexity for overloaded systems that allow skips*. IEEE Real Time Systems Symposium, December 1995. 45, 50, 51, 52, 53

[59] G. N. Khan and A. Sydhom. *Fault-Tolerant Scheduling of Real-Time Tasks Having Software Faults*. IEEE CCECE/CCGEI, Saskatoon, May 2005.

[60] C.-D. Locke. *Best-effort Decision Making for Real-Time Scheduling*. PhD Thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[61] A. L. Liestman and R.H. Campbell. *A fault-tolerant scheduling problem*. IEEE Transactions on Software Engineering, 12(11) :1089-95, November 1986. 47, 49

[62] J. Y-T. Leung. *Handbook of Scheduling. Algorithms, Models, and Performance Analysis*. CRC Press LLC 2004. 24

[63] J. LI. *Garantir la qualité de service temps réel selon l'approche (m,k)-firm*. Thèse de doctorat, Institut National Polytechnique de Lorraine 14 Février 2007.

[64] C. Liu and J. Layland. *Scheduling algorithms for multiprogramming in real-time environment*. Journal of ACM, 1(20) :46-61, October 1973. 27, 28, 29

[65] J.-W.-S. Liu, K.-J. Lin, R. Bettati, D. Hull, A. Yu. *Use of Imprecise Computation to Enhance Dependability of Real-Time Systems*. In Gary M. Koob and Clifford G. Lau, editors, Foundations of Dependable Computing : Paradigms for Dependable Applications, chapter 3.1, pp 157-182. Kluwer Academic Publishers, 1994. 43

[66] J. W. S. Liu, K. J. Lin, and S. Natarajan. *Scheduling real-time, periodic jobs using imprecise results*. In Proceedings of the IEEE Real-Time System Symposium, December 1987. 43

[67] J. W. S. Liu, K. Lin, W. Shih, A. Yu, C. Chung, J. Yao, and W. Zhao. *Algorithms for scheduling imprecise computations*. IEEE Computer, 24(5) :58-68, May 1991. 43

[68] J. Leung and M. Merril. *A Note on Preemprive Scheduling of Periodic Real-Time Tasks*. Information Processing Letters, 11(3) : 115-118, 1980. 27

[69] S. Lauzac, R, Melhem, and D, Mosse. *An Improved Rate-Monotonic Admission Control and Its Applications*. IEEE transactions on coputers, Vol. 52, No. 3, MARCH 2003.

[70] B. Li, K. Nahrstedt. *A Control Theoretical Model for Quality of Service Adaptations*. In IEEE International Workshop on Quality of Service, May 1998.

[71] K. J. Lin, S. Natarajan, and J. W. S. Liu. *Concord : a system of imprecise computation*. In Proceedings of the 1987 IEEE Compsac, October 1987. 43

[72] J.-P. Lehozcky, S. Ramos-Thuel. *An optimal algorithm for scheduling soft aperiodic tasks in fixed-priority preemptive systems*. In Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 110-123, 1992.

[73] C.Lee, R. Rajkumar and C. Mercer. *Experiences with processor reservation and dynamic QoS in real-time mach*. In Proceedings of Multimedia Japan 96, April 1996.

[74] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son and M. Marley. *Performance Specifications and Metrics for Adaptive Real-Time Systems*. IEEE Real-Time Systems Symposium, Orlando, FL, Dec 2000.

[75] J.-P. Lehozcky, L. Sha, Y. Ding, *The rate-monotonic scheduling algorithm : exact characterization and average case behaviour*. In Proceedings of the IEEE Real-Time Systems Symposium, pp. 166-171, 1989.

[76] J.-P. Lehozcky, L. Sha, K.-K. Strosnider. *Enhanced aperiodic responsivess in hard real-time environments*. In Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 261-270, 1987. =

[77] C. Lu, J. A. Stankovic, S. H. Son, G. Tao. *Feedback Control Real-Time Scheduling : Framework, Modeling, and Algorithms*. Journal of Real-Time Systems, 23, 85-126, 2002.

[78] C. Lu, J. A. Stankovic, G. Tao and S. H. Son. *Design and Evaluation of a Feedback Control EDF Scheduling Algorithm*. 20th IEEE Real-Time Systems Symposium, December 1999.

[79] T.-W. Lam, K.-K. To. *Performance Guarantee for Online Deadline Scheduling in the Presence of Overload*. In Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, pp 755-764, New-York, USA, 2001.

[80] J. Lee, A. Tim and J. Yen. *A Fuzzy Rule-Based Approach to Real-Time Scheduling*. 0-7803-1896-XI94 4.00 01994 IEEE.

[81] C. Lu. *Feedback Control Real-Time Scheduling*. PhD thesis, University of Virginia. May 2001.

[82] J.-Y.-T. Leung, J. Whitehead. *On the complexity of fixed-priority scheduling of periodic real-time tasks. Performance evaluation*. 2 :237-250, 1982. 28

[83] J.-Y.-T. Leung, C.S. Wong. *Minimizing the number of late tasks with error constraints*. In Proceedings of the 11th Real-Time Systems Symposium (RTSS'90), Orlando, USA, 1990.

[84] A. Mohammadi and S. G. Akl. *Scheduling Algorithms for Real-Time Systems*. School of Computing, Queen's University. Technical Report No. 2005-499. July 15, 2005.

[85] P. Martineau, *Ordonnancement en-ligne dans les systèmes informatiques temps-réel*. PhD Dissertation, University of Nantes, 1994.

[86] P. Marti. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. Ph.D. dissertation, Automatic Control and Computer Engineering Department, Technical University of Catalonia.

[87] A. Marchand. *Ordonnancement temps réel avec contraintes de qualité de service. De la théorie à l'intégration*. Thèse de doctorat, Université de Nantes 02 octobre 2006. 53, 54

[88] D. Masson. *Intégration des événements non périodiques dans les systèmes temps réel - Application à la gestion des événements dans la spécification temps réel pour Java*. Thèse de doctorat, université Paris-Est. 8 décembre 2008.

[89] A. K. Mok and D. Chen. *A Multiframe Model for Real-Time Tasks*. IEEE Transactions on Software Engineering, 23(10), October 1997.

[90] A. K. Mok and M. L. Dertouzos. *Multiprocessor scheduling in hard Real-Time environment*. In Proceedings of the 7th Texas Conference on Computer systems, 1978. 30

[91] P. Meumeu Yomsi. *Prise en compte du coût exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples*. Thèse de doctorat, université Paris-Sud11. 02Avril 2009.

[92] C. Montez, J. Fraga. *Dealing with overloading in Tasks Scheduling*. In proceedings of the 12th International Conference of the Clilean Computer Science Society (SCCC'02), Copiapo, Atacama, Chile, 2002. 46

[93] P. Marti, G. Fohler, K. Ramamritham, and J.M. Fuertes. *Improving Quality-of-Control using Flexible Timing Constraints : Metric and Scheduling Issues*. Real-Time Systems Symposium, Dec. 2002.

[94] P. Mejia-Alvarez, R .Melhem, D. Mosse. *An Incremental Approach to Scheduling During Overloads in Real-time Systems*. In Proceedings of the Real Time Systems Symposium (RTSS'00), Orlando, Florida, USA, 2000.

[95] D. Mosse, M. Pollack, Y. Ronen. *Value Density Algorithms to Handle Transient Overloads in Scheduling*. In Proceedings of the 11th Euromicro Conference on Real-Time Systems, York, England, 1999.

[96] B. Li, K. Nahrstedt. *A Control-based Middle-ware Framework for Quality of Service Adaptations*. IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms, 17(9), pp. 1632-1650, Sep. 1999.

[97] T. Nakajima, *Resource reservation for adaptive QoS maping in real-time mach.* In sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), April 1998.

[98] N. Navet, *Systèmes temps réel 2*. Hermes science, 2006. 32

[99] J. Nilsson, *Real-time control systems with delays*. Ph.D. dissertation, Department of Automatic Control, Lund Institute of Technology, Sweden, Jan. 1998.

[100] T. Nakajima, Hiroshi Tezuka, *A continuous media application supporting dynamic QoS contron on real-time mach.* In ACM Multimedia, 1994.

[101] Y. Oh *The design and analysis of scheduling algorithms for real-time and fault-tolerant computer systems*. Ph.D. Thesis, University of Virginia. 1994.

[102] S. K. Oh, and G. MacEwen *Toward fault-tolerant adaptive real-time distributed systems*. External Technical Report 92-325, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada. 1992.

[103] Y. Oh, and S. H. Son *Enhancing fault-tolerance in rate-monotonic Scheduling*. The Journal of Real-Time Systems 7(3) :315 ?329. 1994.

[104] R. M. Pathan. *Scheduling Algorithms For Fault-Tolerant Real-Time Systems*. Thesis for the degree of licentiate of engineering. Chalmers University of Technology. Göteborg, Sweden 2010.

[105] S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, J. Bigus. *Using Control Theory to Achieve Service Level Objectives in Performance Management*. IFIP/IEEE International Symposium on Integrated Network Management, 2001.

[106] Pandya, M., and Malek, M. *Minimum achievable utilization for fault-tolerant processing of periodic tasks*. Technical Report TR 94-07, Univ of Texas at Austin, Dept of Computer Science. 1994.

[107] Pradhan, D. K. *Fault Tolerant Computing : Theory and Techniques*. Prentice-Hall, NJ. 1986.

[108] E. Poggi, Y.-Q. Song, A. Koubaa, Z. Wang. *Matrix-DBP for (m,k)-firm guarantee*. In Real-Tme and Embedded Systems, Paris, 2003.

[109] S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, Department of Computer Science, University of York, June 1997.

[110] G. Quan, X. Hu. *Enhanced fixed-priority scheduling with (m,k)-firm guarantee*. In Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00), Orlando, Florida (USA), pp. 7988, 2000.

[111] A. Rahni . *Contributions à la validation d'ordonnancement temps réel en présence de transactions sous priorités fixes et EDF*. Thèse de doctorat, université de Poitiers, 05 décembre 2008.

[112] Ramos-Thuel, S. *Enhancing fault tolerance of real-time systems through time redundancy*. Ph.D. Thesis, Carnegie Mellon University. 1993.

[113] P. Ramanathan. *Graceful degradation in real-time control applications using (m,k)-firm guarantee*. IEEE 1997.

[114] P. Ramanathan. *Overload management in real-time control applications using (m,k)-firm guarantee*. IEEE Transactions on Parallel and Distributed Systems, 10(6), 1999.

[115] M. Ryu and S. Hong. *Toward Automatic Synthesis of Schedulable Real-Time Controllers*. Integrated Computer-Aided Engineering, 5(3) 261-277, 1998.

[116] S. G. Robertz, D. Henriksson, A. Cervin. *Memory-Aware Feedback Scheduling of Control Tasks*. 1-4244-0681-1/06/ 20.00 '2006 IEEE.

[117] F. Ridouard . *Contributions à des problèmes d'ordonnancement en- ligne : l'ordonnancement temps réel de tâches à suspension et l'ordonnancement par une machine à traitement par lot*. Thèse de doctorat, université de Poitiers, 14 novembre 2006.

[118] Y. Ronen, D. Mosse, M.-E. Pollack. *Using Value-Density Algorithms to Handle Transient Overloads in Deliberation Scheduling*. In IEEE Expert, Special Issue on Real-Time Intelligent Systems, 2003.

[119] P. Richard and J. Goossenst. *Ordonnancement de tâches indépendantes avec suspension*. In Proceedings of the 13th International Conference on Real-Time Systems, 2005.

[120] K. Ramamritham and J. A. Stankovic. *Dynamic task scheduling in distributed hard real-time systems*. IEEE Software, Vol. 1, No. 3, July 1984.

[121] F. Rodrigues de la Rocha, R. Silva de Oliveira, C. Montez. *Uniprocessor Scheduling Under Time-Interval Constraints*. ETFA'2007 - 12th IEEE Int. Conf. on Emerging Technologies and Factory Automation.

[122] M. Spuri, G.-C. Buttazzo, F. Sensini. *Robust Aperiodic Scheduling under Dynamic Priority Systems*. In Proceedings IEEE Real-Time Systems Symposium, Pisa, Italy, 1995.

[123] D. Simon, E. Castillo, and P. Freedman, *Design and analysis of synchronization for real-time closed-loop control in robotics*. IEEE Trans. on Control Systems Technology, vol. 6, no. 4, pp. 445 ?461, july 1998.

[124] M. Sabeghi, H. Deldari, V. Salmani, M. Bahekmat and T. Taghavi. *A Fuzzy Algorithm for Real-Time Scheduling of Soft Periodic Tasks on Multiprocessor Systems*. ISBN : 972-8924-09-7 © 2006 IADIS.

[125] M. Silly. *La tolérance aux fautes dans un système temps réel à contraintes strictes*. INRIA, rapport de recherche No 512, 1986.

[126] M. Silly-Chetto. *Sur la problématique de l'ordonnancement dans les systèmes informatiques temps-réel*. Rapport d'HDR, Université de Nantes, 1993.

[127] M. Silly. *The EDL server for scheduling periodic and soft aperiodic tasks with resource constraints*. The Journal of Real-Time Systems, 17 : 1-25, 1999. 33

[128] Y.-Q. Song, A. Koubaa. *Gestion dynamique de la QdS temps-réel selon (m,k)- firm*. École Temps Réel (ETR'03), Toulouse, 2003.

[129] Siewiorek, D. P., Kini, V., Mashburn, H., McConnel, S., and Tsao, M. *A case study of C.mmp, Cm, and C.vmp : Part 1 ?Experiences with fault tolerance in multiprocessor systems*. Proceedings of the IEEE, 66(10) :1178 ?1199. 1978. 38

[130] W.-K. Shih, W.-S. Liu. *Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error*. In Proceedings of the IEEE Transactions on Computers, 44(3), 1995.

[131] J.-K. Strosnider, J.-P. Lehoczky, L. Sha. *The Deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. IEEE Transactions on Computer, 44(1), 1995.

[132] D. Seto, J.-P. Lehoczky, L. Sha and K.G. Shin. *On task schedulability in real-time control system*. In Proceedings of the IEEE real-time systems Symposium, December 1996.

[133] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. *The Case for Feedback Control Real-Time Scheduling*. EuroMicro Conference on Real-Time Systems, June 1999.

[134] K. G. Shin and C. L. Meissner. *Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment*. EuroMicro Conference on Real-Time Systems, June 1999.

[135] P.-G. Sorenson, *A methodology for real-time system developement*. PhD Thesis, University of Toronto, Canada, 1974.

[136] M. Spuri, G.-C. Buttazzo. *Efficient aperiodic service under earliest deadline scheduling*. In Proceedings of the IEEE Real-Time Systems Symposium, 1994.

[137] M. Spuri, , G.-C. Buttazzo. *Scheduling aperiodic tasks in dynamic priority systems*. The Journal of Real-Time Systems, 10(2), 1996.

[138] L. Sha, R. Rajkumar, J.-P. Lehoczky. *Priority inheritance protocols : An approach to real-time synchronisation*. IEEE Transactions on Computers, 578-585, 1990.

[139] B. Sprunt, L. Sha, J.-P. Lehoczky. *Aperiodic task scheduling for hard real-time systems*. The Journal of Real-Time Systems, 1 : 27-60, 1989.

[140] J.-A. Stankovic, M. Spuri, K. Ramamritham, G.-C. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 273p., 1998.

[141] J-A. Stankovic, *A Serious Problem for Next-Generation Systems*. IEEE Computer, 21(10) :10-19, 1988.

[142] P. Santiprabhob, T. Thumthawatworn, K. Jitwongtrakul and R. Chiersilp. *Intelligent Process Scheduling and Dispatching for FPGA-Based Computing Platform*.IEEE AC paper 1311, Version 3, Updated January 5, 2005.

[143] H. Streich, *Task Pair-scheduling : An approach for dynamic real-time systems*. In Proceedings of the 2nd workshop on Parallel and Distributed Real-Time Systems, Cancun, Mexico, 1994.

[144] C. Lu, J. A. Stankovic, G. Tao and S. H. Son. *Design and Evaluation of a Feedback Control EDF Scheduling Algorithm*. IEEE Real-Time Systems Symposium, Phoenix, AZ, Dec 1999.

[145] M. S. Abu Talip, A. H. Abdalla, A. Asif, A. Ali Aburas. *Fuzzy Logic Based Algorithm for Disk Scheduling Policy*. 2009 International Conference of Soft Computing and Pattern Recognition. 978-0-7695-3879-2/09 26.00 © 2009 IEEE.

[146] M. S. Abu Talip, A. H. Abdalla, A. Ali Aburas, A.H.M. Zahirul Alam, A. Asif. *Knowledge-Based Disk Scheduling Policy using Fuzzy Logic*. International Conference on Computer and Communication Engineering (ICCCE 2010), 11-13 May 2010, Kuala Lumpur, Malaysia. 978-1-4244-6235-3/10/ 26.00 ©2010 IEEE.

[147] T.-S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.-C. Wu, J.-W.- S. Liu. *Probabilistic performance guarantee for real-time tasks with varying computation times*. In Proceedings of the Real-Time Technology and Applications Symposium, pp 164-173, 1995.

[148] T-S. Tia, J.-W.-S. Liu, M. Shankar. *Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems*. The Journal of Real-Time Systems, 1995.

[149] West, R., Ganev, I., Schwan, K., *Window-constrained process scheduling for Linux systems*. In Proceedings of the 3rd Real-Time Linux Workshop, Milan, Italy, 2001. 46

[150] K. Wang, T. H. Lin. *Scheduling adaptive tasks in real-time systems*. In Proceedings of the 21st IEEE Real-Time Systems Symposium, December 1994. 44

[151] B.Wang, Q. Li, X. Yang and X. Wang. *Robust and Satisfactory Job Shop Scheduling under Fuzzy Processing Times and Flexible Due Dates*. Proceedings of the 2010 IEEE International Conference on Automation and Logistics August 16-20 2010, Hong Kong and Macau. 978-1-4244-8376-1/10/26.00 ©2010 IEEE.

[152] F. Wang, K. Ramamritham and J.A. Stankovic. *Determining Redundancy Levels for Fault Tolerant Real-Time Systems*. IEEE Transactions on Computers. Vol. 44, No. 2, February 1995.

[153] L. R. Welch and B. A. Shirazi, *A Dynamic Real-time Benchmark for Assessment of QoS and Resource Management Technology*. IEEE Real-time Technology and Applications Symposium, June 1999. 46

Bibliographie personnelle

Ould Sass Mohamed, Chetto Maryline. *BGW : A Novel QoS Model for Firm Real-time Computing Systems*. Rapport interne de l'IRCCYN, 2013. http ://hal.archives-ouvertes.fr/docs/00/82/25/57/PDF/Technical-reportIRCCyN.pdf

Ould Sass Mohamed, Chetto Maryline. *BGW model : An Approach for Overload Management in Fault Tolerant Real-time Systems*. 13ièmme journée des doctorant de l'EDSTIM JDoc2013 Saint-Nazaire 2013.

Ould Sass Mohamed, Chetto Maryline, Queudet Audrey. *The BGW model for QoS aware scheduling of real-time embedded systems.* MobiWac '13 Proceedings of the 11th ACM international symposium on Mobility management and wireless access. Pages 93-100 ACM, November 3-8, 2013, Barcelona, Spain.

Ould Sass Mohamed, Chetto Maryline. *Schedulers for BGW Tasks to Guarantee Quality of Service of Embedded Real-time Systems*. PECCS 2015, 5th International Conference on Pervasive and Embedded Computing and Communication Systems. February, 2015, Angers, France.

l'université nantes angers le mans
COMMUNAUTÉ D'UNIVERSITÉS ET ÉTABLISSEMENTS

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Mohamed OULD SASS

Le modèle BGW pour les systèmes temps réel surchargés

Ordonnancement monoprocesseur

BGW: New task model for overloaded real time systems

Monoprocessor scheduling

Résumé

Les systèmes temps-réel embarqués se retrouvent dans des domaines d'application très variés : avionique, automobile, environnement, santé, etc. Ils doivent offrir un nombre croissant de fonctionnalités et fournir un niveau maximal de Qualité de Service (QdS) et ce, malgré des défaillances liées à l'occurrence de fautes ou de surcharges de traitement. Pour ce type de système informatique, la QdS se mesure principalement en termes d'échéances respectées car les programmes sont caractérisés par des dates de fin d'exécution au plus tard. Dans cette thèse, nous considérons une architecture monoprocesseur pour une application temps réel dite ferme. La première contribution tient dans la proposition d'un nouveau modèle de tâche appelé BGW qui permet de spécifier la nature de ses contraintes temporelles. Ce modèle est tiré des deux approches Skip-Over et Deadline Mechanism. La première est dédiée à la gestion des surcharges de traitement par la perte contrôlée de certaines instances de tâches. La seconde est une technique de tolérance aux fautes temporelles basée sur de la redondance logicielle dynamique passive avec deux versions. Dans une seconde partie, nous proposons de nouveaux ordonnanceurs temps réel basés sur EDF (Earliest Deadline First) pour des tâches BGW. Nous montrons comment maximiser la QdS tout en tenant compte des critères d'équité de service. Une étude de performance en termes de QdS et d'overheads conforte nos propositions.

Mots clés

Système temps réel, modèle de tâches périodiques, ordonnancement, monoprocesseur, gestion de surcharge.

Abstract

Real-time embedded systems are found in various application domains. They have to offer an increasing number of functionalities and to provide the highest Quality of Service despite possible failures due to faults or processing overloads. In such systems, programs are characterized by upper bounds on finishing times and the QoS is assessed by the ratio of successful deadlines. In this thesis, we deal with this issue. We focus on a uniprocessor architecture in the framework of a firm real-time application that accepts deadline missing under some specified limits. Tasks are assumed to be periodic. Our first contribution lies in the proposition of a novel model for tasks which is called BGW model. It is drawn from two approaches respectively known as the skip-over model and the Deadline Mechanism. The first one provides timing fault-tolerance through passive dynamic software redundancy with two versions. The second one copes with transient processing overloads by discarding instances of the periodic tasks in a controlled and pre-specified way. We give a feasibility test for this model. In a second part, we describe the behavior of dynamic priority schedulers based on EDF (Earliest Deadline First) for BGW task sets. A performance analysis is reported which is mainly related to QoS evaluation and measurement of overheads (complexity of the scheduler). The following contribution concerns more sophisticated schedulers that permit to enhance the QoS as to improve service balancing.

Key Words

Real-time systems, periodic task model, scheduling, monoprocessor, overload management.