Université de Nantes

École Doctorale

Sciences et Technologies de l'Information et des Matériaux

Année : 2006

Thèse de Doctorat de l'Université de Nantes

Spécialité : Automatique et Informatique Appliquée

Présentée et soutenue publiquement par :

Audrey Marchand

le 02 octobre 2006
à l'École Centrale de Nantes

ORDONNANCEMENT TEMPS RÉEL AVEC CONTRAINTES DE QUALITÉ DE SERVICE

- De la théorie à l'intégration -

Jury

Rapporteurs	YeQiong Song	Prof. des universités - INPL Nancy
	Yves Sorel	Directeur de Recherche - INRIA Rocquencourt
Examinateurs	Frédéric Benhamou	Prof. des universités - Université de Nantes
	Alfons Crespo	Prof. des universités - Université de Valencia (Espagne)
	Joël Goossens	Maître de conférences - Université de Bruxelles (Belgique)
	Maryline Silly-Chetto	Prof. des universités - Université de Nantes
Invité	Philippe Gerum	Industriel - Société Open Wide Paris

Directrice de thèse : Maryline Silly-Chetto
Laboratoire : IRCCyN
Composante de rattachement du directeur de thèse : IUT de Nantes

N° ED 366-266

Remerciements

Je remercie M. Frédéric BENHAMOU, Professeur à l'Université de Nantes, qui m'a fait l'honneur de présider le jury de cette soutenance.

Je remercie également M. Yeqiong SONG, Professeur à l'INPL de Nancy et M. Yves SOREL, Directeur de Recherche à l'INRIA Rocquencourt d'avoir bien voulu accepter, pendant la période estivale, la charge de rapporteur.

Je remercie M. Alfons CRESPO, Professeur à l'Universite Polytechnique de Valencia et M. Joël GOOSSENS, Maître de conférences à l'Université Libre de Bruxelles, de s´être déplacés depuis leurs laboratoires étrangers pour venir apprécier ce travail.

Je remercie M. Philippe GERUM, Ingenieur chez Open Wide Paris, d'avoir accepté l'invitation à la soutenance et d'avoir apporté sa vision industrielle.

Je tiens tout particulièrement à remercier ma directrice de thèse, Maryline SILLY-CHETTO, qui par sa disponibilité et sa compétence, m'a permis de mener à bien ces trois années de recherche. Merci pour toutes nos discussions enrichissantes menées dans un cadre communicatif et amical.

J'associe à mes remerciements l'ensemble des permanents et doctorants du laboratoire qui m'ont accompagnée pendant ce travail de thèse. En particulier, je tiens à remercier Isabelle, pour nos agréables conversations sur le chemin du RU, Thibault, joyeux apprenti pâtissier, Julien, journaliste des news locales, Morgan, agent compétent du service de reproduction des thèses de l'équipe, et tous les autres Jordan, Pierre-Em, Didier et Mikaël qui m'ont aidée et supportée durant ma thèse.

Je remercie aussi de tout coeur ma famille qui m'a soutenue et encouragée durant la durée de ce travail. J'aimerais pour finir remercier mon amour Fred, pour sa présence à mes côtés au quotidien, et à qui j'ai dû accorder moins de temps qu'il ne le méritait durant ces 3 dernières années.

A mes parents, à Fred,
et à la mémoire de mon papy...

"Le bonheur est souvent la seule chose que l'on puisse donner sans l'avoir et c'est en le donnant qu'on l'acquiert."

Voltaire

Table des matières

Table des figures

Liste des tableaux

Introduction générale

La perspective traditionnelle de la théorie de l'ordonnancement temps réel s'est largement reposée sur la communauté du temps réel à contraintes strictes (TRCS) pour lesquels la violation de l'échéance d'une tâche (dite critique), a souvent des conséquences catastrophiques sur le système contrôlé ainsi que sur son environnement.

Cependant, cette perspective peut sembler parfois restrictive et peu adaptée pour un certain nombre d'applications émergentes pour lesquelles, la gestion des cas de surcharge réside dans le fait d'écarter certaines instances de tâches, de façon à améliorer l'utilisation effective des ressources, tout en minimisant la dégradation causée par le rejet de ces instances. L'objectif consiste alors à concevoir pour ce type d'applications des stratégies d'ordonnancement assurant une dégradation contrôlée de la Qualité de Service (QoS), en particulier dans des phases de surcharge.

Considérons à titre illustratif l'exemple d'une plate-forme robotique mobile conçue pour le transport de bacs industriels au sein d'un atelier flexible ou *FMS* (*Flexible Manufacturing System*). Cette plate-forme est chargée d'acheminer des matières d'un poste de travail à un autre. Les postes de travail sont localisés via des balises et repérées dans l'espace grâce à un système de vision embarqué sur le robot mobile. Supposons à présent que le système détecte un obstacle sur sa route, entraînant l'activation de la fonction associée, c'est-à-dire un arrêt d'urgence de la plate-forme. Dans ce cas et pour des raisons de sécurité, la requête apériodique associée doit être exécutée le plus tôt possible, même si elle provoque une surcharge temporaire de traitements. Le problème souligné ici est alors le suivant : comment le système peut-il fournir un niveau de QoS satisfaisant sous des conditions de surcharge ? En fait, nous devons considérer dans ce cas qu'un certain degré d'échéances non respectées peut être toléré au niveau des activités périodiques du système, sans pour autant compromettre le comportement global du système temps réel. Considérons l'exemple de la fonction *Vision*. Si celle-ci n'a pas assez de temps pour décoder une image, alors le système peut très bien sauter l'instance de tâche correspondante ou afficher une image partielle. En conséquence, nous observerons une QoS moindre sans souffrir d'une dégradation significative de la performance globale du système.

A partir de l'analyse menée ci-dessus, nous observons que la théorie du temps réel *dur* apparaît trop restrictive et inadaptée. Par conséquent, une approche plus fine consiste à apposer des contraintes de pertes à toutes les tâches périodiques (garantie d'une QoS minimale) et à ordonnancer les requêtes apériodiques au plus tôt grâce à un serveur adapté.

Le travail de thèse présenté dans ce rapport a pour finalité de proposer des solutions à l'ordonnancement dans les systèmes informatiques temps réel à contraintes fermes (TRCF). Le système est supposé assurer l'ordonnancement de tâches périodiques définies sous des contraintes de QoS. Il doit aussi être capable de gérer l'occurrence de tâches apériodiques pouvant induire une surcharge temporaire de traitement.

Le chapitre 1 constitue une introduction à l'ordonnancement dans les systèmes temps réel, en rappelant les concepts de base liés à cette problématique.

Le chapitre 2 est consacré à une synthèse bibliographique sur l'ordonnancement en présence de surcharge. Les principaux schémas d'ordonnancement et modèles de résolution dédiés aux systèmes surchargés, sont présentés.

Dans le chapitre 3, deux stratégies d'ordonnancement de tâches périodiques définies sous des contraintes de QoS utilisant le modèle Skip-Over, sont proposées. Le modèle Skip-Over est une technique qui consiste à gérer deux types d'instances pour chaque tâche : des instances *rouges* qui doivent obligatoirement s'exécuter dans le respect de leur échéance, et des instances *bleues* qui peuvent être abandonnées à tout moment. Une évaluation des performances des stratégies proposées est présentée.

Le but du chapitre 4 est d'étudier l'ordonnancement d'un ensemble hybride de tâches constitué de tâches périodiques définies sous des contraintes de QoS et de tâches apériodiques. La contribution des travaux de thèse se focalise sur l'utilisation du serveur optimal EDL (Earliest Deadline as Late as possible) avec des tâches périodiques ordonnancées selon les stratégies introduites dans le chapitre 3. Une étude des performances des différents modèles d'ordonnancement envisagés est rapportée.

Dans le chapitre 5, nous nous intéressons à deux critères d'évaluation des stratégies d'ordonnancement en présence de surcharge, la stabilité et la robustesse. S'appuyant sur des travaux précédents relatifs à la tolérance aux fautes, nous présentons deux nouveaux algorithmes d'ordonnancement pour le modèle Skip-Over, qui améliorent le comportement d'un système avec des contraintes de QoS.

Le chapitre 6 présente l'intégration sous Linux temps réel des différentes stratégies d'ordonnancement sous contraintes de QoS étudiées et proposées dans le cadre de la thèse. A la fin de ce chapitre, une évaluation des composants d'ordonnancement intégrés, est exposée.

En conclusion, nous décrivons les applications de notre travail et présentons ses axes de développements futurs.

Chapitre I

Introduction à l'ordonnancement dans les systèmes temps réel

Ce premier chapitre constitue une introduction aux systèmes informatiques temps réel. Dans un premier temps, nous proposons une présentation de la terminologie utilisée. Ensuite, les concepts de base de l'ordonnancement temps réel ainsi que les principaux résultats scientifiques liés à cette problématique (typologie, propriétés...) sont rappelés. Puis, nous nous attachons plus précisément à l'état de l'art relatif à l'ordonnancement de tâches périodiques d'une part et de tâches apériodiques d'autre part.

---

1 Le temps réel : terminologie et modèles

1.1 Le concept de temps réel

Le qualificatif de *temps réel* est lié à la capacité du système informatique à réagir en-ligne et dans le respect de contraintes temporelles, à l'occurrence d'événements issus de l'environnement extérieur. La terminologie *temps réel* est directement liée à l'aptitude du système à présenter des temps de réponse (appelés aussi temps de réaction) en adéquation avec les dynamiques de l'environnement contrôlé. Ceux-ci doivent donc être suffisamment petits vis-à-vis de la vitesse d'évolution des sollicitations émises par l'environnement extérieur.

La définition du *temps réel* largement adoptée dans le domaine est celle de Stankovic [Sta88] : *"la correction du système ne dépend pas seulement des résultats logiques des traitements, mais dépend en plus de la date à laquelle ces résultats sont produits"*. Le système doit ainsi être capable de réagir suffisamment rapidement pour que la réaction ait un sens. Par conséquent, une application temps réel implique généralement des activités auxquelles sont associées des contraintes temporelles. Parmi les plus courantes, on peut citer par exemple les échéances temporelles strictes des traitements.

1.2 Les spécificités des systèmes temps réel

1.2.1 Définitions

Un système temps réel est un système qui d'une part, interagit avec un environnement externe en évolution dans le temps, et qui d'autre part, réalise certaines fonctionnalités en relation avec cet environnement, et ceci en exploitant le plus souvent des ressources limitées. Un système temps réel se caractérise par sa nature intégrée : il constitue une composante d'un système plus imposant doté d'interactions avec le monde physique. C'est souvent là que réside la principale source de complexité des systèmes en temps réel. Le monde physique se comporte généralement de manière *non-déterministe* : les événements se produisent sans synchronisme, concurremment, dans un ordre imprévisible. Dans ce contexte, deux contraintes sont pourtant à vérifier pour qu'un système temps réel soit fonctionnel : l'*exactitude logique (logical correctness)* et l'*exactitude temporelle (timeliness)*. En effet, le système doit non seulement fournir des sorties adéquates en fonction des entrées mais aussi produire les résultats sur les sorties au bon moment. Le *temps de réponse* (appelé aussi *temps de réaction*) d'un système en temps réel est la durée entre la présentation des entrées à un système et l'apparition des sorties suite aux traitements effectués par ce système sur ces entrées. La spécification du temps de réponse est ainsi lié à l'échelle de temps d'évolution de l'environnement.

On peut résumer le comportement que doit adopter un système temps réel à travers trois caractéristiques fondamentales qui le définissent complètement : *prévisibilité*, *déterminisme* et *fiabilité* [BD99]. En effet, les activités doivent être prévues et exécutées dans les contraintes de temps spécifiées. Pour garantir cela, le concepteur de systèmes temps réel doit toujours se placer dans le *pire-cas*. Le déterminisme consiste à écarter toute incertitude sur le comportement des activités individuelles, y compris lorsqu'elles doivent interagir. Parmi les sources de non-déterminisme, on peut citer la charge de calcul, les entrées-sorties, les interruptions, etc. Concernant la contrainte de fiabilité, les composants matériels et logiciels se doivent d'être fiables dans un contexte temps réel. C'est pourquoi, généralement, les systèmes temps réel sont conçus de façon à être tolérants aux fautes (*fault-tolerant*).

1.2.2 Classification traditionnelle

La classification traditionnelle des systèmes temps réel repose par ailleurs sur trois classes caractérisant les conditions temps réel apposées sur ceux-ci. En effet, on distingue classiquement différents *niveaux de contraintes temporelles* décrites ci-après.

On appelle *systèmes temps réel à contraintes strictes (TRCS)* [Ser72, LL73, Bla76], les systèmes pour lesquels tous les traitements doivent impérativement respecter toutes leurs contraintes temporelles en condition nominale de fonctionnement. Ces systèmes doivent être prédictibles [Law83] au niveau de leurs performances logiques et temporelles. L'incapacité du système à satisfaire les contraintes temporelles provoque la faute du système et entraîne souvent des conséquences catastrophiques sur l'environnement contrôlé. Une seule faute temporelle peut alors avoir un coût intolérable en termes de vies humaines, de dommages matériels ou de pertes économiques. Ces systèmes informa-

tiques impliquant des traitements temps réel strict ou dur (*hard* en anglais) sont surtout présents dans les domaines de l'aéronautique, de l'aérospatiale, de la robotique, de la supervision de centrales chimiques ou nucléaires, etc.

Les *systèmes temps réel à contraintes relatives (TRCR)* [Hor74, LLR76, GLL+79], par opposition, tolèrent les fautes temporelles (non-respect transitoire des contraintes) ; on parle alors de traitements temps réel souples (*soft* en anglais). La performance du système est dégradée sans engendrer de conséquences dramatiques sur l'environnement et sans remettre en cause l'intégrité du système. Les dépassements d'échéances auront un certain coût pour le système qui se traduira par exemple par une précision de calcul moindre, une cadence de rafraîchissement de données plus faible, etc.

A la frontière entre les deux niveaux de contraintes précédents, on rencontre les *systèmes temps réel à contraintes fermes (TRCF)* [BBL01] dans lesquels les traitements temps réel, dits fermes (*firm* en anglais) reposent sur des contraintes strictes mais où une faible probabilité de manquer les limites temporelles peut être tolérée. La mesure du respect des contraintes temporelles peut alors prendre la forme d'une donnée probabiliste : la *Qualité de Service (QoS)*. Celle-ci est directement liée à un service offert par le système et/ou au comportement du système dans son ensemble. Les violations d'échéances autorisées entraînent uniquement des dégradations de QoS [AB98]. La performance du système est localement ou globalement dégradée tout en le conservant dans un état *sécuritaire*. Dans le contexte des contraintes *fermes*, il apparaît important de définir les besoins en termes de QoS d'un système, afin de délivrer un service à la hauteur des besoins décrits du point de vue de l'utilisateur. Ces nouveaux types d'applications affichant des contraintes temps réel *fermes* sont pour la plupart des applications émergentes dans les domaines du multimédia, du contrôle automatique, et de la télésurveillance.

Considérons par exemple un système d'antiblocage des roues [HR97] dans lequel typiquement, une tâche temps réel détermine le début du freinage en scrutant périodiquement la vitesse de rotation échantillonnée de chaque roue. Dans un tel système, on se rend bien compte qu'il n'est pas nécessaire que toutes les instances de la tâche s'exécutent dans le respect de leurs échéances mais plutôt dans certaines limites de pertes d'échéances bien spécifiées. Il faut en particulier veiller à limiter le nombre d'instances consécutives qui violent leurs échéances. Une situation similaire peut être rencontrée dans une application multimédia où le rejet de certaines instances de tâches de transmission de paquets de pixels par exemple, se traduira par une détérioration de l'image au niveau du processus de reconstruction. Là encore, on s'aperçoit de la nécessité de contrôler et d'apposer des contraintes sur le nombre de paquets à transmettre (vis-à-vis d'une période temporelle ou du nombre de requêtes de la tâche).

1.2.3 La notion de la Qualité de Service (QoS)

Le terme de *qualité de service* est employé pour qualifier un large spectre de domaines de l'informatique. En effet, on rencontre de plus en plus cette notion dans l'étude des systèmes temps réel critiques, des systèmes distribués ou encore dans le domaine des applications multimédia ou des réseaux de communication. Si l'on reprend la définition

donnée par l'ISO[1] et l'ITU-T[2], la *qualité de service* représente un *"ensemble d'exigences de qualité dans le comportement collectif d'un ou plusieurs objets"* [ISO95]. Il s'agit là d'une mesure collective du niveau de service fourni au client.

Les besoins de clients, exprimés en termes de *besoins de QoS*, peuvent être liés à une ou plusieurs caractéristiques, ces caractéristiques étant définies selon l'ISO et l'ITU-T de la manière suivante : *"les caractéristiques de QoS représentent un aspect de la QoS d'un système, service ou ressource, qui peut être quantifié et qui est défini indépendamment de la façon dont il est représenté ou contrôlé"* [ISO98].

Dans le cadre des applications multimédia, le terme de *qualité de service* désigne en général la résolution d'affichage d'une image ou d'une animation, la qualité d'un échantillon sonore, le taux de compression, etc. Pour les systèmes répartis, on entend en général par *qualité de service*, la capacité du système à acheminer des informations d'une entité du système à une autre dans les délais attendus et avec un taux d'erreurs acceptable. La QoS s'apparente également dans le cas d'un réseau de communication à un certain nombre de critères de performance de base tels que la disponibilité, le temps de réponse, le débit de données et la rapidité de détection et de correction de fautes.

Dans les systèmes temps réel, le terme de *qualité de service* tend à désigner la capacité du système à respecter des contraintes temporelles plus ou moins fortes. On retrouvera ainsi des caractéristiques telles que la prédictibilité, le temps de réponse, le taux de requêtes ayant respecté leurs échéances, le taux d'acceptation de requêtes apériodiques critiques, etc.

1.3 La caractérisation et la modélisation des tâches temps réel

1.3.1 Définitions

On appelle *tâche* une entité logicielle réalisant une fonction particulière au sein d'un logiciel d'application. Elle correspond aux exécutions d'une séquence d'opérations donnée sur le processeur. Cette séquence d'opérations relative au service fourni par la tâche peut être réexécutée plusieurs fois. Chacune des exécutions est alors considérée individuellement sous forme d'*instances* (ou *travaux*).

Dans un environnement multitâche, les tâches peuvent être dans l'un des quatre états suivants [DP91] : *courant* (executing), *prêt* (ready), *en attente* (suspended) ou *endormi* (dormant). Les trois premiers états sont considérés comme des états actifs (les tâches existent et assurent un service particulier pour l'application), le dernier étant un état inactif (les tâches n'existent pas ou plus du point de vue de l'application). Le passage d'un état à l'autre se fait sur décision de l'ordonnanceur. Dans la pratique, un changement d'état aura souvent pour conséquence un *changement de contexte*. La figure I.1 illustre les états actifs ainsi que les transitions d'un état vers un autre.

Une tâche à l'état *courant* a le contrôle du processeur et exécute son code. La tâche en exécution est celle qui, à un instant donné, est considérée comme la plus prioritaire parmi les candidates à l'attribution du processeur. Une tâche *en attente* n'est pas candidate

[1] International Organization for Standardization
[2] International Telecommunication Union - Telecom

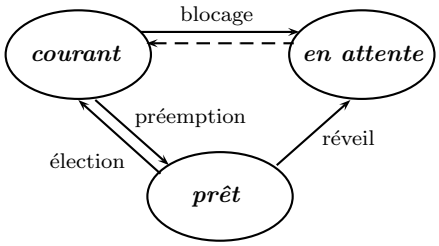

FIG. I.1 – Diagrammes des états actifs d'une tâche

à l'attribution du processeur. Son exécution est temporairement suspendue jusqu'à ce qu'elle obtienne la ressource qui lui manque (en plus du processeur) pour s'exécuter. Une tâche *prête* attend d'être élue pour pouvoir s'exécuter. Une tâche *endormie* est, soit non encore créée, soit terminée définitivement.

La dénomination de *tâche temps réel* se réfère à une tâche à date critique, c'est-à-dire à une tâche dont la fin d'exécution au plus tard est bornée. L'ensemble des tâches à exécuter sur une machine donnée est appelée *configuration de tâches*. Lorsque toutes les tâches de cette configuration sont caractérisées par leur échéance, on parlera de configuration de tâches critiques (ou à dates critiques).

D'autre contraintes (autres que temporelles) peuvent bien sûr être associées à l'exécution de tâches temps réel. Parmi celles-ci, nous pouvons citer [Sil93] :

- *les contraintes de ressources*, c'est-à-dire celles dérivées de l'accès en exclusion mutuelle aux ressources critiques. Les ressources requises par une tâche au cours de son exécution ne sont pas toujours supposées disponibles dès que cette tâche demande à s'exécuter.
- *les contraintes de synchronisation* qui peuvent être décrites par un ensemble de relations de précédence (ou d'antériorité) qui déterminent l'ordre dans lequel les tâches doivent subir leur traitement. Lorsqu'il n'existe aucune relation de précédence entre les tâches, on parle alors de tâches indépendantes.
- *les contraintes d'exécution* qui reposent sur deux modes d'exécution des tâches, respectivement qualifiés de *préemptif* et *non-préemptif*. Lorsque toute tâche est préemptible, cela signifie que son exécution peut être interrompue à un instant quelconque et être reprise ultérieurement (contrairement au cas non-préemptif dans lequel la tâche se réserve l'accès au processeur du début à la fin de son exécution).
- *les contraintes de placement* qui portent sur l'identité du (ou des) processeurs d'un système distribué sur lequel une tâche est autorisée à s'exécuter.

Deux types de tâches sont par ailleurs communément recensés : les tâches périodiques et les tâches apériodiques. Le modèle canonique de chacune de ces tâches est décrit ci-dessous.

1.3.2 Modélisation des tâches périodiques

Une tâche temps réel périodique $T_i(r_{i,k}, C_i, D_i, P_i)$ est définie par [CDK+00] :

- sa date de réveil $r_{i,k}$,
- sa période d'activation P_i,
- sa durée d'exécution maximale ou au pire-cas C_i,
- son délai critique D_i,
- son échéance absolue $d_{i,k} = r_{i,k} + D_i$.

La date de réveil $r_{i,k}$ correspond à la date de début d'exécution au plus tôt de la $k^{ième}$ requête de la tâche. La durée d'exécution au pire-cas C_i est l'intervalle de temps maximal séparant le début de la fin d'exécution (sans interruption) sur un processeur référencé. Le délai critique D_i (ou *échéance relative*) figure l'intervalle de temps séparant la date de réveil de l'échéance. Enfin, l'échéance absolue $d_{i,k}$ (aussi appelée date critique) correspond à la date de fin d'exécution au plus tard de la $k^{ième}$ requête de la tâche.

La figure I.2 représente deux requêtes d'une tâche périodique. Le temps est reporté sur l'axe des abscisses tandis que l'axe des ordonnées permet d'indiquer si la tâche est active (état haut) ou bien inactive (état bas).

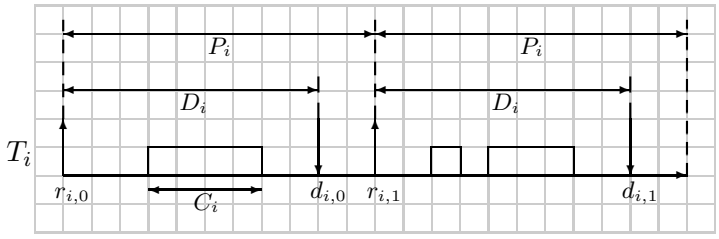

FIG. I.2 – Modèle d'une tâche périodique T_i

La $n^{ième}$ requête d'une tâche T_i se réveille à l'instant $r_{i,0} + (n-1)P_i$ et doit se terminer avant l'instant $r_{i,0} + (n-1)P_i + D_i$. Une tâche périodique est dite à *échéance sur requête (ER)* si son délai critique est égal à sa période d'activation (i.e $D_i = P_i$).

Ce que nous dénommons ici *tâche périodique* est parfois appelé *tâche sporadique* (un terme introduit par Mok [Mok83], même si le concept était connu auparavant) dans la littérature. De notre point de vue, une *tâche sporadique* est une tâche dont les dates d'activation sont inconnues à l'initialisation et dont la période correspond à la *durée minimale* (et non plus exacte), qui sépare deux arrivées successives de travaux pour la tâche. Dans ce qui suit, nous ne considérerons pas le cas de tâches sporadiques. Nos travaux de thèse ayant été menés pour des tâches périodiques uniquement.

Par la suite, on notera $\mathcal{T} = \{T_i(r_i, C_i, D_i, P_i),\ i = 1 \text{ à } n\}$ une configuration de tâches périodiques, avec n le nombre de tâches de la configuration $\mathcal{T}$. On dit qu'une configuration de tâches périodiques est à départ *synchrone* si toutes les tâches de cette configuration ont la même date de réveil initiale, donc si $\forall i, j \in \{1...n\},\ r_{i,0} = r_{j,0}$. Sinon, elle est dite *asynchrone*.

1.3.3 Modélisation des tâches apériodiques

Une tâche apériodique (ou tâche non-périodique) demande à s'exécuter une seule fois. L'activation d'une telle tâche a lieu lors de l'occurrence d'un événement qui peut être soit externe lorsqu'il est émis par l'environnement, soit interne lorsqu'il provient d'une autre tâche. Dans cette famille de tâches, on distingue généralement les tâches apériodiques *non critiques* (tâches ne possédant pas de date d'échéance et dont l'exécution doit s'effectuer au plus tôt), des tâches apériodiques *critiques* (tâches pourvues d'une date d'échéance et dont l'exécution doit s'effectuer dans le respect de cette contrainte temporelle).

1.3.3.1 Les tâches apériodiques non critiques. Les contraintes temporelles d'une tâche apériodique *non critique* que l'on notera R_i sont spécifiées par les paramètres suivants :

- une date d'arrivée a_i : date à laquelle R_i se fait connaître du système,
- une date de réveil r_i : date à laquelle R_i peut commencer son exécution,
- une durée d'exécution au pire-cas C_i : durée maximale d'exécution de R_i sur un processeur de référence.

Dans la présente étude, on supposera que la tâche apériodique R_i peut commencer son exécution dès sa date d'arrivée (i.e $r_i = a_i$). Par conséquent, on désignera par $R_i(r_i, C_i)$, la requête apériodique non critique R_i de date de réveil r_i et de durée d'exécution C_i. La figure I.3 illustre la modélisation d'une telle requête.

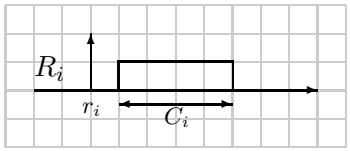

FIG. I.3 – Modèle d'une tâche apériodique non critique R_i

Par la suite, on notera $\mathcal{R} = \{R_i(r_i, C_i),\ i = 1 \text{ à } m\}$ une configuration de tâches apériodiques non critiques, avec m le nombre de tâches de la configuration $\mathcal{R}$.

1.3.3.2 Les tâches apériodiques critiques. Les contraintes temporelles d'une tâche apériodique *critique* que l'on notera S_i sont spécifiées par les paramètres suivants :

- une date d'arrivée a_i : date à laquelle S_i se fait connaître du système,
- une date de réveil r_i : date à laquelle S_i peut commencer son exécution,
- une durée d'exécution au pire-cas C_i : durée maximale d'exécution de S_i sur un processeur de référence,
- une échéance d_i : date avant laquelle S_i doit terminer son exécution où l'on suppose que $a_i \leq r_i$ et $r_i + C_i \leq d_i$.

Dans la présente étude, on supposera que la tâche apériodique S_i peut commencer son exécution dès sa date d'arrivée (i.e $r_i = a_i$). Par conséquent, on désignera par $S_i(r_i, C_i, d_i)$, la requête apériodique critique S_i de date de réveil r_i, de durée d'exécution C_i et d'échéance d_i. La figure I.4 illustre la modélisation d'une telle requête.

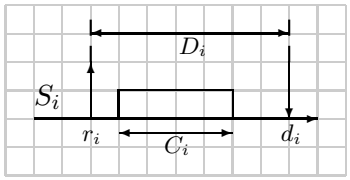

Fig. I.4 – Modèle d'une tâche apériodique critique S_i

Par la suite, on notera $\mathcal{S} = \{S_i(r_i, C_i, d_i),\ i = 1 \text{ à } p\}$ une configuration de tâches apériodiques critiques, avec p le nombre de tâches de la configuration $\mathcal{S}$.

Dans ce qui suit, nous introduisons la problématique de l'ordonnancement en temps réel en considérant les modèles de tâches décrits précédemment.

1.4 Problématique de l'ordonnancement monoprocesseur

Un problème d'ordonnancement de tâches temps réel consiste à résoudre les conflits d'accès au processeur entre les tâches suite à des demandes concurrentes d'exécution, en tenant compte de l'urgence relative des tâches. Cette fonction d'ordonnancement est assurée par une procédure de service du système d'exploitation (l'*ordonnanceur*), qui a pour rôle d'allouer le processeur aux différentes tâches. A partir des contraintes de ressources, de synchronisation, d'exécution, et des contraintes temporelles, et selon les critères de performance de l'application, l'ordonnanceur doit déterminer à tout instant l'identité de la tâche à exécuter. Un ordonnanceur peut mettre en œuvre un ou plusieurs *algorithmes d'ordonnancement* qui spécifient une politique d'allocation du processeur aux tâches. La planification temporelle des tâches construite par un algorithme d'ordonnancement est appelée *séquence* produite par cet algorithme pour la configuration de tâches considérée.

1.4.1 La typologie des algorithmes d'ordonnancement

On classe habituellement les algorithmes d'ordonnancement selon les caractéristiques du système sur lesquels ils sont implantés. Dans cette optique, le découpage suivant est proposé :

– **Monoprocesseur ou multiprocesseur**

  Lorsque toutes les tâches ne peuvent s'exécuter que sur un seul et même processeur, l'ordonnancement est qualifié de *monoprocesseur*. Dans le cas contraire, on parle d'ordonnancement *multiprocesseur* pour signifier le fait que plusieurs processeurs sont disponibles dans le système.

– **Oisif ou non oisif**

Avec un ordonnancement *non oisif*, lorsqu'une tâche est prête, elle ne peut se trouver en attente un seul instant avant d'être élue si la ressource processeur est libre. On dit alors que l'ordonnanceur fonctionne sans insertion de temps creux. Avec un ordonnancement *oisif*, lorsqu'une tâche est prête, elle peut, soit être élue, soit attendre un certain temps avant d'être activée, même si la ressource processeur est libre.
L'ordonnancement oisif a un intérêt particulier dans le cas non-préemptif. En effet, certaines configurations de tâches non ordonnançables en non-préemptif non-oisif le sont en non-préemptif oisif [Dec02].

– **Optimal ou non optimal**

Par définition, un algorithme d'ordonnancement est dit *optimal* [GRS96] dans une classe de problèmes d'ordonnancement donnée, s'il peut correctement ordonnancer un ensemble de tâches chaque fois que cet ensemble est ordonnançable. Par conséquent, si un ensemble n'est pas ordonnançable par un algorithme optimal d'une classe donnée, alors il ne peut l'être par aucun algorithme de la même classe.

– **En-ligne ou hors-ligne**

La méthode la plus simple consiste à planifier l'ordonnancement *hors-ligne*. Au moment de l'exécution, les tâches sont exécutées en fonction de cet ordonnancement. L'ordonnancement hors-ligne convient tout particulièrement quand les activités sont majoritairement appelées périodiquement. Néanmoins, en règle générale, ceci est peu applicable au contexte temps réel. L'ordonnancement statique cyclique ne convient pas lorsque les événements sont susceptibles de se produire de manière apériodique, il est alors préférable de se tourner vers des ordonnanceurs en-ligne. L'ordonnancement *en-ligne* doit toujours être assorti d'une analyse hors-ligne ou d'une méthode analytique, afin d'assurer le respect de la contrainte de temps à l'exécution. Une solide théorie fondée sur l'ordonnancement par préemption des priorités et l'analyse déterministe permet de prédire le temps de réponse aux événements dans la pire éventualité, compte tenu de la cadence maximale de leur arrivée, de la durée d'exécution maximale et de l'ordre de priorité des tâches.

– **Préemptif ou non préemptif**

Une condition nécessaire pour qu'un ordonnanceur soit *préemptif* est que toutes les tâches soient préemptibles. Dans ce cas, lorsqu'une tâche est jugée plus urgente ou plus prioritaire, elle peut gagner la ressource processeur et ce, au détriment d'une autre tâche qui voit la fin de son exécution interrompue et reprise plus tard. On remarquera que dans le cas d'ordonnanceurs monoprocesseur non préemptifs,

la problématique de gestion des ressources critiques n'a plus lieu d'être, puisqu'il ne peut pas y avoir d'accès concurrent aux ressources en l'absence de préemption.

– **Centralisé ou réparti**

L'ordonnancement est *réparti* lorsque les décisions d'ordonnancement sont prises par un algorithme localement en chaque noeud. Il est *centralisé* lorsque l'algorithme d'ordonnancement pour tout le système, distribué ou non, est déroulé sur un noeud privilégié.

1.4.2 Propriétés des algorithmes d'ordonnancement

Nous présentons ici une description des propriétés des algorithmes d'ordonnancement présentées en termes d'analyse d'*ordonnançabilité* et de *faisabilité* :

Définition 1 *Une* ***séquence d'ordonnancement valide*** *délivrée par un algorithme d'ordonnancement pour une configuration de tâches donnée, est une séquence pour laquelle toutes les contraintes des différentes tâches considérées sont respectées.*

Définition 2 *Une* ***configuration de tâches ordonnançable*** *ou faisable est une configuration pour laquelle il existe au moins un arrangement entre les tâches tel que les échéances soient respectées.*

Définition 3 *Etant donné une configuration de tâches ordonnançable, un* ***algorithme optimal*** *est un algorithme qui, par définition, est toujours capable de trouver une séquence valide (s'il en existe une).*

Définition 4 *Un* ***test d'ordonnançabilité*** *détermine, avant exécution, si oui ou non les échéances d'une configuration de tâches donnée seront toujours respectées.*

1.4.3 Complexité des algorithmes d'ordonnancement

Associée aux métriques précédemment décrites, l'efficacité d'un algorithme est également évaluée en fonction de sa *complexité* calculatoire [AHU74, HS76]. De façon générale, celle-ci est calculée en évaluant le nombre d'opérations élémentaires mises en œuvre, c'est-à-dire le nombre d'instructions de base de tout langage de programmation (addition, soustraction, affectation, test, ...) et ce, en fonction du nombre de données d'entrée du problème.

Soit f la fonction de complexité d'un algorithme représentant le plus grand nombre d'opérations élémentaires que cet algorithme requiert pour résoudre un problème Π, et n la taille du problème Π, c'est-à-dire la quantité de données d'entrée requises pour décrire Π. Ceci nous amène aux définitions suivantes [Bou91] :

Définition 5 *Un algorithme est dit en temps* ***polynomial*** *si sa fonction de complexité f est en $O(p(n))$ où p est un polynôme. Lorsque p est linéaire, l'algorithme est dit de complexité linéaire*

Définition 6 *Un algorithme est dit en temps* ***pseudo-polynomial*** *si sa fonction de complexité f possède la forme d'une fonction polynomiale mais n'est pas à la taille du problème. Le temps d'exécution dépend alors non seulement de la taille des entrées du problème mais également de la grandeur de celles-ci.*

Définition 7 *Un algorithme est dit en temps* ***exponentiel*** *si sa fonction de complexité f est en $O(n!)$ ou $O(k^n)$, $k > 1$ ou encore $O(n^{\log n})$.*

Les algorithmes polynomiaux sont bien sûr les plus intéressants car ils conduisent à la solution d'un problème d'ordonnancement dans un temps considéré comme raisonnable, au contraire des algorithmes de complexité exponentielle. Notons de plus que, plus la complexité algorithmique d'un ordonnanceur est élevée, plus son overhead d'implémentation sera important.

2 L'ordonnancement de tâches périodiques

Les théories classiques en ordonnancement portent sur l'ordonnancement *statique*. Ce dernier se réfère au fait que l'algorithme d'ordonnancement possède une connaissance complète de l'ensemble des tâches et de leurs contraintes (échéances, durées d'exécution, contraintes de précédences, instants de réveil futurs, etc.). L'algorithme d'ordonnancement produit alors une seule séquence d'ordonnancement sur cet ensemble de tâches, et celle-ci reste immuable durant toute la durée de vie de l'application associée.

En contrepartie, un algorithme d'ordonnancement *dynamique* (dans le contexte des travaux de thèse) possède une parfaite connaissance de l'ensemble courant des tâches actives, mais de nouvelles occurrences, non connues de l'algorithme au moment de l'ordonnancement de l'ensembles de tâches courant, peuvent se produire dans le futur et être prises en compte pour remettre en question l'ordonnancement précédemment construit. Par ailleurs, la majorité des ordonnanceurs temps réel dynamiques reposent sur la notion de *priorité*. Si la priorité est fixée à l'initialisation pour toutes les tâches, l'algorithme est dit *à priorités fixes*. Si elle évolue au cours du temps, l'algorithme est dit *à priorités dynamiques*.

L'étude reportée dans cette thèse porte sur l'ordonnancement dynamique, préemptif, non oisif, sans contraintes de ressources ni de précédences.

Nous présentons ci-après les algorithmes à priorités les plus couramment rencontrés. Pour chaque algorithme considéré, nous fournissons une description de son comportement assortie de ses performances et conditions d'ordonnançabilité qui lui sont associées.

2.1 L'ordonnancement à priorités fixes

2.1.1 L'algorithme RM

2.1.1.1 Description. L'algorithme d'ordonnancement le plus utilisé est l'algorithme Rate-Monotonic (RM). Selon cet algorithme, la priorité d'une tâche est inversement proportionnelle à sa période d'activation (les conflits étant résolus arbitrairement) [LL73].

En d'autres termes, une tâche est d'autant plus prioritaire que sa période d'activation P_i est petite. Considérons l'exemple ci-dessous de l'ordonnancement d'un ensemble de tâches $\mathcal{T} = \{T_i(C_i, D_i, P_i), i = 1..3\}$ par l'agorithme RM. Les tâches considérées sont toutes à échéances sur requêtes (i.e $D_i = P_i$). Elles sont également supposées synchrones avec une date de réveil initial $r_i = 0$.

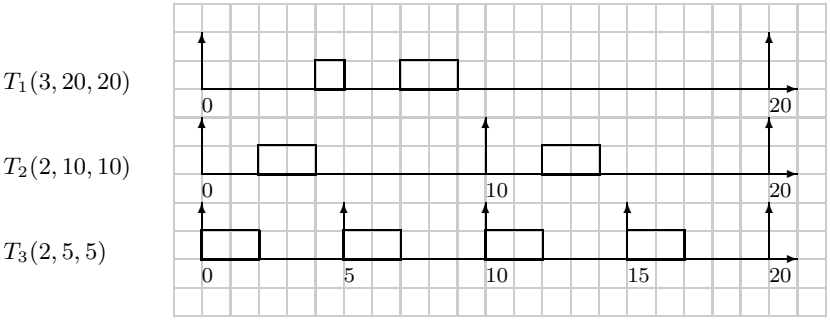

Fig. I.5 – Illustration du fonctionnement de RM

La tâche de plus petite période (T_3 dans l'exemple) est la plus prioritaire. Elle s'exécute dès le début de sa période d'exécution en interrompant si nécessaire des tâches moins prioritaires (T_1 est interrompue à l'instant $t = 5$ au profit de T_3). L'intérêt d'une telle stratégie d'ordonnancement repose en outre sur la simplicité de son implémentation où une seule structure de liste est nécessaire au fonctionnement de l'algorithme.

2.1.1.2 Résultats d'optimalité. RM est optimal dans la classe des algorithmes préemptifs à priorités fixes pour des tâches périodiques indépendantes à échéances sur requêtes [LL73]. Cela signifie que si une configuration de tâche $\mathcal{T}$ est ordonnançable par un quelconque algorithme préemptif à priorités fixes, alors elle est également ordonnançable par RM.

RM n'est plus optimal en contexte non-préemptif, et dès que l'hypothèse $\forall i = 1..n$, $D_i = P_i$ n'est plus vérifiée.

2.1.1.3 Conditions d'ordonnançabilité. Dans le cas de l'algorithme d'ordonnancement RM, une condition suffisante d'ordonnançabilité a été énoncée par Liu et Layland [LL73] :

Théorème 1 *Toute configuration de n tâches périodiques à échéances sur requêtes est ordonnançable par RM si son facteur d'utilisation U vérifie :*

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{I.1}$$

On peut alors affirmer que lorsque le nombre de tâches tend vers l'infini, le facteur d'utilisation minimal garantissant l'ordonnançabilité de la configuration tend vers $\log 2$, c'est-à-dire environ 69.3%. Notons cependant que pour RM, la borne d'ordonnançabilité est plus élevée lorsque les périodes des tâches présentent une relation harmonique, c'est-à-dire sont multiples les unes des autres.

Beaucoup de travaux ont été effectués dans le but d'améliorer la borne d'ordonnançabilité de l'algorithme RM et de relaxer quelques-unes des hypothèses restrictives sur l'ensemble de tâches considéré. Lehoczky *et al.* [LSD89] ont mené une étude statistique et montré que pour des ensembles de tâches dont les paramètres sont générés aléatoirement, l'algorithme RM est capable d'ordonnancer des ensemble de tâches avec un facteur d'utilisation du processeur de près de 88% (il ne s'agit cependant que d'un résultat statistique soulignant le pessimisme du test énoncé dans le Théorème 1). Par ailleurs, un test d'ordonnançabilité de même complexité que celui établi par Liu et Layland, connu sous le nom de *Hyperbolic Bound (HB)*, a été proposé par Bini *et al.* en 2001 [BBB01]. Le test établi, selon lequel *un ensemble de tâches périodiques est ordonnançable par RM si* $\prod_{i=1}^{n}(U_i+1) \leq 2$, permet une amélioration du taux d'acceptation de configurations d'un facteur maximum de $\sqrt{2}$. Dans le cas général, des tests d'ordonnaçabilité exacts conduisant à des conditions nécessaires et suffisantes ont été dérivées de façon indépendantes par Joseph et Pandya [JP86], Lehoczky *et al.* [LSD89] et Audsley *et al.* [ABR+93].

2.1.2 L'algorithme DM

2.1.2.1 Description. Dans un ordonnancement Deadline-Monotonic (DM), la priorité d'une tâche est inversement proportionnelle à son échéance relative (les conflits étant résolus arbitrairement) [LW82]. En d'autres termes, une tâche est d'autant plus prioritaire que son délai critique D_i est petit. L'algorithme DM est illustré ci-dessous avec un ensemble de tâches $\mathcal{T} = \{T_i(C_i, D_i, P_i), i = 1..3\}$. Les tâches sont supposées synchrones avec une date de réveil initial $r_i = 0$.

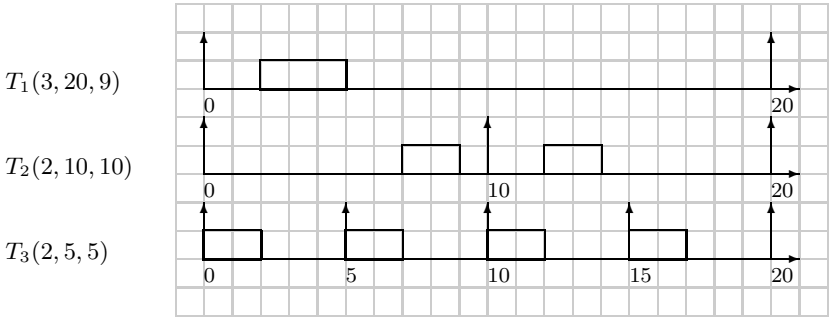

FIG. I.6 – Illustration du fonctionnement de DM

2.1.2.2 Résultats d'optimalité. DM est optimal dans la classe des algorithmes préemptifs non oisifs à priorités fixes pour des tâches périodiques indépendantes telles que $D_i \leq P_i$ [LW82, ABR+91, But97]. Cela signifie que, sous ces hypothèses, si une configuration de tâche $\mathcal{T}$ est ordonnançable par un quelconque algorithme préemptif à priorités fixes, alors il est également ordonnançable par DM.

DM n'est optimal en contexte non-préemptif que si $\forall(i,j)$, $C_i \leq C_j \Rightarrow D_i \leq D_j$ [Bat98].

2.1.2.3 Conditions d'ordonnançabilité. Dans le cas de l'algorithme d'ordonnancement DM, la même condition d'ordonnançabilité que celle établie pour l'algorithme RM peut être énoncée :

Théorème 2 *Toute configuration de n tâches périodiques à échéances sur requêtes est ordonnançable par DM si son facteur d'utilisation U vérifie :*

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{I.2}$$

Sous DM, on observe donc aussi une borne supérieure ($\approx 69.3\%$) pour le facteur d'utilisation du processeur lorsque $n \to \infty$.

Des conditions de faisabilité relatives à l'algorithme DM, ont par ailleurs été développées par Audsley et Burns [AB90], dont un test de faisabilité *nécessaire et suffisant* reposant sur un algorithme de complexité dépendant des données (*data-dependent*). Ce test est capable de déterminer la faisabilité de tout ensemble de tâches à priorités fixes pour lesquelles $D_i \leq P_i$ et ce, quel que soit le critère d'assignation des priorités utilisé. Par la suite, Audley *et al.* [ABR+93] ont établi un test exact de complexité pseudo-polynomiale, basé sur la théorie de l'*analyse des temps de réponse* (*Response Time Analysis*), enoncé ci-après :

Théorème 3 *Un ensemble de tâches périodiques telles que $D_i \leq P_i$ est ordonnançable avec l'algorithme DM si et seulement si le temps de réponse au pire-cas de chaque tâche est inférieur ou égal à son échéance relative.*

Le temps de réponse au pire-cas d'une tâche T_i est alors calculé de la manière suivante :

$$T_{rep} = C_i + B_i + I_i \tag{I.3}$$

où C_i correspond à sa durée d'exécution au pire-cas, B_i représente son temps de blocage dans le pire-cas lié à l'attente d'une ressource partagée en cours d'utilisation par une ou plusieurs autres tâches, et I_i traduit l'interférence due aux autres tâches (c'est-à-dire le temps maximum durant lequel la tâche T_i peut être préemptée par des tâches de plus haute priorité).

2.2 L'ordonnancement à priorités dynamiques

2.2.1 L'algorithme EDF

2.2.1.1 Description. Par définition, l'algorithme Earliest Deadline First (EDF) donne la priorité, à chaque instant, à la tâche dont l'échéance absolue d_i est la plus proche [Jac55, Ser72, LL73]. En cas de conflits, la tâche dont la date de réveil est la plus ancienne peut être exécutée la première. L'algorithme EDF est illustré ci-dessous avec un ensemble de tâches $\mathcal{T} = \{T_i(C_i, D_i, P_i), i = 1..3\}$.

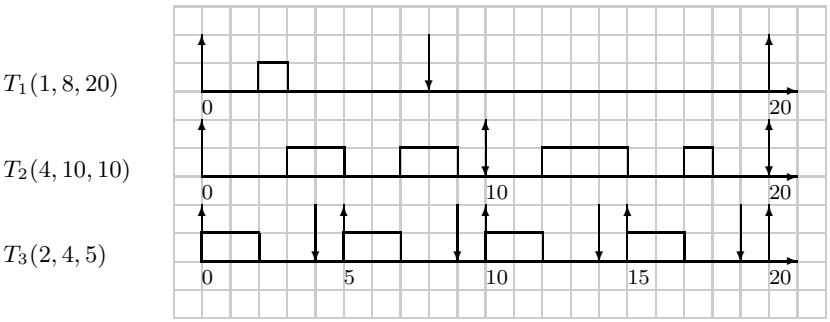

FIG. I.7 – Illustration du fonctionnement de EDF

2.2.1.2 Résultats d'optimalité. EDF est optimal dans la classe des algorithmes préemptifs pour des tâches périodiques indépendantes [Der74]. Si une configuration de tâches périodiques indépendantes est ordonnançable par un algorithme quelconque alors, elle l'est aussi par EDF.

Dans le cas non-préemptif, le problème d'ordonnancement est connu pour être NP-difficile [RRC05]. Cependant, si l'on considère uniquement les ordonnanceurs non oisifs, le problème est de nouveau soluble. Dans cette sous-classe des ordonnanceurs non-préemptifs, l'algorithme EDF est optimal comme ont pu le montré George *et al.* dans [GMR95].

2.2.1.3 Conditions d'ordonnançabilité. Un test d'ordonnançabilité reposant sur une condition *nécessaire et suffisante* [LL73], peut être rapporté concernant l'algorithme EDF :

Théorème 4 *Toute configuration de n tâches périodiques à échéances sur requêtes est ordonnançable par EDF si et seulement si son facteur d'utilisation U vérifie :*

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{I.4}$$

Toute configuration de tâches qui ne satisfait pas ce dernier test ne peut pas être ordonnancée par EDF, ni par aucun autre algorithme.

Sous EDF, l'analyse d'ordonnançabilité de tâches périodiques dont les échéances relatives sont inférieures ou égales aux périodes, peut aussi être menée en utilisant le *critère de demande du processeur* (*processor demand criterion*) proposé par Baruah *et al.* [BHR90] :

Théorème 5 *Un ensemble de tâches est ordonnançable par EDF si et seulement si :*

$$\forall L > 0, \sum_{i=1}^{n} \lfloor \frac{L + P_i - D_i}{P_i} \rfloor C_i \leq L \tag{I.5}$$

La complexité de ce test de faisabilité est pseudo-polynomiale. Cependant, dans le cas où toutes les tâches périodiques de l'ensemble considéré sont à échéances sur requêtes, les auteurs ont montré que l'analyse d'ordonnançabilité sous EDF se ramenait à une complexité en $O(n)$.

L'algorithme EDF est un algorithme très performant. Il permet en effet d'atteindre un facteur d'utilisation du processeur de 100% sous la conditions énoncée dans le théorème 4. Notons cependant que l'algorithme EDF est très instable en cas de surcharge comme nous le verrons dans le chapitre suivant.

2.2.2 L'algorithme LLF

2.2.2.1 Description. L'algorithme LLF (Least Laxity First) choisit d'exécuter en priorité la tâche possédant la *laxité dynamique* (*slack*) la plus faible, ce qui se traduit de la manière suivante : à chaque instant t, la tâche la plus prioritaire est celle dont la laxité du processeur à t, définie par $L(t) = r_i + D_i - (t + C_i(t))$, est la plus petite [MD78].

Cette quantité illustrée sur la figure I.8 représente le temps maximum où le processeur peut rester libre après t sans entraîner le manquement d'une échéance pour une tâche. Notons que $C_i(t)$ figure la durée d'exécution restante de la tâche à l'instant t. $\forall t$, on doit avoir $L(t) \geq 0$.

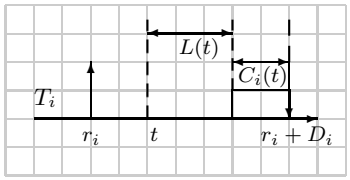

FIG. I.8 – Laxité du processeur à l'instant t

L'algorithme LLF est illustré ci-dessous avec un ensemble de tâches $\mathcal{T} = \{T_i(C_i, D_i, P_i), i = 1..3\}$. Les tâches sont supposées synchrones avec une date de réveil initial $r_i = 0$.

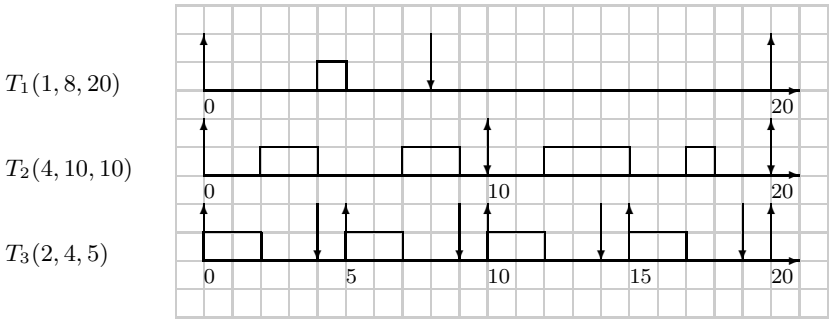

FIG. I.9 – Illustration du fonctionnement de LLF

Cette politique d'ordonnancement permet de maximiser le retard minimum de tout ensemble de tâches. En comparaison de l'algorithme EDF, LLF prend en compte non seulement l'urgence du travail à accomplir mais également la notion de durée du travail.

2.2.2.2 Résultats d'optimalité. Sorenson [Sor74] a démontré l'optimalité de cet algorithme : LLF est optimal dans la classe des algorithmes préemptifs pour des tâches périodiques indépendantes telles que $D_i \leq P_i$.

2.2.2.3 Conditions d'ordonnançabilité. La condition d'ordonnançabilité pour LLF est la même que pour EDF :

Théorème 6 *Toute configuration de n tâches périodiques à échéances sur requêtes est ordonnançable par LLF si et seulement si son facteur d'utilisation U vérifie :*

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{I.6}$$

LLF est un algorithme aussi performant que EDF à la différence près que l'algorithme LLF a l'inconvénient de générer potentiellement un grand nombre de préemptions, et qu'il n'est plus optimal si la préemption n'est pas autorisée [GRS96].

2.3 Résultats fondamentaux

L'un des résultats fondamentaux concernant l'ordonnancement préemptif est celui établi par Leung et Merril [LM80] :

Théorème 7 *La séquence produite par tout algorithme préemptif sur une configuration de tâches périodiques est toujours périodique de période égale au plus petit commum multiple (ppcm) des périodes des tâches de la configuration, noté P.*

Un tel résultat permet d'affirmer que si le processeur est inactif à un instant t donné, il le sera également à tout instant du type $t + kP$ ($k \in \mathbb{N}$). De même, s'il a travaillé x_i unités de temps sur une instance de T_i à l'instant t, il aura travaillé aussi x_i unités de temps sur une autre instance de T_i à l'instant $t + kP$.

Ce résultat permet également de ramener l'analyse d'ordonnançabilité dans une fenêtre de référence égale en durée au PPCM des périodes.

3 L'ordonnancement conjoint de tâches périodiques et de tâches apériodiques

Les algorithmes d'ordonnancement mis en œuvre pour répondre au problème de l'exécution *conjointe* de tâches périodiques et de tâches apériodiques, doivent non seulement être capables de garantir les échéances des tâches périodiques critiques mais également fournir des temps de réponses moyens acceptables pour les tâches apériodiques non critiques, même lorsque les occurrences des requêtes apériodiques ont un caractère non-déterministe. Dans le cas de tâches apériodiques critiques, l'objectif poursuivi est de maximiser le taux d'acceptation moyen de ces requêtes au sein du système.

Deux approches sont couramment rencontrées : les serveurs de tâches à priorités *fixes* et ceux à priorités *dynamiques*, en plus de l'approche traditionnelle qui consiste à servir les tâches apériodiques en arrière-plan. Ces approches sont successivement décrites dans les sections qui suivent.

3.1 L'approche basée sur un traitement en arrière-plan

3.1.1 Le serveur BG

3.1.1.1 Description. Le serveur de tâches apériodiques *Background* (*BG*) [LSS87] ordonnance les tâches apériodiques non critiques lorsqu'il n'y aucune activité périodique (c'est-à-dire, lorsqu'il n'y a aucune tâche périodique à l'état prêt) au sein du système. Le principal avantage de ce serveur réside dans sa simplicité de mise en œuvre, en plus du fait qu'il garantisse que les tâches apériodiques n'affecteront pas le comportement des tâches périodiques. Par contre, l'inconvénient majeur provient du fait que le temps de réponse aux requêtes apériodiques peut être prohibitif. En effet, si le taux d'utilisation du processeur alloué aux tâches périodiques est élevé, peu d'opportunités sont laissées aux tâches apériodiques.

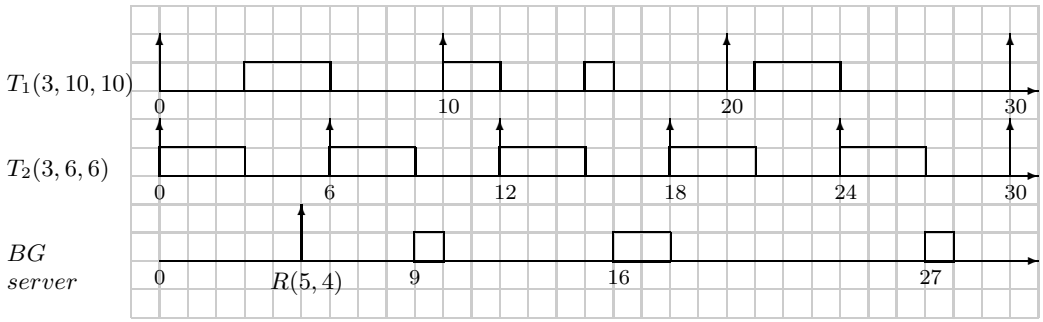

FIG. I.10 – Illustration du fonctionnement du serveur BG

Dans l'exemple illlustratif de la figure I.10, l'ensemble $\mathcal{T}$ dont les paramètres sont décrits dans la Table I.1, est ordonnancé au plus tôt selon l'algorithme EDF.

Task	T_1	T_2
C_i	3	3
P_i	10	6
D_i	10	6

TAB. I.1 – Un ensemble basique de tâches périodiques

Une requête apériodique R de durée 4 unités de temps, survient à l'instant $t = 5$. Le temps de réponse à la requête, via le service BG, s'élève dans ce cas à 23 unités de temps.

3.1.1.2 Cas de tâches apériodiques critiques. L'acceptation des tâches apériodiques sous BG nécessite la construction préliminaire de la fonction de disponibilité f^{EDS}. Celle-ci représente l'ensemble des temps creux disponibles lorsque les tâches périodiques sont exécutées au plus tôt. On appelle *temps creux* un intervalle de temps pendant lequel le processeur est inactif.

La séquence EDS (Earliest Deadline as Soon as possible) doit en effet être déterminée pour pouvoir calculer la somme des temps creux entre l'instant d'occurrence d'une tâche apériodique critique et son échéance, de manière à valider ou non l'acceptation de cette nouvelle tâche au sein du système. Le calcul de f^{EDS} pour la localisation et la détermination de la durée des temps creux, repose sur deux vecteurs : le *vecteur des dates de réveil* et le *vecteur des temps creux*. Le vecteur des dates de réveil représente l'ensemble des instants auxquels un temps creux est observable sur $[0, P[$, avec $P = ppcm(P_1, P_2, ..., P_n)$, l'hyperpériode égale au plus petit commun multiple des périodes des tâches. Le vecteur des temps creux est utilisé pour consigner la durée des temps creux associés à ces différents instants. Les équations pour la localisation et la détermination des temps creux EDS [CC90] sont décrites ci-après.

Calcul du vecteur statique des dates de réveil. Le vecteur statique $\mathcal{E} = (e_0, e_1, ..., e_i, e_{i+1}, ..., e_q)$, représente l'ensemble des instants considérés sur l'hyperpériode $[0, P]$, qui marque la fin d'un temps creux. Il se construit dans le cas général, à partir des dates de réveil distinctes des instances périodiques. Etant donné que les tâches considérées ici sont à échéances sur requêtes, nous nous trouvons dans le cas particulier où ce vecteur correspond également aux échéances distinctes des tâches. Par conséquent, les instants e_i sont définis par le théorème suivant [CC90] :

Théorème 8 *Les instants e_i du vecteur $\mathcal{E} = (e_0, e_1, ..., e_i, e_{i+1}, ..., e_q)$ calculés sur $[0, P[$ sont définis comme suit :*

$$e_i = x.P_j \tag{I.7}$$

avec $x = 1, ..., \frac{P}{P_j}$, $e_i < e_{i+1}$, $e_0 = \min\{P_j; 1 \leq j \leq n\}$ et $e_q = P$

Calcul du vecteur statique des temps creux. Le vecteur statique des temps creux $\mathcal{D} = (\Delta_0, \Delta_1, ..., \Delta_i, \Delta_{i+1}, ..., \Delta_q)$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des dates de réveil. Δ_i représente la longueur du temps creux se terminant au temps e_i. Le théorème suivant [CC90] fournit la formule de récurrence pour le calcul du vecteur $\mathcal{D}$:

Théorème 9 *Les durées des temps creux du vecteur $\mathcal{D} = (\Delta_0, \Delta_1, ..., \Delta_i, \Delta_{i+1}, ..., \Delta_q)$ calculées sur $[0, P[$ sont définies comme suit :*

$$\Delta_i = \sup(0, F_i) \quad \textit{pour } i = 0 \textit{ à } q \tag{I.8}$$

$$\textit{avec} \quad F_i = e_i - \sum_{j=1}^{n} (\lceil \frac{e_i}{P_j} \rceil) c_j - \sum_{k=0}^{i-1} \Delta_k$$

Illustration du calcul de f^{EDS}. Considérons l'ensemble de tâches $\mathcal{T} = \{T_1, T_2\}$ constitué de deux tâches périodiques, $T_1(3, 10, 10)$ et $T_2(3, 6, 6)$. En appliquant les formules I.7 et I.8, on obtient le vecteur des échéances $\mathcal{E} = (6, 10, 12, 18, 20, 24, 30)$ et le vecteur des temps creux $\mathcal{D} = (0, 1, 0, 2, 0, 0, 3)$. Le calcul des intervalles de temps creux fournit la séquence EDS représentée sur la figure I.11. A titre d'exemple, la somme des temps creux sur l'intervalle $[12, 30]$, est égale à $\Omega^{EDS}_{\mathcal{T}(12)}(12, 30) = 5$.

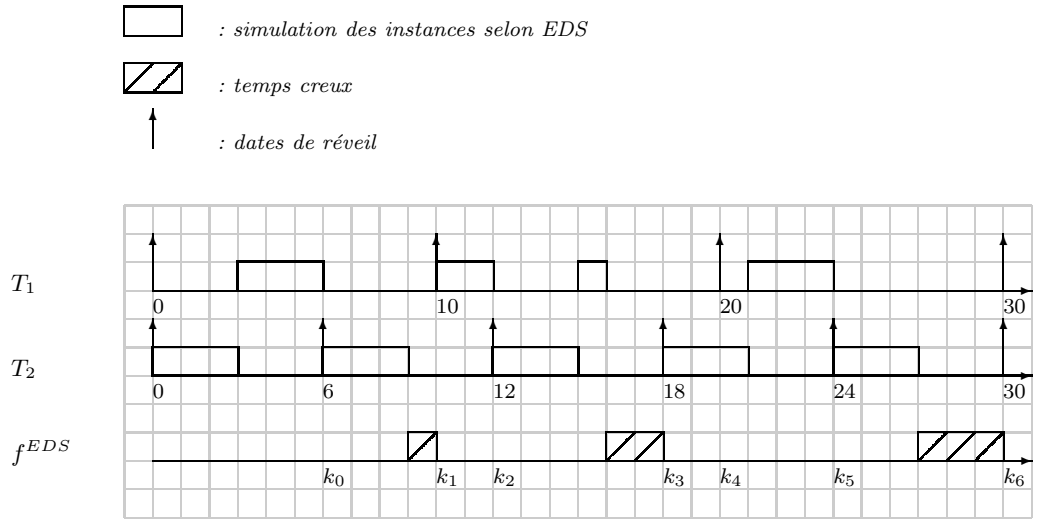

FIG. I.11 – Calcul des temps creux statiques selon EDS

Test d'acceptation des apériodiques critiques sous BG. Considérons à présent le problème de l'acceptation de tâches apériodiques à contraintes strictes. On suppose que ces tâches apériodiques critiques sont synchrones en ce sens que leur date d'arrivée correspond avec leur date de réveil. Soit τ le temps courant coïncidant avec l'arrivée d'une requête apériodique critique $R(\tau, c, d)$, où c représente sa durée d'exécution au pire cas et d son échéance.

On suppose que le système peut présenter plusieurs tâches apériodiques critiques à l'état prêt au temps τ correspondant aux différentes requêtes précédemment acceptées. Par conséquent, celles-ci n'ont pas terminé leur exécution à τ. On note $\mathcal{R}(\tau) = \{R_i(c_i(\tau), d_i), i{=}1 \text{ à } q(\tau)\}$ l'ensemble de tâches apériodiques critiques présentes dans le système à τ. $c_i(\tau)$ désigne la durée d'exécution dynamique de R_i (c'est-à-dire sa durée d'exécution restante) et d_i son échéance. L'ensemble $\mathcal{R}(\tau)$ est ordonné tel que $i < j$ implique $d_i \leq d_j$.

Dans [CS93], les auteurs proposent le test d'acceptation *optimal* suivant pour des tâches apériodiques critiques, lorsque les tâches périodiques sont ordonnancées au plus tôt selon EDS :

Théorème 10 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDS}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{I.9}$$

$\delta_i(\tau)$, appelée laxité de la tâche R_i au temps τ, représente la longueur de l'intervalle de temps qui sépare sa fin d'exécution au plus tôt de son échéance. $\Omega^{EDS}_{\mathcal{T}(\tau)}(\tau, d_i)$ représente la somme totale des temps creux calculée entre τ et d_i. La sommation $\sum_{j=1}^{i} c_j(\tau)$ traduit la somme des exécutions restantes à l'instant τ sur l'ensemble des tâches apériodiques critiques dont l'échéance est inférieure ou égale à d_i. Notons qu'une fois acceptées, les tâches apériodiques critiques sont ordonnancées au plus tôt selon l'algorithme EDF, conjointement avec les tâches périodiques.

3.2 Les approches basées sur un serveur à priorités fixes

3.2.1 Le serveur PS

La méthode du *Polling Serveur (PS)* [LSS87] consiste à créer une tâche périodique (C_s, P_s), dite *serveur*, de période P_s et de durée d'exécution C_s (aussi appelée *capacité du serveur*), afin d'exécuter les tâches apériodiques. Les tâches apériodiques en attente sont servies par le serveur selon la technique FIFO, et à des instants réguliers déterminés par les réveils périodiques du serveur. En cas d'absence de tâche apériodique en attente lors du réveil du serveur, la capacité de celui-ci est perdue.

Son fonctionnement est illustré sur la figure I.12 dans laquelle nous considérons un ensemble $\mathcal{T} = \{T_1(2, 6, 6), T_2(1, 4, 4)\}$ de deux tâches périodiques et un serveur $PS(2, 5)$ de capacité égale à 2 unités de temps et de période 5 unités de temps. Toutes les tâches périodiques (y compris le serveur) sont ordonnancées selon l'algorithme RM. Durant l'hyperpériode représentée, 4 requêtes apériodiques R_i surviennent respectivement aux instants $t = 2$, $t = 8$, $t = 12$ et $t = 19$. Pour éclairer le lecteur, nous figurons l'évolution de la capacité du serveur PS au cours du temps, capacité évoluant en fonction du service assuré vis-à-vis des requêtes apériodiques occurrentes.

Nous pouvons remarquer qu'à l'instant $t = 0$, étant donné qu'aucune tâche apériodique n'est en attente, le serveur se suspend jusqu'à l'occurrence de sa prochaine requête et le temps qui lui était alloué est perdu. Par conséquent, la requête $R_1(2, 2)$ n'est servie qu'à partir de l'instant $t = 5$.

Les performances obtenues en termes de temps de réponse observé au niveau des requêtes apériodiques, sont meilleures que celles obtenues avec la gestion des tâches apériodiques en tâches de fond. De plus, cette approche présente de faibles complexités de calcul et d'implémentation. Cependant, un des inconvénients de cette méthode réside dans le choix délicat des paramètres du serveur PS, de manière à ce que celui-ci préserve

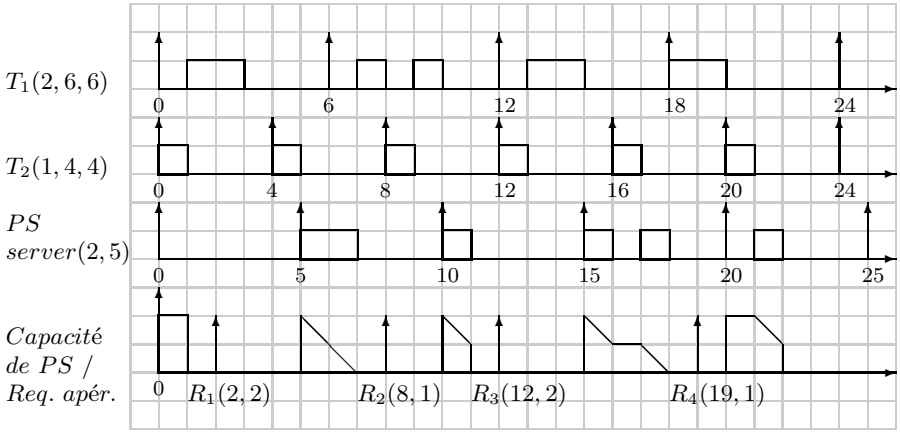

FIG. I.12 – Illustration du fonctionnement du serveur PS

l'ordonnançabilité des tâches périodiques de l'application. En pratique, pour assurer un service acceptable aux requêtes apériodiques, il est souhaitable que sa durée d'exécution soit la plus grande possible et que sa période soit la plus courte possible.

3.2.2 Le serveur DS

La méthode du *serveur différé* (*Deferrable Server*) [SLS95] repose également sur l'exécution d'une tâche serveur (C_s, P_s) de période P_s et de capacité C_s, pour servir les tâches apériodiques. Le comportement du serveur DS est identique à celui du serveur PS à l'exception que le serveur DS *conserve* sa capacité courante jusqu'à la fin de sa période d'activation. Si à la fin de sa période, une partie de sa capacité n'a pas été utilisée, celle-ci sera perdue.

Son fonctionnement est illustré sur la figure I.13 dans laquelle nous considérons le même ensemble de tâches périodiques que dans la section précédente, et un serveur $DS(2,5)$ de capacité et de période égales à celles utilisées pour l'exemple illustratif du serveur PS. L'activité apériodique considérée est également identique. Toutes les tâches périodiques (y compris le serveur) sont ordonnancées selon l'algorithme RM. L'évolution de la capacité du serveur DS au cours du temps est figurée en-deçà de la séquence d'ordonnancement des tâches périodiques.

Nous constatons cette fois-ci que la requête $R_1(2,2)$ est servie immédiatement par la tâche serveur DS qui dispose encore d'une capacité de traitement à l'instant $t = 2$, ce qui améliore le temps de réponse de la requête apériodique. Globalement, le serveur DS affiche de meilleures performances que le serveur PS. Les complexités de calcul et d'implémentation restent toujours faibles.

Les deux serveurs de tâches apériodiques présentés ci-dessus sont parmi les plus connus. D'autres méthodes basées sur des principes similaires sont celles du *Priority*

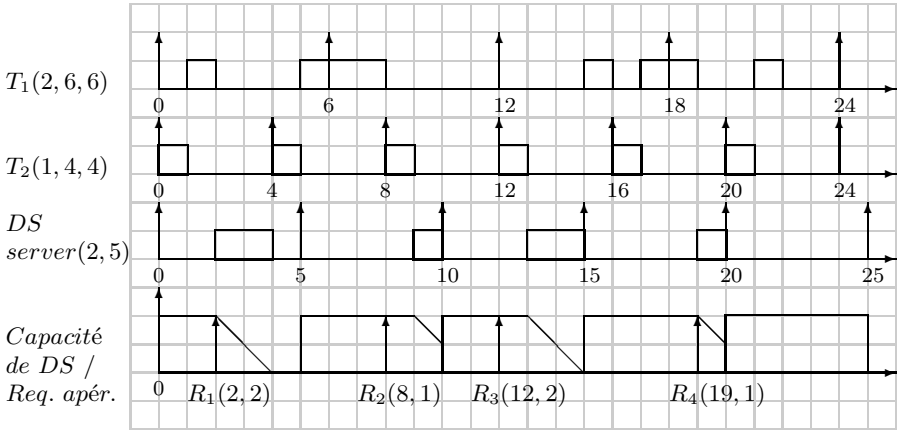

FIG. I.13 – Illustration du fonctionnement du serveur DS

Exchange server [LSS87] et du *Sporadic Server* [SSL89] établies respectivement par Lehoczsky *et al.*, et Sprunt *et al.* Par ailleurs, Lehoczky et Ramos-Thuel proposèrent en 1992 une méthode de service des requêtes apériodiques appelée *Slack Stealer* [LR92]. Celle-ci est basée sur l'idée de récupérer le maximum de temps possible au niveau des tâches périodiques, sans compromettre le respect de leurs échéances. Cette approche, basée sur l'évaluation de la laxité des tâches périodiques, a été étendue par la suite par Davis *et al.* [DTB93] et Tia *et al.* [TLS95].

3.3 Les approches basées sur un serveur à priorités dynamiques

La plupart des méthodes étudiées pour des algorithmes à priorités fixes ont été étendues aux algorithmes à priorités dynamiques. C'est ainsi que plusieurs mécanismes ont été proposés sous EDF d'une part par Ghazalie et Baker [GB95] (*Deadline Deferrable Server*, *Deadline Sporadic Server* et *Deadline Exchange Server*), et d'autre part par Spuri et Buttazzo [SP94, SP96] (*Dynamic Sporadic Server*, *Dynamic Priority Exchange Server* et *Improved Priority Exchange Server*).

Nous ne décrirons pas leur comportement dans cette partie, pour nous intéresser plus spécifiquement aux algorithmes étudiés dans le cadre de la thèse à savoir : les serveurs EDL [CC89], TBS [SP94, SP96], et TB*[BS99], et ce, compte tenu de leur optimalité. Le fonctionnement détaillé est présenté ci-après.

3.3.1 Le serveur EDL

3.3.1.1 Description. Le serveur Earliest Deadline as Late as possible (EDL) [CC89] repose sur un algorithme d'ordonnancement dynamique, capable d'assurer la gestion de tâches apériodiques au sein d'une application temps réel. Le mécanisme du serveur EDL est basé sur l'idée suivante : les temps creux d'un ordonnanceur EDL sont utilisés pour ordonnancer les requêtes apériodiques au plus tôt, en reportant l'exécution des tâches

périodiques au plus tard, de façon similaire à l'effet observé dans le cas du serveur *Slack Stealer* proposé par Lehoczky et Ramos-Thuel [LR92] pour des systèmes à priorités fixes.

Plus précisément, le serveur EDL consiste à exécuter les tâches périodiques au plus tôt lorsqu'il n'y a aucune activité apériodique. Dans le cas contraire, chaque fois qu'une requête apériodique survient, toutes les tâches périodiques sont ordonnancées au plus tard, dans le respect de l'ensemble des échéances des tâches. En d'autres termes, l'algorithme EDL tire profit de la laxité effective (c'est-à-dire de l'intervalle entre la date de fin d'exécution et l'échéance) des tâches périodiques, afin de minimiser le temps de réponse des apériodiques. En effet, lorsqu'il y a occurrences de requêtes apériodiques, les instances périodiques actives possèdent assez de laxité pour être préemptées.

Dans [CC89], Chetto et Chetto présentent une méthode simple pour déterminer la localisation et la durée des temps creux dans n'importe quelle fenêtre d'une séquence produite par les deux différentes implémentations de EDF, à savoir EDS (Earliest Deadline as Soon as poosible) et EDL. La terminologie utilisée par les auteurs est la suivante : f_Y^X représente la fonction de disponibilité définie vis à vis d'un ensemble de tâche Y et d'un algorithme d'ordonnancement X,

$$f_Y^X(t) = \begin{cases} 1 & \text{si le processeur est inactif à } t \\ 0 & \text{sinon} \end{cases} \tag{I.10}$$

Pour tous instants t_1 and t_2, l'intégrale $\int_{t_1}^{t_2} f_Y^X(t)dt$ fournit le nombre total d'unités de temps creux disponibles dans l'intervalle de temps $[t_1, t_2]$, désigné par $\Omega_Y^X(t_1, t_2)$.

3.3.1.2 Détermination des temps creux statiques. Le calcul de f^{EDL} pour la localisation et la détermination de la durée des temps creux, repose sur deux vecteurs : le *vecteur des échéances* et le *vecteur des temps creux*. Le vecteur des échéances représente l'ensemble des instants auxquels un temps creux est observable sur $[0, P[$, avec $P = ppcm(P_1, P_2, ..., P_n)$, l'hyperpériode égale au plus petit commun multiple des périodes des tâches. Le vecteur des temps creux est utilisé pour consigner la durée des temps creux associés à ces différents instants.

Calcul du vecteur statique des échéances. Considérons un ensemble de n tâches périodiques noté $\mathcal{T} = \{T_1, ..., T_n\}$ ordonnancées sur un système monoprocesseur. Chaque tâche $T_i = (C_i, P_i)$ est caractérisée par une durée d'exécution dans le pire cas C_i et par une période P_i.

Le vecteur statique des échéances, noté $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$, représente l'ensemble des instants considérés sur l'hyperpériode $[0, P]$, précédant un temps creux. Il se construit à partir des échéances distinctes des tâches périodiques. En effet, le début de toute période de temps creux coïncide avec l'échéance d'une requête périodique [Sil99]. Par conséquent, les instants k_i du vecteur $\mathcal{K}$ sont définis par le théorème suivant [CC89] :

Théorème 11 *Les instants k_i du vecteur $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$ sont définis comme suit :*

$$k_i = x.d_j \tag{I.11}$$

avec $x = 1, ..., \frac{P}{P_j}$, $k_i < k_{i+1}$, $k_0 = 0$ *et* $k_q = P - \min\{P_j; 1 \leq j \leq n\}$

Calcul du vecteur statique des temps creux. Le vecteur $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des échéances. Δ_i représente la longueur du temps creux débutant au temps k_i. Le théorème suivant [CC89] fournit la formule de récurrence pour le calcul du vecteur $\mathcal{D}^*$:

Théorème 12 *Les durées des temps creux du vecteur* $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ *calculées sur [0,P[sont définies comme suit :*

$$\Delta_q^* = \min\{P_i; 1 \leq i \leq n\} \tag{I.12}$$

$$\Delta_i^* = \sup(0, F_i), \quad \textit{pour } i = q-1 \textit{ à } 0 \tag{I.13}$$

$$\textit{avec} \quad F_i = (P - k_i) - \sum_{j=1}^{n} \lceil \frac{P - k_i}{P_j} \rceil C_j - \sum_{k=i+1}^{q} \Delta_k^*$$

Illustration du calcul hors-ligne de f^{EDL}. Considérons l'ensemble de tâches $\mathcal{T} = \{T_1, T_2\}$ constitué de deux tâches périodiques $T_1(3, 10)$ et $T_2(3, 6)$.

En appliquant les formules I.12 et I.13, on obtient le vecteur statique des échéances $\mathcal{K} = (0, 6, 10, 12, 18, 20, 24)$ et le vecteur statique des temps creux $\mathcal{D}^* = (3, 0, 0, 2, 0, 1, 0)$. Le calcul des intervalles de temps creux fournit la séquence EDL représentée sur la figure I.14.

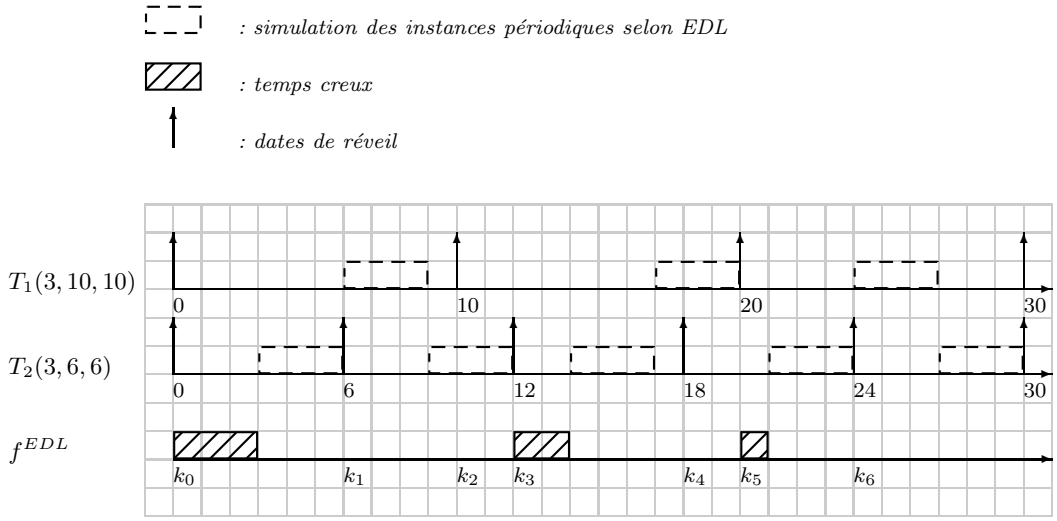

FIG. I.14 – Calcul des temps creux statiques selon EDL

3.3.1.3 Détermination des temps creux dynamiques.

Calcul du vecteur dynamique des échéances. On note τ le temps courant coïncidant avec l'arrivée d'une tâche apériodique. Le vecteur dynamique des échéances $\mathcal{K}(\tau)$ représente les instants supérieurs ou égaux à τ précédant un temps creux. Comme dans le cas statique, il se construit à partir des échéances distinctes des tâches périodiques. Soit h l'index tel que $k_h = \sup\{d; d \in \mathcal{K} \text{ et } d < \tau\}$. Alors, ce vecteur regroupe les instants tels que $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$.

Calcul du vecteur dynamique des temps creux. Le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ traduit les longueurs des temps creux associés aux instants du vecteur $\mathcal{K}$. Δ_i avec $h < i \leq q$ représente la longueur du temps creux débutant au temps k_i. Par ailleurs, chaque tâche périodique T_j est caractérisée par ses paramètres statiques C_j et P_j, mais également par la quantité de temps processeur $A_j(\tau)$ déjà allouée à son instance courante au temps τ, ainsi que par l'échéance d_j de sa requête courante au temps τ. On note M, la plus grande échéance parmi toutes les instances de tâches périodiques actives et l'on figure dans le vecteur des temps creux l'indice f tel que $k_f = \min\{k_i; k_i > M\}$. $\mathcal{D}^*(\tau)$ est alors complètement défini par la relation de récurrence suivante décrite dans le théorème qui suit [CC89] :

Théorème 13 *Les durées des temps creux du vecteur* $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ *sont définies comme suit :*

$$\Delta_i^*(\tau) = \Delta_i^*, \quad \textit{pour } i = q \textit{ à } f \tag{I.14}$$

$$\Delta_i^*(\tau) = \sup(0, F_i(\tau)), \quad \textit{pour } i = f-1 \textit{ à } h+1 \tag{I.15}$$

$$\textit{avec} \quad F_i(\tau) = (P - k_i) - \sum_{j=1}^{n} \lceil \frac{P - k_i}{P_j} \rceil C_j + \sum_{\substack{j=1 \\ d_j > k_i}}^{n} A_j(\tau) - \sum_{k=i+1}^{q} \Delta_k^*(\tau)$$

$$\Delta_h^*(\tau) = (P - \tau) - \sum_{j=1}^{n} (\lceil \frac{P - \tau}{P_j} \rceil C_j - A_j(\tau)) - \sum_{k=h+1}^{q} \Delta_k^*(\tau) \tag{I.16}$$

Illustration du calcul en-ligne de f^{EDL}. Nous pouvons à présent appliquer ces résultats à l'ensemble $\mathcal{T} = \{T_1(3, 10, 10), T_2(3, 6, 6)\}$ précédemment décrit dans la Table I.1. Cet ensemble de tâches périodiques est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 5$. On suppose qu'une requête apériodique de durée 4 unités de temps survient au temps $\tau = 5$, impliquant un calcul en-ligne des temps creux. Supposons que l'on souhaite calculer le maximum de temps processeur libre dans l'intervalle $[5, 30]$ relativement à l'ensemble $\mathcal{T}$. A partir des formules I.14, I.15 et I.16, on extrait le vecteur dynamique des échéances $\mathcal{K}(\tau) = (5, 6, 10, 12, 18, 20, 24)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (3, 2, 0, 2, 0, 1, 0)$, comme l'illustre la figure I.15.

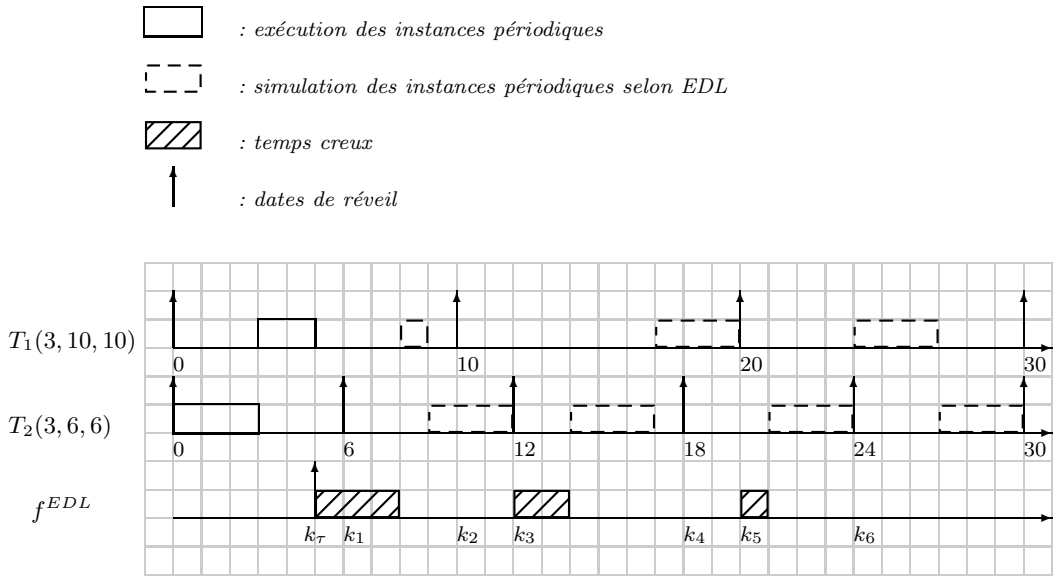

FIG. I.15 – Calcul des temps creux en dynamique à l'instant $\tau = 5$

Dans cet exemple, les vecteurs dynamiques obtenus en exécutant les tâches périodiques au plus tard depuis l'instant $\tau = 5$, fournissent une réponse optimale vis-à-vis de la requête apériodique, à condition que celle-ci soit exécutée dans les temps creux de la séquence EDL établie à $\tau = 5$. On peut observer sur la figure I.16 que la requête apériodique reçoit bien 4 unités de temps (3 unités de temps dans le premier intervalle débutant à $t = 5$ et 1 unité de temps dans le second débutant à $t = 12$). Son temps de réponse (optimal) est ici de 8 unités de temps.

3.3.1.4 Cas de tâches apériodiques critiques. Considérons à présent le problème de l'acceptation de tâches apériodiques à contraintes strictes. On suppose que ces tâches sont synchrones en ce sens que leur date d'arrivée correspond avec leur date de réveil. Soit τ le temps courant coïncidant avec l'arrivée d'une requête apériodique critique $R(\tau, c, d)$, où c représente sa durée d'exécution au pire cas et d son échéance.

On suppose que le système peut présenter plusieurs tâches apériodiques critiques à l'état prêt au temps τ correspondant aux différentes requêtes précédemment acceptées. Par conséquent, celles-ci n'ont pas terminé leur exécution à τ. On note $\mathcal{R}(\tau) = \{R_i(c_i(\tau), d_i)$, i=1 à $q(\tau)\}$ l'ensemble des q tâches apériodiques critiques présentes dans le système à τ. $c_i(\tau)$ désigne la durée d'exécution dynamique de R_i (c'est-à-dire sa durée d'exécution restante) et d_i son échéance. L'ensemble $\mathcal{R}(\tau)$ est ordonné tel que $i < j$ implique $d_i \leq d_j$.

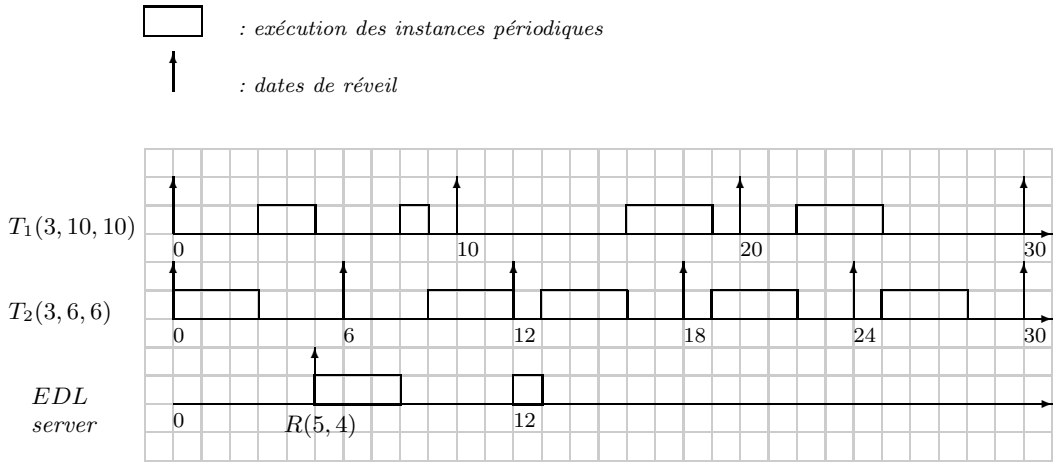

FIG. I.16 – Exécution des apériodiques selon la séquence EDL calculée à $\tau = 5$

Dans [SCE90], Silly *et al.* proposent le test d'acceptation *optimal* suivant pour des tâches apériodiques critiques gérées par le serveur EDL :

Théorème 14 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{I.17}$$

$\delta_i(\tau)$, appelée laxité de la tâche R_i au temps τ, représente la longueur de l'intervalle de temps qui sépare sa fin d'exécution au plus tôt de son échéance. $\Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i)$ représente la somme totale des temps creux calculée entre τ et d_i. La sommation $\sum_{j=1}^{i} c_j(\tau)$ traduit la somme des exécutions restantes à l'instant τ sur l'ensemble des tâches apériodiques critiques dont l'échéance est inférieure ou égale à d_i.

3.3.1.5 Résultats d'optimalité et complexité. Il a été démontré [Sil99] que le serveur EDL nécessite un calcul en-ligne des temps creux du processeur uniquement aux instants relatifs à l'occurrence d'une nouvelle tâche apériodique. La propriété fondamentale du serveur EDL est qu'il garantit le maximum de temps creux dans un intervalle donné pour n'importe quel ensemble de tâches. EDL a été démontré optimal [Sil99]. Ce résultat est rappelé dans le Théorème 15.

Théorème 15 *Soit X un algorithme d'ordonnancement préemptif quelconque. A tout instant t,*

$$\Omega^{X}_{\mathcal{T}}(0, t) \leq \Omega^{EDL}_{\mathcal{T}}(0, t) \tag{I.18}$$

Un autre résultat significatif réside dans la complexité de l'établissement de la séquence EDL. Le calcul en-ligne des temps creux [Sil99] s'effectue en $O(\lfloor \frac{R}{p} \rfloor n)$ où n désigne le nombre de tâches périodiques, R l'échéance la plus éloignée parmi les tâches actives et p la plus petite période.

3.3.2 Le serveur TBS

3.3.2.1 Description. Le serveur *Total Bandwith Server* (*TBS*) [SP94, SP96] est un mécanisme simple pour servir des tâches apériodiques de manière efficace. A chaque fois qu'une requête apériodique entre dans le système, le serveur TBS lui assigne une échéance en fonction de sa largeur de bande (en termes de temps d'exécution CPU).

Lorsque la k-ième requête apériodique arrive au temps $t = r_k$, elle reçoit une échéance :

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k^a}{U_s}. \tag{I.19}$$

où C_k^a est la durée d'exécution de la requête et U_s correspond au facteur d'utilisation du serveur (c'est-à-dire sa largeur de bande). Par définition, $d_0 = 0$. On peut noter que lorsque la nouvelle échéance d_k est assignée, la largeur de bande déjà allouée aux requêtes précédentes est prise en compte par la valeur d_{k-1}.

Une fois que cette échéance *fictive* est assignée à la requête apériodique, celle-ci est ordonnancée conjointement avec les tâches périodiques selon EDF. L'avantage de cette approche consiste dans le fait que l'overhead d'implémentation est pratiquement négligeable.

3.3.2.2 Conditions d'ordonnançabilité. Intuitivement, l'assignation des échéances est telle que dans chaque intervalle de temps, la fraction du temps processeur allouée par EDF aux requêtes apériodiques n'excède jamais le facteur d'utilisation U_s du serveur. Par conséquent, l'ordonnançabilité d'un ensemble de tâches périodiques en présence d'un serveur TBS peut simplement être testée par la condition *nécessaire et suffisante* énoncée dans le théorème 16 et formellement prouvée dans [SP96].

Théorème 16 *Etant donnés un ensemble de n tâches périodiques ayant un facteur d'utilisation du processeur U_p, et un ensemble de tâches apériodiques servies par un serveur TBS ayant un facteur d'utilisation du processeur U_s, le système entier est ordonnançable si et seulement si*

$$U_p + U_s \leq 1. \tag{I.20}$$

3.3.2.3 Illustration. Considérons de nouveau l'ensemble $\mathcal{T} = \{T_1(3, 10, 10), T_2(3, 6, 6)\}$ de la Table I.1. Le facteur d'utilisation du processeur relatif aux tâches périodiques est égal à $U_p = \sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{3}{10} + \frac{3}{6} = 80\%$. La figure I.17 fournit un exemple d'ordonnancement produit par le serveur TBS avec $U_s = 1 - U_p = 20\%$ (tout le temps processeur non utilisé par les tâches périodiques est alloué au serveur). La condition $U_p + U_s \leq 1$ est vérifiée, le système considéré est donc ordonnançable.

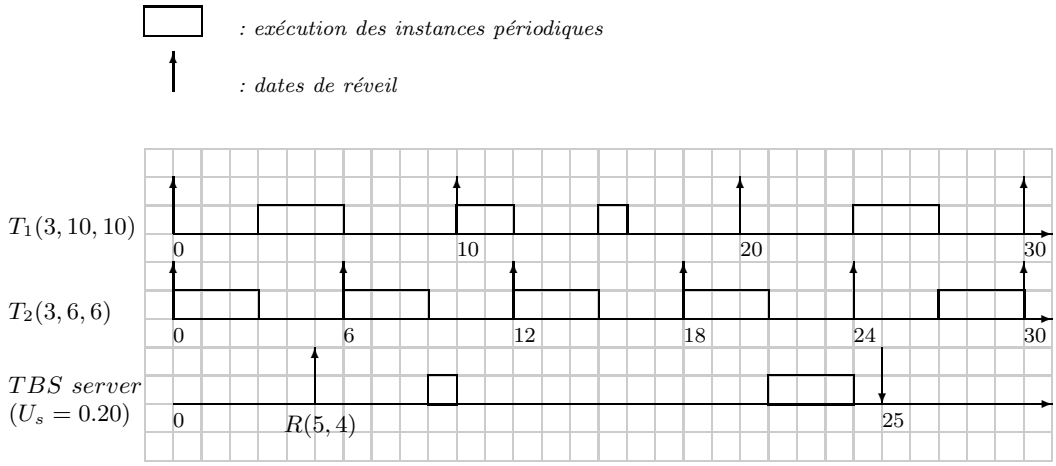

FIG. I.17 – Illustration du fonctionnement du serveur TBS

Une requête apériodique $R(5, 4)$ survient à l'instant $t = 5$. Elle reçoit alors une échéance fictive $d = \max(5, 0) + \frac{4}{0.20} = 5 + 20 = 25$. Les requêtes périodiques dont les échéances sont plus proches que d sont ordonnançées en priorité. Par conséquent, nous pouvons observer un temps de réponse à la requête R de 20 unités de temps.

3.3.3 Le serveur TB*

3.3.3.1 Description. L'idée de l'algorithme TB* (*Optimal Total Bandwidth server*) [BS99] est d'assigner à chaque requête apériodique une échéance plus petite que celle fournie par TBS. Chaque fois qu'une tâche apériodique est prête (c'est-à-dire, dès qu'elle est réveillée alors qu'il n'y a pas de tâches apériodiques à exécuter, ou bien lorsque toutes les requêtes apériodiques précédentes ont terminé leur exécution), TB* lui assigne en premier lieu une échéance selon TBS avec une largeur de bande $U_s = 1 - U_p$. Ensuite, l'algorithme essaie de raccourcir au maximum cette échéance de manière à améliorer le temps de réponse des requêtes apériodiques, tout en conservant l'ensemble des tâches périodiques ordonnançable.

Si d_k^0 correspond à l'échéance assignée à la requête apériodique selon TBS, la nouvelle échéance d_k^1 est calculée comme la date f_k^0 de fin d'exécution au pire cas de la requête apériodique ordonnancée selon l'échéance d_k^0. Le processus de raccourcissement de l'échéance peut être appliqué de façon itérative à chaque nouvelle échéance, jusqu'à ce qu'aucune amélioration ne soit plus possible, étant donné que l'ordonnançabilité de l'ensemble des tâches périodiques doit être conservée. Si d_k^s est l'échéance assignée à la requête apériodique J_k au pas s, l'ordonnançabilité est conservée en assignant à J_k une nouvelle échéance donnée par :

$$d_k^{s+1} = t + C_k^a + I_p(t, d_k^s) \tag{I.21}$$

où t est le temps courant (correspondant à la date de réveil r_k de la requête J_k ou bien à la date de terminaison de la requête précédente), C_k^a est la durée d'exécution au pire cas requise par J_k, et $I_p(t, d_k^s)$ est l'interférence due aux instances périodiques dans l'intervalle $[t, d_k)$.

L'interférence périodique $I_p(t, d_k^s)$ peut être exprimée comme la somme de deux termes, $I_a(t, d_k^s)$ et $I_f(t, d_k^s)$, où $I_a(t, d_k^s)$ est l'interférence due aux instances périodiques actives au temps courant ayant des échéances strictement inférieures à d_k^s, et $I_f(t, d_k^s)$ est l'interférence future due aux instances périodiques activées après le temps courant t et ayant leur échéance avant d_k^s. Par conséquent,

$$I_a(t, d_k^s) = \sum_{\tau_i \; active,\, d_i < d_k^s} c_i(t) \tag{I.22}$$

et

$$I_f(t, d_k^s) = \sum_{i=1}^{n} \max\left(0, \lceil \frac{d_k^s - next\_ri(t)}{T_i} \rceil - 1\right) C_i \tag{I.23}$$

où C_i est la durée d'exécution totale de la tâche τ_i, T_i sa période, $c_i(t)$ le temps d'exécution restant au temps t pour achever l'instance courante, et $next\_ri(t)$ est l'instant supérieur ou égal à t auquel la prochaine instance de la tâche périodique τ_i sera activée.

3.3.3.2 Résultats d'optimalité et complexité. Si la durée d'exécution effective des tâches (*Average Case Execution Time ou ACET*) est égale à leur durée d'exécution dans le pire-cas (*Worst Case Execution Time ou WCET*) alors l'algorithme TB^* est optimal, dans le sens où il minimise le temps de réponse des tâches apériodiques. Cette optimalité repose néanmoins sur deux autres hypothèses à savoir que les requêtes apériodiques sont servies selon la politique FIFO et que les conflits d'échéances sont résolus en faveur des tâches apériodiques. Ce résultat est résumé dans le théorème suivant [BS99] :

Théorème 17 *Etant donné un ensemble de tâches périodiques ordonnancées selon EDF et un ensemble de tâches apériodiques servies selon la technique FIFO, l'algorithme TB^* minimise le temps de réponse de chaque tâche apériodique et ce, parmi tous les algorithmes d'ordonnancement qui respectent les échéances de toutes les tâches périodiques.*

Buttazzo et Sensini dans [BS99] évaluent la complexité de l'algorithme TB^* en fournissant une borne supérieure relative au nombre N d'étapes que doit réaliser l'algorithme pour trouver l'échéance optimale d_k^*, c'est-à-dire l'échéance qui minimise le temps de réponse de la requête apériodique R_k. Cette borne supérieure pour N est obtenue en considérant que dans le pire-cas, exactement une échéance de tâche périodique est "croisée" à chaque étape de raccourcissement de l'échéance. Ceci conduit à l'équation suivante (qui fournit une surestimation du nombre d'itérations) :

$$N < n + \frac{\sum_{i=1}^{n} C_i + U_p C_k^a}{1 - U_p} \sum_{i=1}^{n} \frac{1}{P_i} \tag{I.24}$$

Dans le cas où l'ensemble de tâches périodiques considéré possède un facteur d'utilisation du processeur élevé et que les tâches présentent de petites périodes, beaucoup d'itérations seront nécessaires pour trouver l'échéance optimale. Cependant, dans la plupart des situations pratiques, les auteurs montrent par le biais d'expériences en simulation, que l'algorithme atteint une performance proche de l'optimal suite à un petit nombre d'étapes. Le nombre d'étapes maximum autorisées par TB* peut être ajusté. L'algorithme qui effectue au plus N étapes de raccourcissement est alors dénommé TB(N).

3.3.3.3 Conditions de faisabilité. A l'image de la condition établie pour le serveur TBS, l'ordonnançabilité d'un ensemble de tâches périodiques en présence d'un serveur TB* peut simplement être testée par la condition *nécessaire et suffisante* énoncée dans le théorème 18.

Théorème 18 *Etant donnés un ensemble de n tâches périodiques ayant un facteur d'utilisation du processeur U_p, et un ensemble de tâches apériodiques servies par un serveur TB* ayant un facteur d'utilisation du processeur U_s, le système entier est ordonnançable si et seulement si*

$$U_p + U_s \leq 1. \tag{I.25}$$

3.3.3.4 Illustration. Considérons toujours l'ensemble $\mathcal{T} = \{T_1(3, 10, 10), T_2(3, 6, 6)\}$ de la Table I.1. Le facteur d'utilisation du processeur relatif aux tâches périodiques est égal à $U_p = \sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{3}{10} + \frac{3}{6} = 80\%$. La largeur de bande allouée au serveur TB* est telle que $U_s = 1 - U_p = 20\%$ (tout le temps processeur non utilisé par les tâches périodiques est alloué au serveur). La condition $U_p + U_s \leq 1$ est vérifiée, le système considéré est donc ordonnançable.

Une requête apériodique $R(5, 4)$ survient à l'instant $t = 5$. Elle reçoit alors une échéance $d_k^0 = \max(5, 0) + \frac{4}{0.20} = 5 + 20 = 25$ selon l'algorithme TBS. En appliquant les équations I.22 et I.23, on obtient :

$$I_a(5, 25) = c_1(5) + c_2(5) = 1 + 0 = 1$$
$$I_f(5, 25) = \max(0, \lceil \frac{25 - 10}{10} \rceil - 1)C_1 + \max(0, \lceil \frac{25 - 6}{6} \rceil - 1)C_2 = 3 + 9 = 12$$

et avec l'équation I.21, nous pouvons calculer l'échéance d_k^1 à l'étape 1 :

$$d_k^1 = t + C_k^a + I_a + I_f = 5 + 4 + 1 + 12 = 22$$

Les résultats des 5 autres étapes de raccourcissement de l'échéance sont fournis dans la Table I.2. L'ordonnancement produit par EDF en utilisant la plus petite échéance possible $d_k^* = d_k^5 = 13$ est illustré sur la figure I.18.

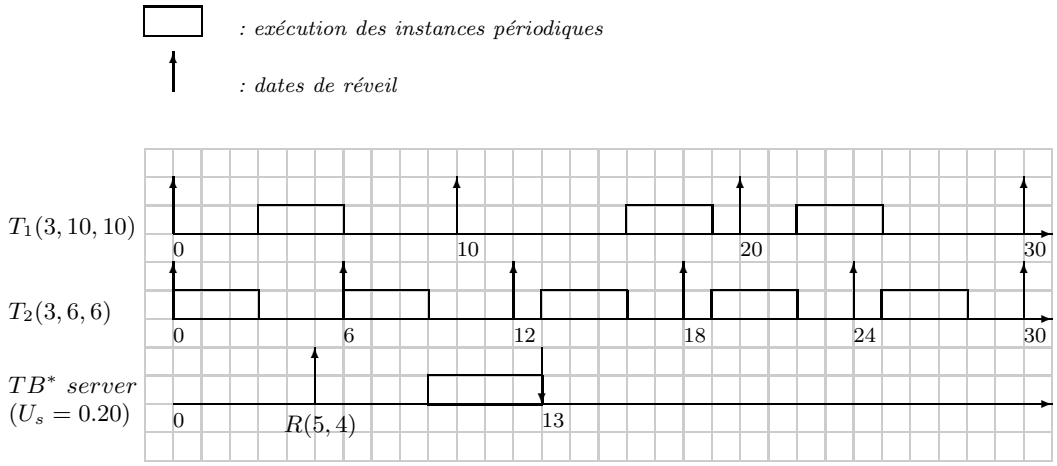

FIG. I.18 – Illustration du fonctionnement du serveur TB*

Etape	0	1	2	3	4	5
d_k^s	25	22	19	16	13	13

TAB. I.2 – Dates d'échéance calculées par l'algorithme TB*

Le temps de réponse optimal observé pour la requête R est alors de 8 unités de temps. Ce temps est le même que celui observé pour le serveur EDL pour le même ensemble de tâches car les deux serveurs sont tous les deux optimaux du point de vue de la minimisation du temps de réponse des requêtes apériodiques. Cependant, remarquons que ce résultat, basé sur l'hypothèse que la durée d'exécution effective des tâches périodiques est égale à leur durée d'exécution au pire-cas, n'est plus vrai dès lors que l'on considère des ensembles de tâches tels que $ACET < WCET$.

En effet, l'observation des ordonnancements de EDL (Figure I.16) et de TB* (Figure I.18) nous révèle que dans le cas où la requête R possèderait une durée d'exécution effective de 3 unités de temps au lieu de 4, les temps de réponse de R pour EDL et TBS seraient respectivement égaux à 3 et 7 unités de temps. Ainsi, nous pouvons affirmer que l'un des avantages du serveur EDL sur les autres approches réside dans le fait que les temps de réponse des requêtes apériodiques sont optimaux, même si le temps d'exécution moyen des tâches n'est pas égal au temps d'exécution au pire-cas. De plus, l'algorithme EDL n'a pas besoin, *a priori*, de connaître les durées d'exécution au pire-cas des requêtes apériodiques occurrentes pour fonctionner (contrairement aux algorithmes TBS et TB*).

4 Conclusion

Nous avons présenté dans ce chapitre une introduction à l'ordonnancement dans les systèmes temps réel. L'objet de ce chapitre était en premier lieu de décrire la terminologie et les concepts relatifs au domaine du temps réel et à la problématique d'ordonnancement. Ensuite, nous avons présenté les principaux algorithmes existants pour l'ordonnancement de tâches périodiques, en rappelant notamment les conditions d'ordonnançabilité et résultats d'optimalité de chacun d'entre eux. Puis, nous nous sommes intéressés à l'état de l'art relatif aux serveurs de tâches apériodiques, en exposant successivement les performances en termes de complexité et d'optimalité, des serveurs appartenant aux deux types d'approches communément rencontrées (serveurs à priorités statiques ou dynamiques).

Dans le chapitre suivant, nous nous focalisons sur l'état de l'art des modèles d'ordonnancement dédiés aux systèmes temps réel expérimentant des surcharges temporaires de traitements.

Chapitre II

Ordonnancement et gestion de surcharge

Ce second chapitre constitue un état de l'art sur l'ordonnancement tolérant les surcharges de traitement dans un environnement monoprocesseur. Nous présentons tout d'abord la notion de surcharge ainsi que ses différents critères d'évaluation. Puis, les principaux schémas d'ordonnancement dédiés aux systèmes surchargés sont exposés. Ensuite, les modèles de résolution existants liés à l'ordonnancement de tâches en présence de surcharge sont présentés. C'est ainsi que nous décrivons successivement les approches basées sur la valeur, les approches à pertes contraintes et les approches à tâches bipartites. En dernier lieu, une synthèse expose les avantages et inconvénients de chacune des approches étudiées.

---

1 La notion de surcharge et ses critères d'évaluation

1.1 Une description des cas de surcharge

Les concepteurs de systèmes temps réel *critiques* tentent habituellement d'anticiper toute éventualité et de l'incorporer dans la conception et le dimensionnement du système. Idéalement, un tel système ne devrait jamais être *surchargé* et son comportement devrait être conforme à la prévision faite initialement. En réalité, les systèmes peuvent être soumis en pratique à des *surcharges temporaires* (*transient overload*) de traitement. Nous reprenons ici la définition relative à la surcharge d'un système temps réel donnée dans [Dar03] :

Définition 8 *On dit d'un système temps réel qu'il est en surcharge pendant une période donnée si sur cette période, l'utilisation du processeur dépasse la borne autorisée par l'algorithme d'ordonnancement mis en œuvre.*

Lorsque les ressources nécessaires pour exécuter les tâches du système sont plus importantes que les ressources disponibles (en particulier les ressources processeur), le système entre en phase de *surcharge*. Il est alors impossible de respecter toutes les échéances des tâches. En particulier, des tâches critiques peuvent manquer leur échéance, ce qui peut avoir des conséquences catastrophiques sur l'environnement contrôlé. Dans certains cas, le comportement global du système peut être compromis. Par conséquent, pour faire face aux surcharges, la performance du système doit pouvoir être dégradée localement ou globalement de manière à conserver le système dans un état *sécuritaire*. En effet, cette maîtrise est nécessaire car un système qui s'effondre et subit une importante perte de performance lors d'une situation d'urgence tendra davantage à empirer la situation qu'à l'améliorer.

Le comportement de Rate-Monotonic en surcharge est tel que ce sont les tâches de plus longues périodes qui violent leur échéance [Del94]. Une solution pratique consiste alors à réduire la période d'exécution d'une tâche jugée importante. Cependant, cette action délicate entraîne une augmentation de l'utilisation processeur de la tâche et une confusion des notions d'urgence et d'importance.

Quant à l'algorithme EDF, son comportement est instable en cas de surcharge. Des expériences menées par Locke [Loc86] ont montré que EDF est sujet à l'*effet domino* (cas d'avalanche de fautes temporelles, les tâches qui manquent leur échéances retardent les autres qui à leur tour manquent leur échéances) et que sa performance est rapidement dégradée sur les intervalles de surcharge. Ceci est dû au fait que l'algorithme EDF octroît toujours la plus grande priorité à la tâche dont l'échéance est la plus proche. En cas de surcharge, EDF ne fournit aucun type de garantie sur l'identité des tâches qui respecteront leurs contraintes temporelles : ce comportement qualifié d'indéterministe est particulièrement indésirable lorsque des cas de surcharges, dus à des modifications de l'environnement par exemple, surviennent inopinément et de manière intermittente. L'effet domino sous EDF est illustré sur la Figure II.1.

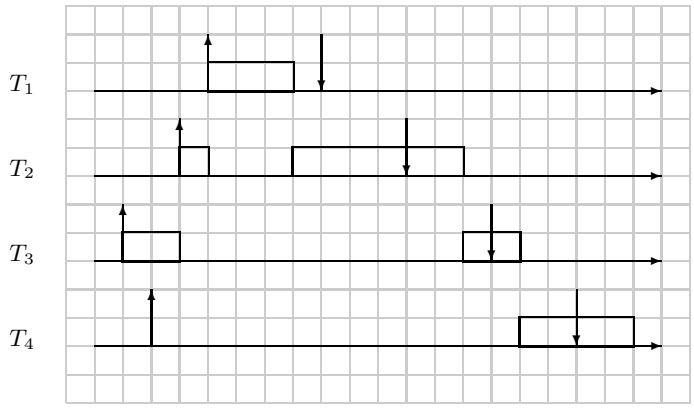

FIG. II.1 – Effet domino sous EDF

L'occurrence inattendue de la tâche T_1 retarde l'exécution de la tâche T_2 qui n'est plus en mesure de respecter son échéance. Ce retard repousse l'exécution de T_3 qui viole également son échéance, puis de T_4, etc.

Par conséquent, l'algorithme d'ordonnancement doit être explicitement conçu pour prendre en compte l'occurrence de surcharges temporaires de traitement, de manière à éviter les effets Domino et à minimiser les dommages causés par la violation d'une échéance.

1.2 Les causes de surcharge

On distingue plusieurs causes de surcharge relevant de paramètres divers et variés associés au *système* et à son *environnement*.

En premier lieu, la surcharge peut être conséquente à une *mauvaise conception du système* lui-même. Les durées d'exécution réelles des tâches peuvent s'avérer supérieures aux durées évaluées dans le pire cas au niveau spécification. Il peut également arriver que les surcoûts associés au système d'exploitation ou au matériel aient été sous-estimés ou mésestimés.

D'autre part, le *mauvais fonctionnement d'un périphérique* d'entrée peut mener à un cas de surcharge du système. Les *exceptions système* soulevées par le noyau sont également des facteurs influant sur la charge potentielle infligée au système. De même, des *variations au niveau de l'environnement*, ou bien *l'arrivée d'un grand nombre d'événements rapprochés dans le temps*, vont contribuer à moduler l'évolution de la charge.

Une ligne de communication encombrée, une transmission défectueuse ou la contention sur une ressource critique sont autant de facteurs pouvant influer sur la *fin d'exécution tardive d'une tâche*, donnant lieu à un cas de surcharge.

Enfin, on peut citer d'autres causes de surcharge : un *flux élevé de tâches apériodiques*, un *test d'ordonnançabilité erroné*, etc.

1.3 L'évaluation de la charge d'un système

1.3.1 Le facteur d'utilisation du processeur

De manière générale, le *facteur d'utilisation du processeur* (*Effective Processor Utilization noté EPU*) pour un ensemble de n tâches périodiques est défini comme la somme sur chacune des tâches du rapport de sa durée d'exécution au pire-cas sur sa période [LL73] :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \tag{II.1}$$

Plus précisément, étant donné un intervalle de temps de surcharge (intervalle durant lequel l'utilisation du processeur dépasse la borne autorisée par l'algorithme d'ordonnancement mis en œuvre), débutant au temps t_s et se terminant au temps t_f, le facteur

d'utilisation du processeur sur cet intervalle est calculé comme suit :

$$\frac{\sum_{i \in Q} C_i[t_s, t_f[}{t_f - t_s} \tag{II.2}$$

où Q dénote l'ensemble des tâches respectant leur échéance sur l'intervalle $[t_s, t_f]$ et $C_i[t_s, t_f]$ représente le service reçu par la tâche i sur l'intervalle $[t_s, t_f]$.

Plus généralement, l'EPU constitue une *garantie de performance* dans le pire cas.

1.3.2 Le seuil d'utilisation

Il s'agit d'un degré d'utilisation du processeur au-dessus duquel certaines tâches commencent à violer leurs contraintes temporelles. L'efficacité d'un algorithme d'ordonnancement peut ainsi être évaluée puisque, plus le *seuil d'utilisation* (*breakdown utilization*) sera grand pour un algorithme, plus le temps processeur alloué à l'exécution des tâches temps réel sera important.

1.3.3 Le facteur de charge

Le *facteur de charge* (*load factor*) représente la proportion de temps que le processeur dédie à l'exécution d'une tâche de manière à ce que son échéance soit respectée. Sa définition dépend du type de tâches considérées et de leurs caractéristiques temporelles.

1.3.3.1 Cas des tâches apériodiques à contraintes relatives. Le facteur de charge du processeur pour un ensemble de n tâches apériodiques à contraintes relatives est donné par [SSR+98] :

$$\rho = \lambda \bar{C} \tag{II.3}$$

où $\lambda = \frac{1}{n-1} \sum_{i=1}^{n-1} t_{i-1} - r_i$, taux moyen d'arrivée entre 2 tâches (*average inter-arrival time*),

et $\bar{C} = \frac{1}{n-1} \sum_{i=1}^{n-1} c_i$, taux moyen de temps de service (*average computation time*).

1.3.3.2 Cas des tâches périodiques. Le facteur de charge du processeur pour un ensemble de n tâches périodiques à contraintes strictes est défini comme la somme du rapport des durées d'exécution au pire-cas sur les délais critiques des tâches :

$$\rho = \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{II.4}$$

1.3.3.3 Cas des tâches périodiques à échéances sur requêtes. Le facteur de charge du processeur pour un ensemble de n tâches périodiques à échéances sur requêtes ($D_i = P_i$) est donné par [SSR+98] :

$$\rho = \sum_{i=1}^{n} \frac{C_i}{D_i} = \sum_{i=1}^{n} \frac{C_i}{P_i} \quad (= U) \tag{II.5}$$

La définition peut être étendue à celle du *facteur de charge instantané*, défini comme le maximum de la demande processeur calculé sur tous les intervalles définis entre la période courante et les échéances de toutes les tâches actives [BS94] :

$$\rho(t) = \max_i \rho_i(t) \tag{II.6}$$
$$\text{avec} \quad \rho_i(t) = \frac{\sum_{r_k \leq t, d_k \leq d_i} c_k(t)}{d_i - t}$$

Le facteur de charge instantané est illustré sur la Figure II.2 :

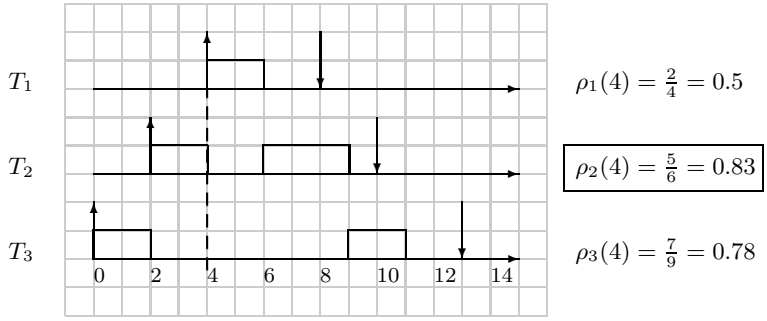

FIG. II.2 – Calcul du facteur de charge instantané à $t = 4$

Si $\rho > 1$, alors il y a *surcharge* de traitements. Pour calculer plus précisément le facteur de charge, il faut en outre prendre en compte le surplus de temps processeur requis par l'exécutif pour gérer l'ensemble des tâches, temps appelé *overhead*.

1.3.4 La laxité du processeur

Du fait des échéances, la connaissance parfaite du facteur d'utilisation du processeur et du facteur de charge n'est pas suffisante pour évaluer l'effet de surcharge sur les contraintes temporelles. On introduit donc le paramètre $LP(t)$, la *laxité du processeur à*

t (*processor slack*), comme le temps maximum où le processeur peut rester libre (*idle*) après t sans entraîner le manquement d'une échéance pour une tâche. $\forall t$, on doit avoir $LP(t) \geq 0$. Pour calculer la laxité, l'ordre des tâches sur le processeur doit être connu ; il est alors possible de calculer la laxité conditionnelle $LC_i(t)$ de chaque tâche i de la manière suivante [CDK+00] :

$$LC_i(t) = D_i - \sum_j C_j(t) \tag{II.7}$$

où la somme sur j calcule le temps d'exécution de toutes les tâches en attente (y compris la tâche i), activées au temps t et qui précèdent la tâche i dans la séquence d'ordonnancement.

La laxité $LP(t)$ correspond alors à la plus petite valeur de la laxité conditionnelle $LC_i(t)$.

1.3.5 Les temps creux du processeur

On entend par *temps creux du processeur* (*processor idle times*) pour une séquence d'ordonnancement donnée, l'ensemble des intervalles de temps pour lesquels le processeur peut rester inactif.

1.3.6 Le taux de respect

Le *taux de respect* (*hit ratio*) aussi appelé *taux de succès* (*success rate*) représente la proportion de travaux des tâches qui respectent leurs contraintes temporelles.

1.3.7 Le taux de garantie

Lorsque le schéma d'ordonnancement inclut un test d'acceptation, on parle de *taux de garantie* (*guarantee ratio*), appelé aussi *taux d'acceptation* (*acceptance ratio*). Ce taux exprime la proportion de tâches dont l'exécution est garantie vis-à-vis du nombre de tâches qui demandent à être exécutées.

2 Les schémas d'ordonnancement pour la résolution de surcharge

On distingue habituellement plusieurs classes d'ordonnanceurs :

– les ordonnanceurs *au mieux* (*best-effort*) sans test d'acceptation pour lesquels les tâches sont ordonnancées jusqu'à leur terminaison ou jusqu'à dépassement d'échéance,

– les ordonnanceurs avec test d'acceptation et sans remise en cause des tâches acceptées (dits aussi *à garantie*),

– les ordonnanceurs avec exécution conditionnelle (test à l'activation ou avant démarrage) et politique de rejet (dits aussi *robustes*).

2.1 Les ordonnanceurs au mieux (Best-Effort)

Selon *Best-Effort*, chaque fois qu'une nouvelle tâche arrive, l'algorithme accepte systématiquement la tâche et l'insère dans la file d'attente des tâches prêtes.

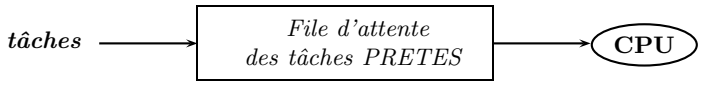

FIG. II.3 – Ordonnancement Best-Effort

Il n'y a ici aucune prédiction des cas de surcharges. La performance est contrôlée via une affectation de priorité (basée sur la valeur) jugée appropriée. Ce type d'approche est le plus souvent sensible à l'effet Domino.

2.2 Les ordonnanceurs à garantie (Guarantee)

Selon un algorithme *à garantie*, chaque fois qu'une nouvelle tâche arrive, l'algorithme vérifie l'ordonnançabilité : la tâche est soumise à un test d'acceptation qui empêche le système d'être surchargé. Si le test réussit, la tâche est acceptée ; sinon, elle est rejetée.

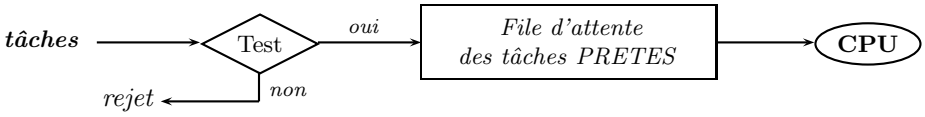

FIG. II.4 – Ordonnancement à garantie

Contrairement aux ordonnanceurs best-effort, les ordonnanceurs basés sur la garantie ont l'avantage d'empêcher les effets d'avalanche de fautes temporelles. Cependant, ils ne prennent pas en compte les valeurs associées à l'importance des tâches. De ce fait, en cas de surcharge, les nouvelles tâches seront toujours rejetées.

2.3 Les ordonnanceurs robustes (Robust)

Selon un algorithme *robuste*, l'ordonnancement et le rejet des tâches sont contrôlés par deux politiques séparées. Les tâches sont ordonnancées selon leurs échéances et rejetées

selon leur valeur d'importance. Les tâches rejetées peuvent être reprises plus tard dans le temps, si la charge du système le permet.

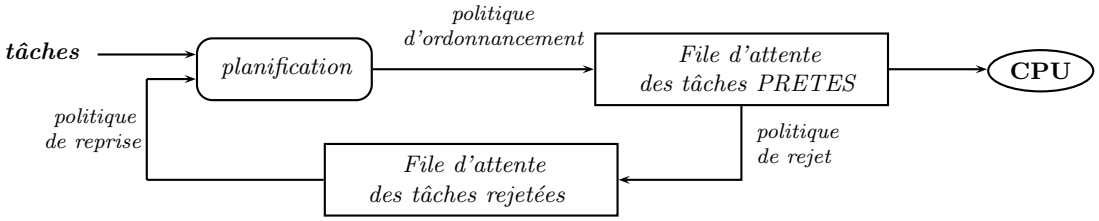

FIG. II.5 – Ordonnancement robuste

Les algorithmes d'ordonnancement basés sur ce modèle s'avèrent généralement plus performants en cas de surcharge temporaire de traitements, que ceux basés sur les deux modèles précédents [BS93].

En présence de surcharge, la solution pour contrôler le comportement du système est de borner ou de diminuer les besoins en ressources. Une fois la surcharge détectée ou anticipée (ce problème a été démontré NP-complet dans le cas général [BHR93]), plusieurs modèles de résolution peuvent être mis en œuvre dans l'objectif de diminuer les besoins en ressource processeur. Lorsque le système est surchargé, l'exécution de certaines instances doit être écartée dans le but de permettre à d'autres instances de s'accomplir dans le respect de leur échéance. Ainsi, durant une phase de surcharge, une façon intuitive de mesurer la performance d'un algorithme d'ordonnancement consiste à évaluer la quantité de traitements effectuée par l'ordonnanceur. Plus cette quantité sera importante, meilleur sera l'algorithme.

Un premier courant se compose des approches dites *basées sur la valeur* qui reposent sur le constat suivant : étant donné que toutes les échéances ne peuvent pas être satisfaites, il est préférable que les tâches en retard soient les tâches les moins importantes vis-à-vis de l'application. En d'autres termes, faire la distinction entre l'*importance* d'une tâche et son *urgence* devient significatif dans le cadre de ces approches.

Le second courant, consiste à établir des modèles intégrant directement le fait que des tâches puissent être totalement ou en partie abandonnées. Celui-ci regroupe d'une part les *approches à tâches bipartites* où seule une partie de la tâche est garantie hors-ligne (l'exécution du reste dépendant de la charge du système) et d'autre part, les *approches à pertes contraintes* pour lesquelles la spécification du modèle précise quelles instances d'une tâche peuvent être supprimées.

Dans la suite, nous décrivons toutes ces approches d'ordonnancement visant à résoudre les cas de surcharge, en soulignant les objectifs poursuivis ainsi que les limitations inhérentes à chacune d'elles. Les algorithmes qui s'y rapportent sont brièvement présentés. Un accent tout particulier est accordé aux approches à pertes contraintes qui constituent l'un des piliers des travaux de thèse présentés dans les chapitres suivants.

3 Les approches basées sur la valeur

3.1 L'introduction d'un critère d'importance

Une nouvelle grandeur a été introduite dans le cadre de l'initiative Alpha [JNC+89], pour coder le caractère primordial des tâches dans l'application temps réel : la *fonction valeur* $v_i(t)$. Celle-ci traduit l'importance de la tâche vis-à-vis de l'application et ce, en fonction du temps auquel son résultat est produit. L'importance de l'exécution d'une tâche est ainsi exprimée en termes du bénéfice $v_i(t)$ qu'elle fournit au système ; bénéfice fonction du temps auquel l'exécution se termine comme l'illustre la figure II.6.

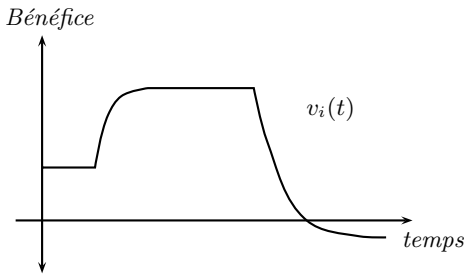

FIG. II.6 – Fonction valeur associée à une tâche

La *fonction valeur* associée à une tâche peut ne pas dépendre de l'instant auquel la tâche termine son exécution. Elle peut être alors définie par un *entier* dont la valeur est fixée à l'initialisation et reste immuable au cours de la vie de l'application. Il peut s'agir d'une constante arbitraire ($v_i = V_i$), de la durée d'exécution de la tâche T_i ($v_i = C_i$) ou d'une densité de valeur ($v_i = \frac{V_i}{C_i}$).

La *fonction valeur* peut en revanche être une fonction du temps. Dans un système temps réel, la valeur d'une tâche dépend à la fois de sa durée d'exécution et de son urgence.

On distingue donc dans la suite les cas des tâches non temps réel de ceux relatifs aux tâches temps réel à contraintes strictes, fermes ou relatives [JLT85].

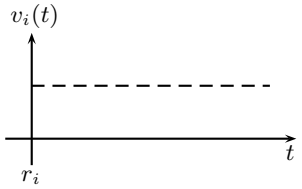

FIG. II.7 – Exemple de fonction valeur associée à une tâche non-TR

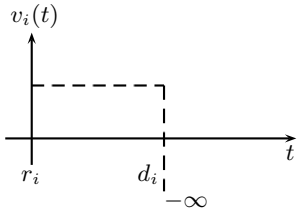

FIG. II.8 – Exemple de fonction valeur associée à une tâche TRCS

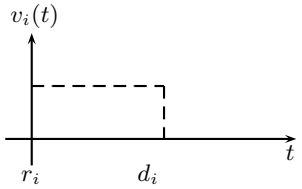

FIG. II.9 – Exemple de fonction valeur associée à une tâche TRCF

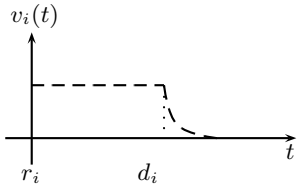

FIG. II.10 – Exemple de fonction valeur associée à une tâche TRCR

Dans le cas des tâches non temps réel (cf. Figure II.7), le résultat apporte le même bénéfice quel que soit l'instant auquel il est produit. Pour des tâches temps réel à contraintes strictes (cf. Figure II.8), passée la date d_i, le résultat n'a plus à être produit pour ne pas amputer la performance du système. Si l'on considère les tâches à contraintes fermes (cf. Figure II.9), le résultat n'apporte rien s'il est produit après la date d_i. Quant aux tâches temps réel à contraintes relatives (cf. Figure II.10), passée la date d_i, on peut observer par exemple une décroissante exponentielle.

Lorsqu'il y a surcharge, le critère d'importance prévaut systématiquement sur le critère d'urgence et ce sont les tâches pour lesquelles la valeur de la *fonction valeur* est la plus élevée, qui sont garanties en priorité à tout instant.

3.2 Métriques associées à la fonction valeur

3.2.1 La valeur cumulative

Le but poursuivi par la politique de contrôle de charge basée sur la valeur consiste à *maximiser un cumul* noté Γ représentant la somme des importances des tâches garanties. Les tâches sont donc éliminées de façon à obtenir un Γ maximum. La performance d'un algorithme d'ordonnancement A est exprimée par sa *valeur cumulative* Γ_A (*cumulative value* ou *completion count*) définie comme suit [BSS95] :

$$\Gamma_A = \sum_{i=1}^{n} v_i(t) \tag{II.8}$$

où $v_i(t)$ représente la valeur apportée au système par la terminaison de la tâche T_i à l'instant t.

On remarquera que si une tâche temps réel à contraintes strictes viole son échéance alors $v_i(t) = -\infty$ et donc $\Gamma_A = -\infty$ et ce, même si les autres tâches de l'application s'accomplissent dans le respect de leur échéance.

Cette métrique considérée, un algorithme d'ordonnancement est dit *optimal* s'il maximise la valeur cumulative réalisable pour un ensemble de tâches donné. Par conséquent, la valeur cumulative obtenue par un algorithme d'ordonnancement optimal est définie par :

$$\Gamma^* = \max_A \Gamma_A \tag{II.9}$$

En condition de surcharge, il n'y a *aucun* algorithme en ligne optimal capable de garantir une valeur cumulative égale à Γ^*, d'où la nécessité d'introduire la notion de *facteur de compétitivité* [SSR+98].

3.2.2 Le facteur de compétitivité

La valeur cumulative Γ^* ne peut être obtenue que par un algorithme d'ordonnancement clairvoyant[1] optimal. Bien que cet algorithme clairvoyant soit une pure abstraction théorique, il peut être utilisé comme modèle de référence pour évaluer la performance des algorithmes d'ordonnancement en-ligne réels, en conditions de surcharge.

Soit Γ^*, la valeur cumulative maximale précédemment définie réalisable par un algorithme clairvoyant optimal. Un algorithme A a un *facteur de compétitivité* φ_A, s'il garantit que, *pour n'importe quel ensemble de tâches*,

$$\Gamma_A \leq \varphi_A \Gamma^* \tag{II.10}$$

Par conséquent, $\varphi_A \in [0, 1]$ peut être calculé de la manière suivante :

$$\varphi_A = \min \frac{\Gamma_A}{\Gamma^*} \tag{II.11}$$

Ainsi, étant donnée n'importe quelle configuration de tâches, l'algorithme A peut atteindre une valeur cumulative Γ_A qui correspond *au moins* à φ_A fois la valeur cumulative obtenue par un algorithme clairvoyant optimal. Plus le facteur de compétitivité est élevé, meilleur est l'algorithme.

3.3 Les limites théoriques des ordonnanceurs en-ligne

Un résultat théorique important sur la compétitivité des algorithmes d'ordonnancement temps réel en-ligne a été apporté par [BKM+91]. Baruah *et al.* ont prouvé qu'il existait une borne supérieure pour le facteur de compétitivité de *n'importe quel* algorithme en ligne.

Théorème 19 *Dans les systèmes où le facteur de charge est supérieur à 2 ($\rho > 2$; situation de surcharge) et où les valeurs associées aux tâches sont proportionnelles à leur durée d'exécution, aucun algorithme d'ordonnancement en-ligne ne peut garantir un facteur de compétitivité supérieur à 0.25.*

Cependant, il est important de savoir que la borne supérieure est obtenue sous des hypothèses très restrictives : toutes les tâches ont une laxité égale à zéro, les durées d'exécution des tâches peuvent être arbitrairement petites, et la surcharge a une durée arbitraire (mais finie). Dans la plupart des applications, les caractéristiques des tâches sont moins restrictives. Par conséquent, la borne de 0.25 est un repère uniquement théorique comme l'illustre le théorème suivant [BKM+91].

Théorème 20 *Même si le système est seulement très légèrement surchargé (utilisation processeur très légèrement supérieure à 100%), 0.385 représente une borne supérieure pour le facteur de compétitivité de tout algorithme en-ligne.*

[1] Un ordonnancement *clairvoyant* est une abstraction théorique. Il s'agit d'un ordonnancement en ayant une connaissance a priori de tous les paramètres des tâches.

En règle générale, la borne supérieure du facteur de compétitivité est une fonction de la charge et varie de la manière suivante :

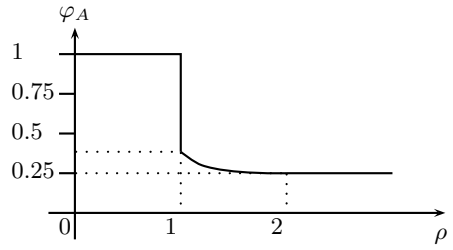

FIG. II.11 – Evolution du facteur de compétitivité en fonction du facteur de charge

3.4 Quelques algorithmes basés sur la valeur

3.4.1 Les algorithmes de type best-effort

Les premières approches basées sur la valeur [JLT85, Loc86] sous la dénomination d'ordonnancement *BE* (*Best-Effort*) ou *LBESA (Locke's Best-Effort Scheduling Algorithm*) ont été développées tout particulièrement pour l'ordonnancement de tâches à contraintes relatives. L'algorithme *BE* utilise l'approche *EDF* toutes les fois que l'ensemble des tâches courantes est ordonnançable (c'est-à-dire, *non surchargé*), et procède dans le cas contraire à un allègement de la charge par l'évaluation des *densités de valeurs* associées aux tâches. L'algorithme *BE* a été évalué en utilisant une variété de fonctions de valeur dépendant du temps [RMP03]. Des variantes, *BE-h* et *BE-v* sont décrites dans [MPR99]. D'autres algorithmes comme l'algorithme *DP* (*Dynamic Priority*) [NT87] ou l'algorithme *DASA* (*Dependant Activity Scheduling Algorithm*) [Cla90] sont également basés sur un calcul de densité de valeur pour traiter les cas de surcharge. Citons également l'algorithme *BEST* (*Best-effort scheduler Enhanced for Soft real-time Time-sharing*) qui s'affranchit de la connaissance *a priori* de la charge du système [BB01, BB02].

3.4.2 Les algorithmes de type garantie

Au niveau des modèles d'ordonnancement à garantie, Buttazzo et Stankovic proposent dans [BS93] l'algorithme *GED* (*Guarantee Earliest Deadline*) basé sur l'algorithme *EDF*. Son originalité repose sur l'introduction d'un test d'ordonnançabilité conditionnant l'acceptation de la tâche dans la liste d'exécution. Les résultats existants sur les garanties de performance de *EDF* sont limités aux seuls cas des systèmes non surchargés. Lam et To dans [LT01] étudient l'algorithme *GED* appelé *EDF-ac* (*EDF with admission control*) en montrant que l'on peut atteindre, pour les systèmes surchargés,

des garanties de performance similaires à celles rencontrées dans les systèmes non surchargés, sous certaines conditions. Citons également l'approche présentée dans [BLA98] où l'on considère un modèle de tâches *élastiques* vis-à-vis de leur période P_i nominale. Grâce à cette garantie élastique, selon les variations de charge du système, une tâche qui aurait été rejetée par un ordonnanceur rigide, peut être acceptée dans ce modèle moyennant une augmentation des périodes des autres tâches. Le modèle proposé permet ainsi de faire face à des situations de surcharge et ceci, de manière flexible. L'avantage principal de ce modèle repose sur la politique de sélection de la solution implicitement *encodée* dans les coefficients élastiques fournis par l'utilisateur et choisis selon le profil de l'application [BA02].

3.4.3 Les algorithmes de type robuste

Plusieurs algorithmes [BS93] proposés par Buttazzo et Stankovic en 1993, contribuent à gérer les situations de surcharge en utilisant un schéma d'ordonnancement robuste. Les auteurs introduisent la notion de *tolérance d'échéance* (*deadline tolerance*) définie par la durée durant laquelle une instance peut continuer à s'exécuter après son échéance en produisant toujours un résultat valable. On distingue plusieurs versions robustes selon la politique d'assignation des priorités adoptée [BSS95] : *RED* (*Robust Earliest Deadline*), *RHVF* (*Robust Highest Value First*), *RHDF* (*Robust Highest Density First*), *RMIX* (*Robust MIXed rule*). Une extension de l'algorithme RED procédant à des rejets de tâches multiples, appelé *MED* (*Robust EDF with Multiple running task rejection*), a également été proposée. La différence réside dans le fait que, si un nouvel ensemble de tâches n'est pas ordonnançable et que la tâche qui vient d'arriver est critique alors, le système peut rejeter plus d'une tâche de moindre importance de façon à rendre l'ensemble de tâches résultant ordonnançable. Une autre solution robuste est celle de Baruah et Haritsa qui proposent dans [BH93] un algorithme en-ligne nommé *ROBUST* (*Resistance to Overload By Using Slack Time*). Celui-ci tire avantage de la laxité des tâches pour fournir une solution améliorant les performances en cas de surcharge. Des travaux basés sur une heuristique intéressante sont par ailleurs rapportés dans [BKM+91]. Ils portent sur l'algorithme D^* qui en cas de surcharge abandonne la tâche active si et seulement si la valeur obtenue en exécutant la nouvelle tâche active est supérieure à la valeur cumulée de toutes les tâches abandonnées depuis la dernière date à laquelle une tâche a satisfait son échéance. Dans [KS92], une solution (l'algorithme D^{over}) améliorant l'algorithme D^* au niveau de la structure de données utilisée et au niveau du test mis en œuvre en cas de surcharge est exposée. Enfin d'autres approches robustes basées sur la valeur diffèrent par le modèle de tâches considéré. Citons le modèle *MIXED* et sa généralisation GENMIXED [CEN+02], ou encore le modèle *OR-ULD* (*Overload Resolution using Utility Loss Density* [HTS00] qui s'appuie sur le modèle du calcul imprécis [LNL87, LLN87].

4 Les approches à tâches bipartites

4.1 Le modèle du Mécanisme à échéance (Deadline Mechanism)

4.1.1 Description

Cette méthode consiste à implanter chaque tâche en *deux versions* [CAM79]. Une version dite *Primaire* produit un résultat de bonne qualité de service mais son temps d'exécution n'est, à priori, pas connu. Une version dite *Secondaire* fournit un résultat plus dégradé en termes de QoS mais dans un temps borné. L'algorithme d'ordonnancement est alors conçu de manière à ce que l'ensemble des tâches de l'application respectent leurs contraintes temporelles, soit en exécutant le primaire, soit en faisant appel au secondaire. On distingue deux politiques d'ordonnancement basées sur ce principe [Sil86] :

- *la technique de la première chance* : elle consiste à garantir d'abord une exécution fiable à toutes les versions secondaires avant de tenter l'exécution de leurs versions primaires respectives. On remarquera donc que cette politique revêt un aspect préventif.
- *la technique de la dernière chance* : elle revient à déterminer la date de début d'exécution au plus tard de chaque version secondaire garantissant une exécution fiable. Ceci libère du temps au plus tôt mis à profit pour tenter l'exécution des versions primaires. Cette technique consiste donc d'abord à tenter d'exécuter le primaire et en cas de réussite, le secondaire n'est pas exécuté.

4.1.2 Les algorithmes basés sur le mécanisme à échéance

Campbell *et al.* ont proposé une stratégie *sous-optimale* dans [CAM79] basée sur un ordonnancement statique non-préemptif, avec une hypothèse très restrictive sur la durée d'exécution des secondaires, à savoir que la somme des durées d'exécution de toutes les tâches doit être inférieure ou égale à la plus petite période. Les essais en simulation rapportés dans [Sil86, Clo88] ont montré que la technique de la dernière chance fournit de meilleurs résultats que celle de la première chance vis-à-vis du pourcentage de primaires réussis.

Dans [LC86], Liestman et Campbell ont introduit un algorithme qui améliore l'approche précédente dans le sens où celui-ci est *FT-optimal*, c'est-à-dire qu'il maximise le nombre d'instances primaires qui réussissent leur exécution.

Un modèle plus général a été étudié dans [CCE88]. L'algorithme de H. Chetto et M. Chetto consiste ici à juxtaposer un ordonnanceur des primaires, un ordonnanceur des secondaires et un algorithme de décision pour commuter de l'un vers l'autre. L'algorithme global d'ordonnancement des primaires et des secondaires est *optimal* au sens où le temps creux réservé pour l'exécution des primaires est à tout instant maximal, et ceci quel que soit l'ordonnancement choisi pour les primaires. Contrairement à [CAM79], aucune condition sur la configuration des secondaires n'est nécessaire puisque l'agorithme d'ordonnancement de ces derniers est préemptif. D'autre part, le choix des périodes est entièrement libre alors que le modèle [LC86] nécessite d'avoir des périodes multiples.

4.2 Le modèle du Calcul imprecis (Imprecise Computation)

La technique du calcul imprécis [LNL87, LLN87] a également été proposée pour gérer les cas de surcharge passagère et améliorer la tolérance aux fautes des systèmes temps réel. Dans un système basé sur cette technique, chaque tâche critique est conçue de manière à pouvoir produire "à temps" un résultat approximatif exploitable, même si une surcharge du système l'empêche de fournir le résultat précis désiré. Dans ce modèle, chaque tâche est logiquement décomposée en une partie obligatoire et une partie optionnelle. La partie *obligatoire* représente la quantité minimale d'exécution nécessaire pour obtenir un résultat acceptable ; du temps de traitement additionnel contribuant à affiner ce résultat est représenté par la *partie optionnelle*. La partie obligatoire doit s'exécuter dans le respect de son échéance.

Le *modèle de tâche* considéré dans la suite est le suivant [LLB+94] : chaque tâche T_i (date de réveil r_i, échéance d_i, durée d'exécution τ_i, poids w_i) est logiquement décomposée en deux tâches, la tâche obligatoire M_i et la tâche optionnelle O_i. Les dates de réveil et les échéances des tâches M_i et O_i sont celles de T_i, et M_i est le prédécesseur immédiat de O_i. Soient m_i et o_i, les durées d'exécution de M_i et O_i, respectivement. m_i et o_i sont des nombres rationnels tels que $m_i + o_i = \tau_i$.

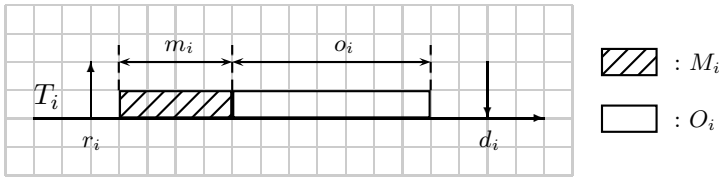

FIG. II.12 – Modèle de tâches pour le calcul imprécis

De nombreuses implantations ont été proposées dont le but est de maximiser le temps processeur disponible pour les traitements des parties optionnelles des tâches. La plupart reposent sur la mesure de l'erreur associée à toute tâche T_i définie par la quantité de temps non exécutée parmi les parties optionnelles. L'objectif des algorithmes d'ordonnancement est alors de minimiser soit l'erreur totale [SL95, LLB+94], l'erreur moyenne [CLL90, LLB+94], l'erreur pondérée totale [LW90], l'erreur normalisée maximale [HLW92], ou le nombre de tâches en retard [LW90].

4.3 Le modèle à transformation de tâche (Transform-task)

Ce modèle [TDS+95] a pour objectif de fournir une *garantie absolue* aux instances de faible durée d'exécution, et une *garantie probabiliste* aux autres instances. Il repose sur un ensemble de N tâches *semi-périodiques* ordonnancées sur un seul processeur. Chaque tâche T_i est qualifiée de *semi-périodique* car par définition, elle est divisée en une *tâche périodique* $P_i(p_i, r_i, c_i)$ et *une tâche sporadique* S_i. La *probabilité d'arrivée* A_i d'une requête sporadique $S_{i,j}$ dans la j^{eme} période de T_i est égale à la probabilité que $c_{i,j} > c_i$ (la durée d'exécution de la j^{ieme} instance de T_i excède la durée d'exécution

fixée au départ pour la tâche). Les paramètres c_i et A_i sont définis de manière à assurer l'ordonnançabilité des tâches périodiques.

La tâche périodique P_i peut être ordonnancée selon une politique à priorité fixe (*Rate-Monotonic*) ou dynamique (*Earliest Deadline First*). Un ou plusieurs serveurs de tâches apériodiques assurent l'ordonnancement des requêtes apériodiques de S_i.

4.4 Le modèle à Exception (TaskPair)

Le modèle d'ordonnancement TaskPair (*TaskPair-Scheduling ou TPS*) repose sur un modèle à bipartition de tâches. Une tâche TP est définie par un couple de 2 versions (*MainTask*, *ExceptTask*). La tâche *ExceptTask* est à contraintes strictes (tâche de plus grande importance) tandis que la tâche *MainTask* est une tâche d'importance relative (basée sur un critère arbitraire). L'algorithme en-ligne proposé par Streich dans [Str94] exécute la tâche *MainTask* seulement si l'ordonnanceur *garantit* que l'une des 2 versions de la tâche s'exécutera dans le respect de son échéance. L'ordonnanceur doit avoir connaissance du *WCET* de la tâche *ExceptTask* ainsi que du *facteur de charge actuel* (à savoir celui prévu pour les tâches garanties). La manière la plus simple d'implémenter l'algorithme TaskPair est d'exécuter les tâches *Maintask* selon une politique round-robin préemptive, en leur affectant une faible priorité. Quant aux tâches *ExceptTask*, elles sont exécutées au plus tard avec une priorité maximale.

4.5 Le modèle INCA

Le *serveur d'ordonnancement incrémental* INCA [MMM00] est une extension de l'algorithme *Earliest Deadline First (EDF)*. Il travaille sur le modèle de tâches du calcul imprécis et toutes les tâches sont supposées périodiques. En réponse à des surcharges passagères, le serveur *INCA ajuste* la charge du système en exécutant une séquence de calculs *approximatifs*, $AP(0), ..., AP(n)$, l'algorithme $AP(i)$ fournissant une solution plus proche de l'optimal que $AP(i-1)$ mais en un temps plus long. Un test d'ordonnançabilité (*UBT : Utilization-Based Test*) est chargé de détecter une surcharge causée par l'occurrence d'une nouvelle tâche dans le système. A partir de ce moment, la solution fournie par $AP(0)$ permet à l'ordonnanceur d'éliminer certaines tâches optionnelles. Le surplus de temps processeur introduit par la suppression de la surcharge est utilisé par l'ordonnanceur pour exécuter les algorithmes approximatifs $AP(k)$ de façon à affiner la qualité de la solution. Le serveur *INCA* stoppe son exécution lorsqu'il n'y a plus de temps processeur disponible ou que le résultat de $AP(k)$ n'est pas meilleur que celui de $AP(k-1)$. Les algorithmes $AP(k)$, pour $k = 1...n$, sont décrits dans [MMM00]. Les résultats de simulation présentés dans [MMM00] montrent que le serveur INCA est un mécanisme *efficace* et *peu coûteux* pour ordonnancer des tâches temps réel dans des conditions de surcharge.

5 Les approches à pertes contraintes

5.1 Le modèle Skip-Over

5.1.1 Description

Dans ce modèle, chaque tâche périodique $T_i(C_i, P_i, s_i)$ est caractérisée par une durée d'exécution dans le pire cas C_i, une période P_i, une échéance relative égale à sa période, et un paramètre de perte s_i, $2 \leq s_i \leq \infty$, qui fournit la tolérance de la tâche à manquer ses échéances. Selon la terminologie introduite par Koren et Shasha dans [KS95], chaque instance de tâche peut être *rouge* (*red*) ou *bleue* (*blue*). Une instance de tâche *rouge* doit être accomplie avant son échéance ; une instance de tâche *bleue* peut être abandonnée à tout moment. Une configuration de tâches est dite *profondément rouge* si l'ensemble des instances réveillées à $t = 0$ sont *rouges*.

Le modèle répond à un certain nombre de règles élémentaires que nous énonçons ci-après :

- une perte survient lorsqu'une instance de tâche bleue atteint sa date d'échéance alors qu'elle n'a pas terminé son exécution.
- la distance entre 2 pertes consécutives doit être d'au moins s_i périodes.
- si une instance de tâche *bleue* est abandonnée, les $s_i - 1$ prochaines instances de la tâche doivent être *rouges*.
- si une instance de tâche *bleue* s'accomplit en respectant son échéance, la prochaine instance de la tâche reste *bleue*.
- les $s_i - 1$ premières instances de chaque tâche doivent être *rouges*.

Ainsi, $s_i \geq 2$ implique que, si une instance de tâche *bleue* a manqué son échéance alors, la prochaine occurrence de la même tâche doit être *rouge*. Lorsque $s_i = \infty$, aucune perte n'est autorisée et l'on retrouve le modèle classique d'une tâche périodique en environnement TRCS.

A titre d'illustration, nous pouvons considérer la figure II.13 :

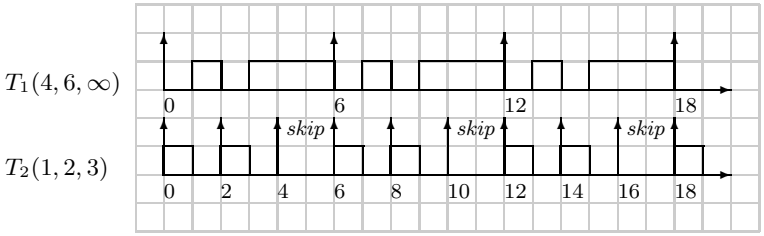

FIG. II.13 – Exemple de pertes au sens Skip-Over

Le système est surchargé ($U_p = \sum_{i=1}^{n} \frac{C_i}{P_i} = \frac{4}{6} + \frac{1}{2} = 1.17$), mais les tâches peuvent être ordonnancées si T_2 perd une instance toutes les 3 instances.

5.1.2 Les algorithmes d'ordonnancement Skip-Over

Le premier algorithme décrit dans [KS95] est l'algorithme *RTO* (*Red Tasks Only*). Cet algorithme rejette systématiquement les tâches *bleues*. Les tâches *rouges* sont ordonnancées suivant EDF. RTO "perd" des échéances selon un mode régulier, à savoir que la distance entre 2 pertes est exactement s_i périodes. Un autre algorithme plus flexible est l'algorithme *BWP* (*Blue When Possible*) qui exécute des tâches bleues lorsque cela n'empêche pas les tâches rouges de respecter leurs échéances. L'algorithme évolue ainsi : Ordonnancer les tâches rouges selon *EDF*. S'il n'y a aucune tâche rouge prête, alors activer une tâche bleue. S'il y a plus d'une tâche bleue prête alors en choisir une selon une *heuristique* donnée [KS95], par exemple, n'importe laquelle ou la tâche bleue dont l'échéance est la plus lointaine, etc. Ces deux algorithmes sont détaillés et illustrés dans le chapitre suivant car ils constituent une base de ce travail de thèse.

5.1.3 Conditions de faisabilité

Koren et Shasha [KS95] ont prouvé que le problème de la faisabilité d'un ensemble de tâches périodiques autorisant des pertes au sens Skip-Over est NP-difficile. Cela signifie que la seule possibilité de trouver un ensemble ordonnançable est l'énumération de toutes les séquences d'ordonnancement possibles, ce qui n'est pas réalisable dans le cas général. Les auteurs ont montré que la condition suivante est une condition *nécessaire* de faisabilité d'un ensemble $\Gamma = \{T_i(P_i, C_i, s_i)\}$ de tâches périodiques tolérantes aux pertes :

$$\sum_{i=1}^{n} \frac{C_i(s_i - 1)}{P_i s_i} \leq 1 \tag{II.12}$$

Par ailleurs, Caccamo et Buttazzo [CB97] ont introduit la notion de *facteur d'utilisation équivalent* (cf. Définition 9) qui représente la charge imposée au processeur dans le mode de fonctionnement le plus dégradé (seules les tâches rouges sont exécutées).

Définition 9 *Etant donné un ensemble* $\Gamma = \{T_i(P_i, C_i, s_i)\}$ *de n tâches périodiques autorisant des pertes au sens Skip-Over, le facteur d'utilisation équivalent du processeur est défini par :*

$$U_p^* = \max_{L \geq 0} \frac{\sum_i D(i, [0, L])}{L} \tag{II.13}$$

où

$$D(i, [0, L]) = (\lfloor \frac{L}{P_i} - \frac{L}{P_i s_i} \rfloor) C_i. \tag{II.14}$$

Les mêmes auteurs ont fourni une condition *suffisante* dans [CB98] garantissant la faisabilité d'une configuration de tâches Skip-Over :

Théorème 21 *Un ensemble* Γ *de tâches périodiques tolérantes aux pertes au sens Skip-Over est ordonnançable si* $U_p^* \leq 1$.

5.2 Le modèle (m,k)-firm

5.2.1 Description

La notion d'échéance (m,k)-firm a été introduite par Hamdaoui et Ramanathan dans [HR95]. L'ordonnancement à garantie *(m,k)-firm* est une technique de gestion de surcharge qui écarte un certain nombre d'instances de tâches de manière à ce que le système reste dans un état sécuritaire. Ceci se ramène alors au problème de la minimisation de la probabilité pour toute tâche de violer son échéance. Considérons un système avec n tâches temps réel indépendantes. Chaque tâche T_i 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 [HR95] : *normal* ou *échec dynamique*. Une tâche temps réel sous contrainte *(m,k)-firm* est dite en *état d'échec dynamique* à un instant t, si à cet instant il existe plus de $(k-m)$ instances ayant raté leurs échéances parmi les k dernières invocations de la tâche. Par extension, un système est dit en *état d'échec dynamique* à un instant t, si au moins une des tâches est en échec dynamique à cet instant.

Pour chaque tâche, le système maintient un historique récent des échéances respectées ou manquées. L'historique pour la tâche T_j est maintenu grâce à une k-séquence $(\delta^j_{i-k+1}, ..., \delta^j_{i-1}, \delta^j_i)$, où δ^j_i représente le statut de la dernière instance de T_j (la i^{eme}) servie ou rejetée. Plus formellement, δ^j_i est une variable aléatoire binaire dénotant l'état de la i^{eme} instance :

$$\delta^j_i = \begin{cases} 0, & \text{si la } i^{eme} \text{ instance de } T_j \text{ viole son échéance} \\ 1, & \text{sinon} \end{cases} \tag{II.15}$$

Ce concept est bien sûr similaire à celui introduit par Koren et Shasha [KS95]. En effet on peut considérer le modèle Skip-Over comme un cas particulier du modèle (m,k)-Firm où $m = k - 1$. On peut également remarquer que cette notion d'échéances *(m,k)-firm* est une généralisation des échéances *firm* et *soft*. Aussi, une tâche à contraintes strictes peut-elle être représentée par une échéance (1,1)-firm. De même, une échéance (1,2)-firm établit la contrainte suivante : la tâche ne doit pas manquer 2 instances consécutives.

5.2.2 Les algorithmes d'ordonnancement (m,k)-firm

Un premier algorithme à priorité dynamique appelé *DBP* (*Distance Based Priority*) a été proposé par Hamdaoui et Ramanathan dans [HR95]. L'idée de cet algorithme est d'assigner une priorité aux tâches basée sur l'état courant de leurs instances respectives. Comme son nom l'indique, DBP définit la notion de *distance* représentée par le nombre consécutifs de bits '0' que l'on doit rajouter à la k-séquence pour atteindre l'état d'échec dynamique. Plus cette distance est petite, plus la priorité affectée à la tâche est élevée. En d'autres termes, plus une tâche est proche d'un état d'échec dynamique, plus sa priorité sera élevée. Par exemple [Kou04], si une tâche est définie sous contrainte (3,5)-firm et qu'elle affiche à un instant donné une k-séquence {11011}, alors sa priorité sera égale

à 2 (si les 2 prochaines instances de la tâche échouent, alors la k-séquence sera égale à {011$\underline{00}$}). Le problème majeur de cette approche réside dans le fait que la contrainte *(m,k)-firm* appliquée aux tâches temps réel, prévaut sur les autres paramètres temporels tels que l'échéance, le temps d'inter-arrivée, le temps d'exécution ainsi que l'importance relative entre les instances des différentes tâches [SK03]. C'est pourquoi dans [PSK+03], les auteurs proposent une amélioration de l'algorithme *DBP* appelée *Matrix-DBP* qui consiste à construire une matrice NxN (N : nombre de tâches périodiques) juxtaposant tous ces paramètres à la contrainte *(m,k)-firm* inhérente au modèle de départ.

Par ailleurs, Ramanathan présente dans [Ram99] un algorithme à priorité fixe reposant sur le modèle (m,k)-firm : *EFP* (*Enhanced Fixed Priority*). La solution proposée est une combinaison des idées issues d'une part de l'approche du calcul imprécis [CLL90] et d'autre part, de la politique d'ordonnancement Rate-Monotonic [LL73]. C'est ainsi que pour s'appliquer au modèle *(m,k)-firm*, chaque tâche est découpée en m instances *obligatoires* (dites critiques) et (k-m) instances *optionnelles*. Ainsi, en cas de surcharge, l'ordonnanceur est en mesure de rejeter une instance optionnelle. Les instances critiques sont ordonnancées selon RM tandis que les instances optionnelles sont servies selon la politique FIFO. Par définition, la 1ère instance de chaque tâche est marquée critique. L'inconvénient de cet algorithme réside principalement dans le fait que la technique de marquage des instances obligatoires ne dépend que du rapport $\frac{m_i}{k_i}$ mais pas de C_i et P_i, ce qui dans certaines situations peut ne pas être optimal.

Quan et Hu ont donc apporté des améliorations à cet algorithme d'origine dans [QH00] en proposant une heuristique (*EFP-amélioré*). Celle-ci vise à caractériser et quantifier les interférences des instances obligatoires entre elles (préemptions, blocages), de manière à les réduire et ainsi exploiter au mieux les contraintes (m,k)-firm.

5.3 Le modèle (m,k)-hard

Cette notion a été proposée par Bernat et Burns [BB97]. Une contrainte *(m,k)-hard* est une contrainte temporelle stricte dans le sens où une tâche est considérée en échec si moins de m échéances sur k activations *successives* de la tâche sont respectées. L'approche consiste à utiliser un schéma d'exécution à double priorité (*Dual priority* [DW95]) qui repose 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*). Cette approche nécessite que les contraintes temporelles des tâches soient garanties par des tests d'ordonnançabilité hors-ligne.

5.4 Le modèle (p+i,k)-firm

L'échéance *(p+i,k)-firm* introduite par C. Montez et J. Fraga dans [MF02] est une extension de l'échéance *(m,k)-firm* qui inclut quelques-unes des propriétés du calcul imprécis décrites dans [CLL90]. Une tâche comporte une partie obligatoire (*imprécise*) et une partie optionnelle (*précise*). p exécutions précises et i exécutions imprécises sur k invocations de la tâche doivent être réalisées. En plus de l'attribution des priorités, un *test d'acceptation de précision* sélectionne la version (précise ou imprécise) de la tâche

qui sera exécutée. Les politiques *d'assignation des priorités* et *d'acceptation de précision* sont détaillées dans [CLL90].

5.5 Le modèle RBE (Rate-Based Execution)

Le modèle RBE (Rate-Based Execution) [JG99] considère des contraintes exprimées en termes d'unités de temps plutôt qu'en termes d'échéances. Une tâche RBE est ainsi uniquement caractérisée par un 4-tuple (x_i, y_i, d_i, c_i) de constantes entières où l'on note par :

- x_i : le nombre maximum d'instances susceptibles d'être réveillées dans tout intervalle de longueur y_i,
- y_i : un intervalle de temps,
- d_i : l'échéance relative de la tâche,
- c_i : sa durée d'exécution dans le pire cas.

L'instance J_{ij} d'une tâche réveillée à t_{ij} doit s'exécuter avant une échéance $D_i(j)$ donnée par la relation de récurrence suivante :

$$D_i(j) = \begin{cases} t_{ij} + d_i & \text{si } 1 \leq j \leq x_i \\ \max(t_{ij} + d_i, D_i(j - x_i) + y_i) & \text{si } j > x_i \end{cases} \tag{II.16}$$

Sous l'hypothèse de l'utilisation de cette fonction d'affectation d'échéance dans le modèle RBE, Jeffay and Godard dans [JG99] montrent que EDF est une politique *optimale* pour l'ordonnancement préemptif, l'ordonnancement non-préemptif, et l'ordonnancement en présence de ressources partagées.

5.6 Le modèle DWCS (Dynamic Window Constrained Scheduling)

Le modèle DWCS [WS99] a été défini à l'origine en tant que discipline de service pour l'ordonnancement de paquets réseau, avec pour principal objectif de maximiser le taux d'utilisation du serveur réseau, en présence de multiples paquets possédant des contraintes temporelles de type (m,k)-firm. Nous présentons ici le modèle étendu [WGS01] dans le sens où DWCS est considéré comme un algorithme d'*ordonnancement de tâches*. Chaque tâche possède ainsi deux attributs : une *échéance* D_i, qui représente la date au plus tard à laquelle la tâche doit *commencer* à s'exécuter, et une *tolérance de pertes* spécifiée par le rapport $\frac{x_i}{y_i}$, où x_i représente le nombre d'instances pouvant être abandonnées ou exécutées tardivement pour toute fenêtre de y_i instances à exécuter pour une tâche τ_i. Il est important de noter qu'à chaque instant, toutes les instances d'une même tâche possèdent la même tolérance aux pertes et sont ordonnancées selon leur ordre d'arrivée. A chaque fois qu'une instance de tâche T_i se réveille, la tolérance aux pertes pour toutes les instances de T_i, est ajustée de manière à refléter l'importance relative de l'exécution de cette instance.

5.7 Le modèle Weakly-hard

Un inconvénient majeur des algorithmes précédemment décrits réside dans le fait que les contraintes *(m,k)-firm* sont garanties seulement sur des fenêtres de temps fixées et par conséquent, des échecs dynamiques peut être rencontrés si l'on considère des fenêtres consécutives (glissantes). De plus, ces modèles sont assez restrictifs dans le sens où ils nécessitent que toutes les tâches possèdent la même fenêtre m et que $D_i = P_i$. Bernat *et al.* [BBL01] introduisent la notion de contraintes temps réel *weakly-hard* (littéralement *faiblement sévères*) comme une généralisation du concept *(m,k)-firm*, dans le but d'étendre la réflexion à d'autres types de modèles à pertes que le modèle classique de tâches. Ainsi, un système temps réel est dit *weakly-hard* s'il peut tolérer que certaines échéances ne soient pas respectées, pourvu que leur nombre soit borné et garanti hors-ligne. Les auteurs soulignent la nécessité d'introduire la notion de consécutivité au niveau de la spécification de la tolérance aux pertes appliquée aux différentes tâches. Cette réflexion conduit à la définition de 4 types de contraintes ($n \geq 1$, $n \leq m$) :

- une tâche τ "respecte n échéances parmi m", noté $\binom{n}{m}$, si, sur toute fenêtre de m requêtes consécutives, il y a au moins n requêtes dans n'importe quel ordre qui respectent leurs échéances,

- une tâche τ "respecte n échéances consécutives parmi m", noté $\langle \begin{smallmatrix} n \\ m \end{smallmatrix} \rangle$, si, sur toute fenêtre de m requêtes consécutives, il y a au moins n requêtes *consécutives* qui respectent leurs échéances,

- une tâche τ "viole n échéances parmi m", noté $\overline{\binom{n}{m}}$, si, sur toute fenêtre de m requêtes consécutives, il n'y a pas plus de n requêtes dans n'importe quel ordre qui violent leurs échéances,

- une tâche τ "viole n échéances consécutives parmi m", noté $\overline{\langle \begin{smallmatrix} n \\ m \end{smallmatrix} \rangle}$, si, sur toute fenêtre de m requêtes consécutives, il n'y a jamais n requêtes *consécutives* qui violent leurs échéances.

Le modèle *weakly-hard* consiste en un ensemble $\Pi = \{T_i = (P_i, C_i, D_i, \lambda_i)\}, 1 \leq i \leq N$ de N tâches périodiques, où chaque tâche T_i est caractérisée par une période P_i, une durée d'exécution C_i, une échéance relative D_i, et une contrainte *weakly-hard* λ_i choisie parmi celles décrites précédemment.

5.8 Le modèle MC (Markov Chain)

Hu *et al.* dans [HLL+03] propose un modèle à QoS basée sur une chaîne de Markov (*Markov Chain*) pour décrire le comportement stochastique désiré en termes de pertes associées à une tâche T_i. Cette *MC*-contrainte de T_i est notée $\mathcal{MC}_i$. Plus précisément, $\mathcal{MC}_i$ est un processus stochastique discret possédant 2 états ou plus. La probabilité de transition d'un état à un autre représente soit la probabilité pour la prochaine instance

d'être sautée, soit au contraire, sa probabilité de réussite. Chaque état est représenté par une chaîne spécifique de bits d'état f_{ij} traduisant le comportement des instances précédentes : un '1' une instance réussie, un '0' une instance sautée.

6 Synthèse

Même si un système temps réel est conçu et dimensionné de façon appropriée, une surcharge temporaire de traitements peut survenir pour différentes raisons telles que, l'occurrence simultanée d'événements asynchrones ou le dysfonctionnement d'un périphérique [But97]. En effet, de part la nature dynamique de l'environnement avec lequel interagit un système temps réel, il est impossible de prévoir si le système sera surchargé ou non à un moment donné. Les conséquences les plus graves sont alors le non-respect des échéances des tâches temps réel à contraintes strictes, pouvant remettre en cause le comportement global du système. Etant donné que tous les systèmes possèdent des ressources limitées, leur capacité à exécuter un ensemble de tâches est elle-aussi limitée. Plus clairement, des conditions de surcharge surviennent dès lors que le système doit traiter un nombre de tâches plus important que ce que permet l'ensemble des ressources disponibles.

Ainsi, puisque que la charge et les lois d'occurrence des instances peuvent varier énormément, il est évident que des algorithmes adaptatifs sont requis, et que les algorithmes dédiés uniquement aux situations de sous-surcharge ou de surcharge, sont à écarter. Nous avons vu dans ce chapitre, qu'un certain nombre d'algorithmes d'ordonnancement ont été proposés dans ce sens.

Un premier courant [JNC+89] a introduit le concept de *fonction valeur* associant une valeur à chaque tâche en fonction de l'instant auquel son exécution s'achève. Ces algorithmes d'ordonnancement prennent en compte aussi bien les contraintes temporelles des tâches que le coût infligé au système suite à la violation d'une échéance. Leur objectif est d'ordonnancer les tâches les plus importantes en priorité, l'importance de chaque tâche étant reflétée, à tout instant, par sa *fonction valeur*. Beaucoup d'algorithmes d'ordonnancement ont ainsi été développés, mais en pratique, peu de travaux de recherche ont été menés pour fournir une mesure de leur performance (la solution proposée est-elle proche de l'optimal ?) et de leur complexité. Par ailleurs, dans la plupart des algorithmes élaborés, le critère de rejet en cas de surcharge, consiste à sélectionner les tâches de moindre valeur, ce qui peut conduire à une sous-utilisation des ressources et mener à des performances médiocres. L'inconvénient majeur de ces approches reposent en outre sur la difficulté du choix de *fonctions valeur* appropriées, traduisant au mieux l'importance relative des tâches. Enfin, Baruah et al. [BKM+91] ont montré qu'aucun algorithme en-ligne ne peut garantir une valeur cumulative (*cumulative value*) supérieure à $\frac{1}{4}$ de la valeur obtenue par un algorithme clairvoyant.

L'approche précédente suppose que le résultat fourni par une tâche n'a pas de valeur si celle-ci ne s'exécute pas complètement. Par opposition, dans les modèles à base de tâches bipartites, nous avons vu que le résultat d'une tâche possède toujours une certaine valeur, même si un résultat dégradé (Mécanisme à Echéance [CAM79]) ou approxima-

tif (Calcul Imprécis [LNL87, LLN87]) est produit. En cas de surcharge, les algorithmes d'ordonnancement reposant sur ces modèles, garantissent un niveau de QoS minimal, correspondant à l'exécution des parties obligatoires des tâches. Cependant, en cas de surcharge prolongée, certaines tâches peuvent afficher des exécutions dégradées répétées, ce qui peut détériorer fortement les performances du système [MF02]. Dans le cas du Calcul Imprécis, une heuristique d'ordonnancement a été proposée dans [CLL90] pour garantir à l'ensemble des tâches, au moins une exécution précise sur k activations successives.

La dernière approche de résolution de surcharge que nous avons présentée, se focalise sur le relâchement des garanties strictes des contraintes temporelles des tâches. Dans un environnement TRCF, une violation d'échéance n'a généralement pas de conséquences catastrophiques, et se traduit uniquement par une dégradation de la QoS. De plus, la violation occasionnelle de certaines échéances peut être tolérée en pratique, du fait du pessimisme relatif aux propriétés temporelles des tâches (durée d'exécution, instant d'occurrence, etc.). Ainsi, la gestion de surcharge dans cette dernière approche, repose sur l'abandon contrôlé de certaines instances de tâches, de manière à améliorer l'utilisation effective des ressources et à minimiser la dégradation causée par les instances abandonnées. Dans tous les algorithmes d'ordonnancement présentés, toutes les tâches périodiques du système doivent maintenir un nombre d'échéances violées en-deçà d'une valeur limite donnée, sous peine d'entraîner une défaillance au niveau du système. Par exemple, dans le modèle *(m,k)-firm* [HR95], chaque tâche périodique doit respecter au moins m échéances sur une fenêtre de k activations. Le nombre maximal de violations d'échéances tolérées est alors égal à $k - m$. Le problème de ce modèle est que la contrainte définie ici peut être respectée même dans le cas où un grand nombre d'instances consécutives violent leur échéance. Par exemple, considérons une contrainte (100,1000)-firm (soit 10% de pertes) apposée sur une tâche T_i. Cette contrainte sera satisfaite même si sur 1000 activations de T_i, 100 instances consécutives violent leurs échéances. En revanche, une spécification d'échéances *(9,10)-firm* (soit toujours 10% de pertes) assurera le même pourcentage de pertes mais présentera l'avantage d'espacer suffisamment les violations d'échéances. Cette spécification correspond à celle définie par le modèle Skip-Over [KS95] dans lequel une tâche est autorisée à violer une échéance parmi s_i invocations. La contrainte Skip-Over se ramène ainsi à celle d'une tâche d'échéance *($s_i - 1$,s_i)-firm*. Notons que le problème de la prise en compte de la consécutivité des violations d'échéances a également été résolu par Bernat et al. par la spécification de contraintes *Weakly-Hard* [BBL01]. Un des avantages de l'ensemble de ces approches réside dans la possibilité de modéliser des tâches à contraintes strictes. Celles-ci sont alors définies par une échéance *(1,1)-firm* ou bien par un paramètre $s_i = \infty$, selon le modèle de tâches envisagé. Plus généralement, les contraintes proposées permettent de spécifier de manière claire et intuitive, le nombre de violations d'échéances toléré sur un intervalle de temps donné. Enfin, ces modèles d'ordonnancement ont l'avantage de fournir un mécanisme de dégradation contrôlée de la QoS.

7 Conclusion

Nous nous sommes intéressés dans ce chapitre à la problématique de l'ordonnancement en présence de surcharge. Après avoir défini la notion de surcharge et énoncé ses principaux critères d'évaluation, nous avons présenté de manière non-exhaustive les différents modèles de résolution existants. Ceux-ci ont été regroupés en 3 classes distinctes, à savoir les modèles basés sur la notion de la valeur, ceux basés sur la définition de tâches bipartites, et ceux s'appuyant sur la spécification d'une contrainte de pertes. L'énumération de ces modèles de résolution a ensuite été complétée par une synthèse illustrant brièvement les avantages et inconvénients de chaque type d'approche.

Dans le chapitre suivant, nous nous intéressons à l'ordonnancement de tâches périodiques définies sous des contraintes de pertes. Les travaux effectués portent sur l'étude et l'amélioration des algorithmes d'ordonnancement Skip-Over.

Chapitre III

Ordonnancement sous contraintes de QoS

Ce chapitre est consacré à l'ordonnancement monoprocesseur de tâches périodiques définies sous des contraintes de QoS. En premier lieu, nous décrivons les ordonnanceurs du modèle Skip-Over basés sur EDF, à savoir les algorithmes RTO et BWP. Ensuite, nous exposons l'adaptation des équations de calcul des temps creux sous EDL aux modèles de tâches RTO et BWP. Puis nous présentons deux nouveaux algorithmes, RLP et RLP/T, basés sur un ordonnancemment EDL. Enfin, nous évaluons dans quelle mesure ces nouveaux algorithmes constituent une amélioration des algorithmes Skip-Over de base, dans le sens où ils tentent de maximiser la qualité de service (i.e., le nombre total d'instances de tâches s'exécutant dans le respect de leur échéance) observée au niveau du système.

---

1 L'ordonnancement Skip-Over basé sur EDF

Nous nous intéressons dans cette partie au problème de l'ordonnancement monoprocesseur de tâches périodiques autorisant *occasionnellement* le non-respect de leurs échéances. Nous supposons que les tâches sont à échéances sur requêtes, qu'elles sont préemptables et qu'elles ne possèdent pas de contraintes de précédence. Une tâche périodique $T_i(C_i, P_i, s_i)$ avec contraintes de QoS au sens Skip-Over [KS95] est alors caractérisée par une durée d'exécution au pire-cas C_i, une période d'activation P_i et un paramètre de tolérance aux pertes s_i. Rappelons que la distance entre deux pertes consécutives doit être au moins s_i périodes. Lorsque $s_i = \infty$, aucune perte n'est autorisée et la tâche T_i est équivalente à une tâche temps réel à contraintes strictes. Le paramètre de pertes s_i peut être vu comme une métrique de QoS (la qualité de service est d'autant plus élevée que la valeur assignée à s_i est grande).

Par ailleurs, une tâche est divisée en instances et chaque instance survient durant

une seule période de la tâche. Chaque instance de tâche peut être *rouge* ou *bleue*. Une instance de tâche *rouge* doit obligatoirement s'exécuter dans le respect de son échéance. Une instance de tâche *bleue* peut être abandonnée à tout moment. Cependant, si une instance bleue réussit à s'éxécuter dans le respect de son échéance, l'instance suivante de la tâche est également bleue.

Dans la suite, nous présentons et illustrons les deux algorithmes basés sur *EDF* établis par Koren and Shasha dans [KS95], reposant sur le modèle de tâches à pertes contraintes Skip-Over.

1.1 L'algorithme RTO

Le premier algorithme proposé est l'algorithme *Red Tasks Only* (*RTO*) [KS95]. Les instances de tâches rouges sont ordonnancées au plus tôt selon l'algorithme *EDF*, tandis que les instances bleues sont systématiquement rejetées. Les conflits d'échéances sont résolus en faveur de la tâche présentant la date de réveil la plus ancienne vis-à-vis du temps courant.

Dans le modèle "purement rouge" (*deeply red*) où les tâches sont activées de manière synchrone et où les $s_i - 1$ premières instances de chaque tâche sont rouges, cet algorithme est optimal [KS95].

1.1.1 Description algorithmique

L'algorithme d'ordonnancement RTO est chargé de gérer 2 listes de tâches : une liste de tâches en attente et une liste de tâches prêtes. Dans ce qui suit, la dénomination *tâche* se réfère en réalité à une *instance de tâche*.

Lorsqu'une tâche de la liste des tâches en attente, possède une date de réveil inférieure ou égale au temps courant, alors celle-ci est évaluée pour passer à l'état prêt. Si son paramètre de pertes dynamiques s_{i_dyn} correspondant au type de l'instance occurrente de la tâche, est strictement inférieur à son paramètre de pertes maximales autorisées s_i alors, la tâche occurrente est une tâche rouge et celle-ci est placée dans la liste des tâches rouges prêtes. Sinon, la tâche est abandonnée et ses paramètres sont mis à jour pour sa prochaine occurrence. Notons qu'à l'initialisation, $s_{i\_dyn} = 1$ pour toutes les tâches.

La description algorithmique de RTO est fournie ci-dessous dans l'algorithme 1.

Algorithme 1
RTO_schedule*(t : temps courant)*
début
  */*Recherche des réveils de tâches associés au temps courant*/*
  tantque *(liste des tâches en attente=***not***(∅))* **faire**
    si *(tâche→ $r_i > t$)*
      break
    finsi
    si *(tâche→ s_{i_dyn} <tâche→ s_i)*
      Retirer la tâche de la liste des tâches en attente

Insérer la tâche dans la liste des tâches rouges prêtes
sinon
$tâche\rightarrow r_i = tâche\rightarrow r_i + task\rightarrow P_i$
$tâche\rightarrow s_{i\_dyn} = 0$
finsi
$tâche\rightarrow s_{i\_dyn} = tâche\rightarrow s_{i\_dyn} + 1$
fintantque
fin

La décision d'ordonnancement est en $O(n^2)$ dans le pire-cas, qui correspond au cas où les n tâches doivent être réveillées simultanément.

1.1.2 Illustration

RTO est illustré sur la figure III.1 sur un ensemble $\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\}$ constitué de 5 tâches dont les paramètres sont décrits dans la table III.1. Les tâches possèdent un paramètre de pertes uniforme $s_i = 2$ et le facteur d'utilisation du processeur sans pertes $U_p = \sum_{i=1}^{n} \frac{C_i}{P_i}$ est égal à 1.15. Le facteur d'utilisation équivalent du processeur intégrant les pertes (cf. Chap II. Définition 9), est quant à lui égal à $U_p^* = 0.75$. Par conséquent, d'après le Théorème 21 (cf. Chapitre II. Section 5.1), l'ensemble $\mathcal{T}$ est ordonnançable au sens RTO (les exécutions des instances obligatoires, à savoir les instances rouges, sont garanties).

Task	T_1	T_2	T_3	T_4	T_5
c_i	3	4	1	7	2
p_i	30	20	15	12	10

TAB. III.1 – Un ensemble basique de tâches avec pertes

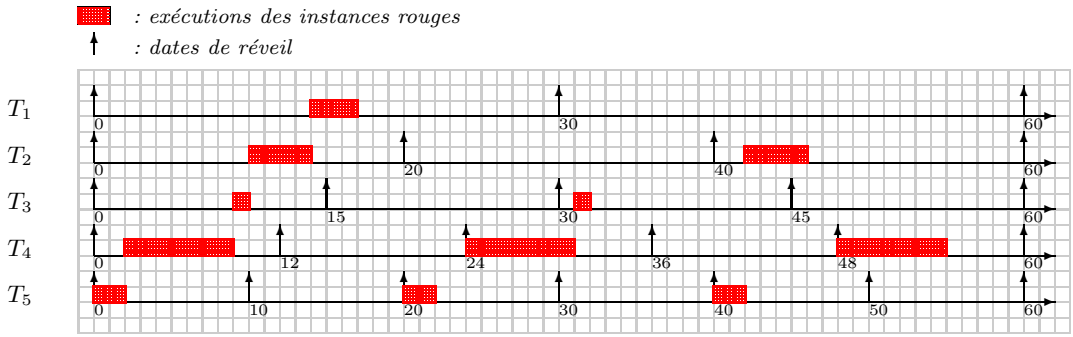

FIG. III.1 – Illustration de l'algorithme d'ordonnancement RTO

Comme nous pouvons l'observer, la distance entre deux pertes est exactement égale à s_i périodes. Par conséquent, RTO garantit seulement un niveau de QoS minimal pour les tâches périodiques.

1.2 L'algorithme BWP

Le second algorithme étudié dans [KS95] est l'algorithme *Blue When Possible* (*BWP*) qui constitue une amélioration de l'algorithme précédent. En effet, BWP ordonnance les instances bleues dès qu'il n'y a plus d'instances rouges à l'état prêt. En ce sens, il opère d'une manière plus flexible. Les conflits d'échéances sont toujours résolus en faveur de la tâche dont la date de réveil est la plus ancienne. Les tâches bleues sont exécutées entre elles selon EDF.

1.2.1 Description algorithmique

L'algorithme d'ordonnancement BWP est chargé de gérer 3 listes de tâches : une liste de tâches en attente et 2 listes de tâches prêtes correspondant aux 2 types de tâches occurrentes (rouges et bleues).

L'algorithme BWP opère en deux phases distinctes. Dans un premier temps, celui-ci examine la liste des tâches bleues prêtes dans le but d'abandonner, si nécessaire, les tâches bleues dont les échéances absolues sont supérieures ou égales au temps courant. Dans un deuxième temps, la liste des tâche en attente est parcourue de manière à réveiller les tâches ayant une date de réveil inférieure ou égale au temps courant. Dans ce cas, si le paramètre de pertes dynamiques s_{i_dyn} de la tâche est strictement inférieur à son paramètre de pertes maximales autorisées s_i alors, la tâche occurrente est une tâche rouge et celle-ci est placée dans la liste des tâches rouges prêtes. Sinon, la tâche occurrente est une tâche bleue et celle-ci est placée dans la liste des tâches bleues prêtes.

La description algorithmique de BWP est fournie ci-dessous dans l'algorithme 2.

Algorithme 2
BWP_schedule*(t : temps courant)*
begin
*/*Recherche des abandons de tâches bleues associés au temps courant*/*
tantque *(liste des tâches bleues prêtes=***not***(∅))* **faire**
si *(tâche→ r_i+tâche→ D_i < t)*
break
finsi
Retirer la tâche de la liste des tâches bleues prêtes
tâche→ r_i =tâche→ r_i+task→ P_i
tâche→ $s_{i\_dyn} = 0$
Insérer la tâche dans la liste des tâches en attente
fintantque
*/*Recherche des réveils de tâches associés au temps courant*/*
tantque *(liste des tâches en attente=***not***(∅))* **faire**

si *(task→ $r_i > t$)*
break
finsi
Retirer la tâche de la liste des tâches en attente
si *(tâche→ s_{i_dyn} <tâche→ s_i)*
Insérer la tâche dans la liste des tâches rouges prêtes
sinon
Insérer la tâche dans la liste des tâches bleues prêtes
finsi
tâche→ s_{i_dyn} =tâche→ $s_{i\_dyn} + 1$
fintantque
fin

1.2.2 Illustration

La figure III.2 montre un exemple illustratif de l'ordonnancement BWP basé sur l'ensemble de tâches décrit précédemment dans la table III.1.

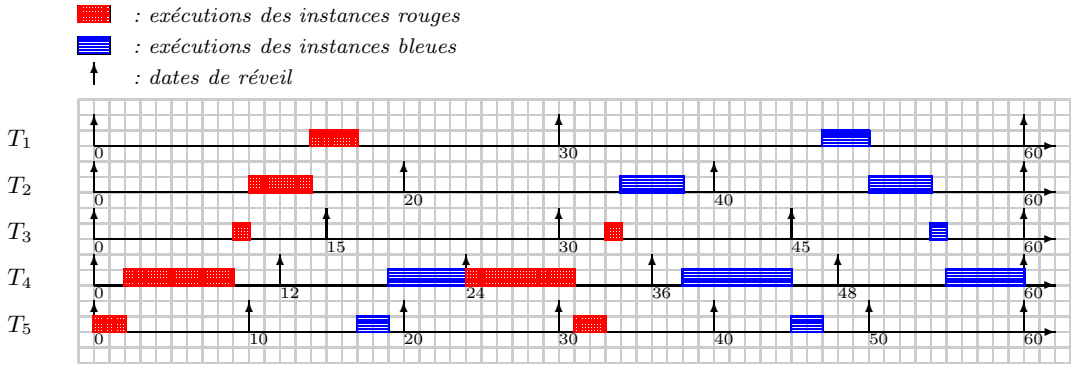

FIG. III.2 – Illustration de l'algorithme d'ordonnancement BWP

Comparé à l'algorithme RTO, un nombre plus important d'instances de tâches s'exécutent dans le respect de leurs échéances avec l'algorithme BWP. Nous observons cependant 5 violations d'échéances relatives à des instances de tâches bleues. Celles-ci surviennent aux instants $t = 24$ (tâche T_4), $t = 30$ (tâches T_3 et T_5) et $t = 60$ (tâches T_4 et T_5). Notons bien que la qualité de service, en termes de taux de succès global des instances périodiques, se trouve d'autant plus réduite que le nombre de violations d'échéances est important.

Dans la suite, nous proposons une amélioration de la qualité de service globale du système basée sur l'utilisation d'un ordonnancement EDL. C'est pourquoi, avant d'introduire les deux nouveaux algorithmes d'ordonnancement Skip-Over établis dans le cadre

des travaux de thèse, nous nous intéressons à l'adaptation de l'algorithme EDL à des tâches RTO (instances rouges uniquement) et BWP (instances rouges et bleues).

2 L'ordonnancement Skip-Over basé sur EDL

2.1 Application de l'algorithme EDL au modèle RTO

Dans cette partie nous exposons l'adaptation des équations de calcul des temps creux sous EDL d'un modèle de tâches périodiques sans pertes à un modèle de tâches RTO. Dans un premier temps, une adaptation basique est présentée. Ensuite, une version optimisée d'un point de vue de la complexité algorithmique du calcul est exposée.

2.1.1 Calcul des temps creux sur $[\tau, kP'[$

Soit $P' = ppcm(s_1P_1, s_2P_2, ..., s_nP_n)$ l'hyperpériode définie sur un ensemble de tâches avec pertes. τ représente le temps courant et kP' avec $k = (\lfloor \frac{\tau}{P'} \rfloor + 1)$ figure la date de fin de l'hyperpériode courante contenant τ. Le calcul des temps creux se réduit à cet intervalle car les instances rouges des tâches périodiques avec pertes sont synchrones toutes les P' unités de temps. Une séquence RTO se répète donc à l'identique toutes les P' unités de temps.

Nous présentons dans un premier temps le calcul des temps creux effectué hors-ligne sur l'ensemble de l'hyperpériode P'.

2.1.1.1 Calcul du vecteur statique des échéances sur $[0, P'[$. Le vecteur statique des échéances, noté $\mathcal{K}$, représente l'ensemble des instants considérés sur l'hyperpériode $[0, P'[$, précédant un temps creux. Il se construit à partir des échéances distinctes des instances périodiques *rouges*. La distance entre deux skips consécutifs est exactement de s_i périodes dans lesquelles on observe m instances rouges suivies d'une instance bleue avec $m{=}(s_i - 1)$, comme l'illustre la figure III.3. Ces m instances rouges, considérées individuellement, sont donc périodiques de période s_iP_i. Toutes les tâches ont leur échéance égale à leur période d'activation (c'est-à-dire que chaque instance doit se terminer avant la prochaine activation de la tâche). Ainsi, pour chaque tâche T_i, on observera sur $[0, P'[$, $\frac{mP'}{s_iP_i}$ échéances d'instances rouges, $1{\leq}\text{i}{\leq}n$. Par conséquent, les instants k_i sous RTO sont définis par le théorème suivant :

Théorème 22 *Les instants k_i du vecteur $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$ calculés sur $[0, P'[$ pour des tâches RTO sont définis comme suit :*

$$k_i = (x.s_j + k).P_j \tag{III.1}$$

avec $x = 0, ..., (\frac{P'}{s_jP_j} - 1)$, $k = 1, ..., m$, $k_i < k_{i+1}$, $k_0 = \min\{P_j; 1 \leq j \leq n\}$, et $k_q = P' - \min\{P_j; 1 \leq j \leq n\}$

Preuve :

Par définition, dans le modèle RTO, chaque tâche T_j de période P_j et de paramètre de pertes s_j, présente sur l'intervalle $[0, s_jP_j]$, $(s_j - 1)$ requêtes d'exécution d'instances rouges suivies d'*une* requête d'exécution d'instance bleue. Ainsi, les échéances sur requêtes rouges observées sur $[0, s_jP_j]$ sont situées aux instants kP_j avec $1 \leq k \leq (s_j - 1)$. Cette séquence de requêtes d'exécution de $(s_j - 1)$ instances rouges et d'*une* instance bleue, est périodique de période s_jP_j. Par conséquent, pour chaque tâche T_j, cette séquence RTO se répète $\frac{P'}{s_jP_j}$ fois sur $[0, P'[$. On en déduit alors que les échéances sur requêtes rouges sur $[0, P'[$ sont situées aux instants $xs_jP_j + kP_j$ soit encore $(xs_j + k)P_j$, avec $1 \leq k \leq (s_j - 1)$ et $0 \leq x \leq \frac{P'}{s_jP_j} - 1$. □

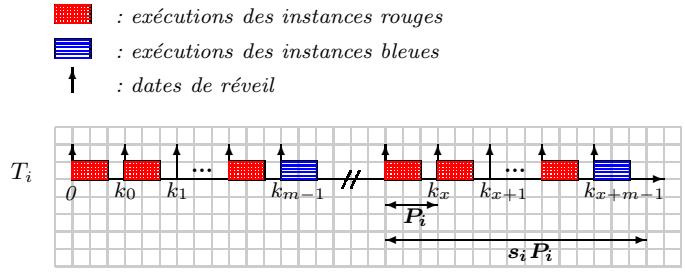

FIG. III.3 – Positionnement des instants $k_i(x, k)$ sous RTO

2.1.1.2 Calcul du vecteur statique des temps creux sur $[0, P'[$. Le vecteur statique des temps creux $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des échéances. Δ_i^* représente la longueur du temps creux débutant au temps k_i. Le théorème suivant fournit la formule de récurrence pour le calcul du vecteur $\mathcal{D}^*$ sous RTO, tenant compte des pertes s_j autorisées par chacune des tâches T_j :

Théorème 23 *Les durées des temps creux du vecteur* $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ *calculées sur* $[0, P'[$ *pour des tâches RTO sont définies comme suit :*

$$\Delta_q^* = \min\{P_i; 1 \leq i \leq n\} \tag{III.2}$$

$$\Delta_i^* = \sup(0, F_i), \quad pour\ i = q - 1\ à\ 0 \tag{III.3}$$

$$avec \quad F_i = (P' - k_i) - \sum_{j=1}^{n} (\lceil \frac{P' - k_i}{P_j} \rceil - \lceil \frac{P' - k_i}{s_jP_j} \rceil) c_j - \sum_{k=i+1}^{q} \Delta_k^*$$

Preuve :

Considérons pour chaque instant k_i, $0 \leq i \leq q$, la durée du temps creux qui lui est associée, notée Δ_i^*, pouvant être égale à zéro. Soit $P' = ppcm(s_1P_1, ..., s_iP_i, ..., s_nP_n)$. Pour chaque tâche T_i, la dernière instance observée sur $[0, P']$ est bleue. Par conséquent, puisque k_q correspond à la dernière échéance rouge survenant sur $[0, P'[$, Δ_q^* est donné par $P' - k_q$ aussi égal à $\min\{P_i; 1 \leq i \leq n\}$.

Dans l'intervalle $[k_i, P']$, avec $0 \leq i \leq q$, le nombre d'échéances sur requêtes, rouges et bleues confondues, pour chaque tâche T_j est égale à $\lceil \frac{P'-k_i}{P_j} \rceil$. Si toutes ces requêtes ont terminé leur exécution sur $[k_i, P']$, alors la durée totale du temps creux sur $[k_i, P']$ est $(P'-k_i) - \sum_{j=1}^{n} \lceil \frac{P'-k_i}{P_j} \rceil c_j$. Dans le modèle RTO, chaque tâche T_j tolère d'écarter exactement une requête d'exécution toutes les s_j requêtes de la tâche. Les pertes totales par tâche sur $[k_i, P']$ s'élèvent alors à $\lceil \frac{P'-k_i}{s_j P_j} \rceil c_j$. Ainsi, la durée totale du temps creux sur $[k_i, P']$ en tenant compte des pertes est $(P' - k_i) - \sum_{j=1}^{n} (\lceil \frac{P'-k_i}{P_j} \rceil - \lceil \frac{P'-k_i}{s_j P_j} \rceil) c_j$. Comme la durée totale du temps creux sur $[k_{i+1}, P']$ est donnée par $\sum_{k=i+1}^{q} \Delta_k^*$, il vient que $\Delta_i^* = (P' - k_i) - \sum_{j=1}^{n} (\lceil \frac{P'-k_i}{P_j} \rceil - \lceil \frac{P'-k_i}{s_j P_j} \rceil) c_j - \sum_{k=i+1}^{q} \Delta_k^*$. Sinon, $\Delta_i^* = 0$. □

Illustration du calcul hors-ligne de f^{EDL} pour des tâches RTO. Considérons l'ensemble de tâches $\mathcal{T} = \{T_1, T_2\}$ constitué de deux tâches périodiques RTO, $T_1(3, 10, 2)$ et $T_2(3, 6, 2)$ avec $s_1 = s_2 = 2$.

En appliquant les formules III.1, III.2 et III.3, on obtient le vecteur statique des échéances $\mathcal{K} = (0, 6, 10, 18, 30, 42, 50, 54)$ et le vecteur statique des temps creux $\mathcal{D}^* = (3, 1, 5, 6, 9, 5, 1, 6)$. Le calcul des intervalles de temps creux fournit la séquence EDL représentée sur la figure III.4.

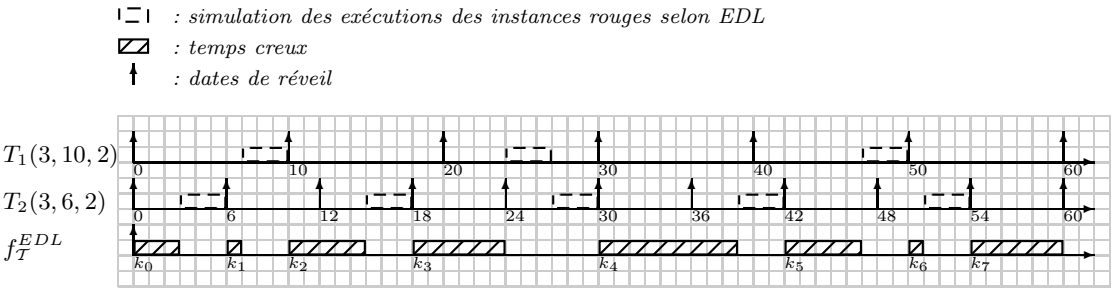

FIG. III.4 – Calcul des temps creux statiques sous RTO

L'établissement de la séquence EDL statique pour un modèle de tâche RTO est obtenue en $O(N'n)$ opérations dans le pire-cas, où N' représente le nombre de requêtes totales pour l'ensemble des tâches périodiques sur l'intervalle $[0, P'[$. Cette complexité dépend non seulement des périodes des tâches périodiques mais aussi des paramètres de pertes associés aux tâches. Elle peut donc s'avérer élevée tout particulièrement lorsque les périodes sont premières entre elles, ou bien lorsque les pertes autorisées au niveau des tâches sont relativement faibles (s_i grand). Cependant, notons que les vecteurs $\mathcal{K}$ et $\mathcal{D}^*$ sont construits une seule fois et que ce calcul s'effectue hors-ligne.

Dans la suite, nous nous intéressons au calcul des temps creux effectué en-ligne à un instant τ donné.

2.1.1.3 Calcul du vecteur dynamique des échéances sur $[\tau, kP'[$. Le vecteur dynamique des échéances $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$ représente les instants supérieurs ou égaux à τ (le temps courant) précédant un temps creux. Soit h l'index tel que $k_h = \sup\{d; d \in \mathcal{K} \text{ et } d < \tau\}$. Comme dans le cas statique, tous les instants k_i correspondent aux échéances distinctes des tâches *rouges*.

2.1.1.4 Calcul du vecteur dynamique des temps creux sur $[\tau, kP'[$. Le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ traduit les durées des temps creux associés aux instants consignés dans le vecteur $\mathcal{K}(\tau)$. $\Delta_i^*(\tau)$ représente la longueur du temps creux débutant au temps k_i, postérieur à τ. Par ailleurs, chaque tâche périodique T_i est caractérisée par la quantité de processeur $A_i(\tau)$ déjà allouée à son instance courante au temps τ. Cette instance possède également une échéance dynamique notée d_i. On note M, la plus grande échéance parmi toutes les instances de tâches périodiques actives et l'on figure dans le vecteur des temps creux l'indice f tel que $k_f = \min\{k_i; k_i > M\}$. $\mathcal{D}^*(\tau)$ est alors complètement défini par la relation de récurrence suivante décrite dans le théorème qui suit :

Théorème 24 *Les durées des temps creux du vecteur* $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ *calculés sur* $[\tau, kP']$ *pour des tâches RTO sont définies comme suit :*

$$\Delta_i^*(\tau) = \Delta_i^*, \quad \textit{pour } i = q \textit{ à } f \tag{III.4}$$

$$\Delta_i^*(\tau) = \sup(0, F_i(\tau)), \quad \textit{pour } i = f-1 \textit{ à } h+1 \tag{III.5}$$

$$\textit{avec} \quad F_i(\tau) = (P' - k_i) - \sum_{j=1}^{n} (\lceil \frac{P' - k_i}{P_j} \rceil - \lceil \frac{P' - k_i}{s_j P_j} \rceil) c_j + \sum_{\substack{j=1 \\ d_j > k_i}}^{n} A_j(\tau) - \sum_{k=i+1}^{q} \Delta_k^*(\tau)$$

$$\Delta_h^*(\tau) = (P' - \tau) - \sum_{j=1}^{n} (\lceil \frac{P' - \tau}{P_j} \rceil - \lceil \frac{P' - \tau}{s_j P_j} \rceil) c_j - A_j(\tau)) - \sum_{k=h+1}^{q} \Delta_k^*(\tau) \tag{III.6}$$

Preuve :

Considérons l'évaluation de la longueur du temps creux $\Delta_i^*(\tau)$ suivant l'instant k_i, avec $\tau < k_i \leq k_q$, dans la séquence EDL établie en-ligne. Soit $A_j(\tau)$ et d_j respectivement la quantité de traitement effectuée sur la requête courante de T_j et l'échéance de cette requête. On définit alors $M = \sup\{d_j; T_j \in \mathcal{T}(\tau)\}$ comme la plus grande échéance parmi l'ensemble $\mathcal{T}(\tau)$ de toutes les instances de tâches actives à l'instant τ. Etant donné que les tâches périodiques sont ordonnancées au plus tôt selon l'algorithme EDF (Earliest Deadline First) jusqu'à l'instant τ, aucune requête apériodique possédant une échéance plus grande que M n'a entamé son exécution à l'instant τ. Par conséquent, la séquence EDL produite pour $\mathcal{T}(\tau)$ sur $[k_f, P']$, avec $k_f = \min\{k_i; k_i > M\}$, est identique à celle produite pour $\mathcal{T}$ hors-ligne. D'où, $\Delta_i^*(\tau) = \Delta_i^*$, pour $i = q$ à f.

Si $d_j < k_i$, le temps total d'exécution requis par la tâche T_j sur $[k_i, P']$, en tenant compte des pertes, est donné par $(\lceil \frac{P'-k_i}{P_j} \rceil - \lceil \frac{P'-k_i}{s_j P_j} \rceil) c_j$. Sinon, il est donné par $(\lceil \frac{P'-k_i}{P_j} \rceil - \lceil \frac{P'-k_i}{s_j P_j} \rceil) c_j - A_j(\tau)$ puisque la requête courante de T_i a été partiellement exécutée. Par conséquent, la durée totale des temps creux sur $[k_i, P']$ est donnée par $(P' - k_i) - \sum_{j=1}^{n} (\lceil \frac{P'-k_i}{P_j} \rceil - \lceil \frac{P'-k_i}{s_j P_j} \rceil) c_j + \sum_{\substack{j=1 \\ d_j > k_i}}^{n} A_j(\tau)$. □

Illustration du calcul en-ligne de f^{EDL} pour des tâches RTO. Nous pouvons à présent appliquer ces résultats à l'ensemble $\mathcal{T} = \{T_1(3, 10, 2), T_2(3, 6, 2)\}$. Cet ensemble de tâches périodiques RTO est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 5$. Supposons que l'on souhaite calculer le maximum de temps processeur libre dans l'intervalle $[5, 30[$ relativement à l'ensemble $\mathcal{T}$. A partir des formules III.4 et III.6, on extrait le vecteur dynamique des échéances $\mathcal{K}(\tau) = (5, 6, 10, 18, 30, 42, 50, 54)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (1, 3, 5, 6, 9, 5, 1, 6)$, comme l'illustre la figure III.5.

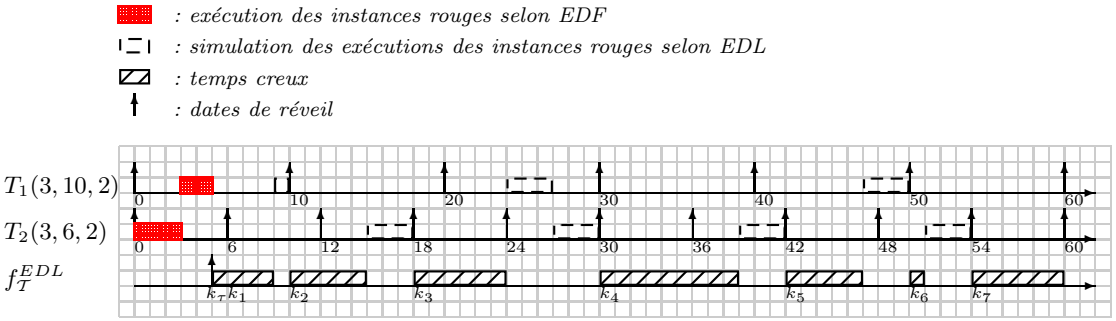

FIG. III.5 – Calcul des temps creux dynamiques à l'instant $\tau = 5$ sous RTO

En ce qui concerne la complexité algorithmique du calcul du vecteur dynamique des temps creux, on observe que la détermination des temps creux avec un modèle de tâches RTO est la même que celle observée avec un modèle de tâches classiques, soit en $O(\lceil \frac{R}{p} \rceil n)$ où n désigne le nombre de tâches périodiques, R la plus grande échéance parmi les tâches prêtes et partiellement exécutées, et p la plus petite période. En effet, étant donné que le vecteur statique des temps creux est mémorisé, la reconstruction de la séquence n'est nécessaire qu'entre τ et la plus grande échéance de tâche rouge parmi les tâches prêtes à τ, d'où le facteur $\lfloor \frac{R}{p} \rfloor n$.

Quant à la complexité spatiale, celle-ci est déterminée par la taille du vecteur statique des temps creux. Dans le pire-cas, ce dernier est en $O(N')$.

2.1.2 Calcul optimisé des temps creux sur $[\tau, kP[$

Dans ce qui a été exposé précédemment, la séquence EDL est calculée sur l'intervalle $[\tau, k'P'[$ où $P' = ppcm(s_1P_1, s_2P_2, ..., s_nP_n)$ représente l'hyperpériode correspondant à des tâches périodiques avec pertes. Comme nous avons pu le souligner précédemment, la durée du calcul des temps creux en statique ainsi que la taille des structures de données sur lesquelles repose le calcul EDL, seront donc d'autant plus importantes que les pertes autorisées au niveau des tâches seront faibles (paramètre s_i grand). Il serait donc intéressant d'établir une séquence EDL pour des tâches périodiques avec pertes, qui soit indépendante des pertes appliquées aux tâches.

L'idée de l'optimisation envisagée ici est donc de ramener le calcul de EDL pour des tâches avec pertes sur l'intervalle $[\tau, kP[$ uniquement, où $P = ppcm(P_1, P_2, ..., P_n)$ représente une sous-hyperpériode des tâches périodiques avec pertes égale à l'hyperpériode des tâches considérées sans pertes, et où kP (avec $k = \lfloor \frac{\tau}{P} \rfloor + 1$) figure la date de fin de la sous-hyperpériode courante contenant τ. Cette optimisation vise bien sûr à réduire la complexité du calcul des temps creux. Les tâches périodiques avec pertes sont synchrones toutes les P unités de temps même si les types d'instances observés pour les tâches à ces instants peuvent différer. Si l'on souhaite appliquer EDL jusqu'à la fin de la sous-hyperpériode courante P uniquement, il est nécessaire de considérer, à l'image du calcul jusqu'à la fin de l'hyperpériode P', qu'à la fin de la sous-hyperpériode courante P, toutes les instances réveillées seront rouges. Cette hypothèse est équivalente au fait que les dernières instances de tâches de la sous-hyperpériode courante soient bleues.

La solution envisagée consiste donc à déterminer, pour chaque tâche T_i, le décalage $pos_i(\tau)$ (avance dans le temps à $t = \tau$, exprimée en nombre de périodes P_i de la tâche) de la séquence des instances de la sous-hyperpériode courante vis-à-vis d'une séquence d'instances considérée sur la même sous-hyperpériode mais dans laquelle la dernière instance serait bleue.

Illustrons notre propos sur un exemple. On considère l'ensemble $\mathcal{T} = \{T_0, T_1, T_2, T_3, T_4\}$ constitué de 5 tâches périodiques dont les paramètres sont décrits dans la Table III.2. Les tâches présentent un paramètre de pertes uniforme $s_i = 3$.

Task	T_0	T_1	T_2	T_3	T_4
c_i	2	2	2	2	2
P_i	30	20	15	12	10

TAB. III.2 – Un ensemble basique de tâches avec pertes

L'ordonnancement de $\mathcal{T}$ par EDL sur la sous-hyperpériode $[0, 60[$ est représenté sur la figure III.6.

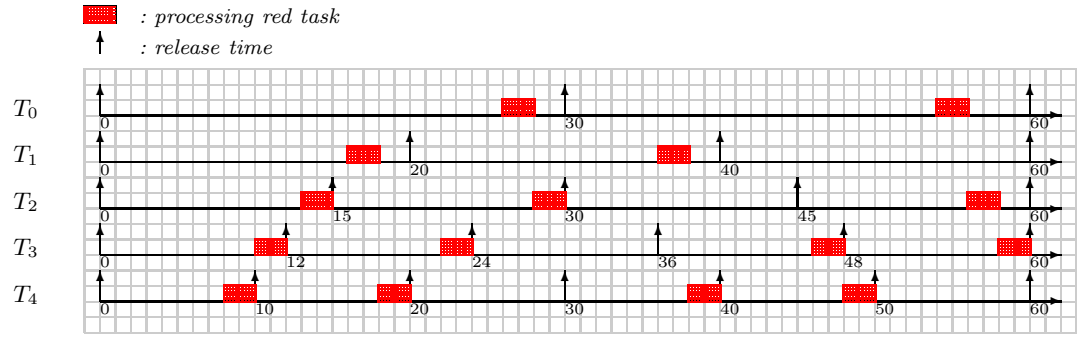

FIG. III.6 – Ordonnancement de $\mathcal{T}$ avec EDL sur la première sous-hyperpériode P

Etablissons à présent l'ordonnancement de $\mathcal{T}$ par EDL dans laquelle la dernière instance dans cette sous-hyperpériode serait bleue (cf. figure III.7).

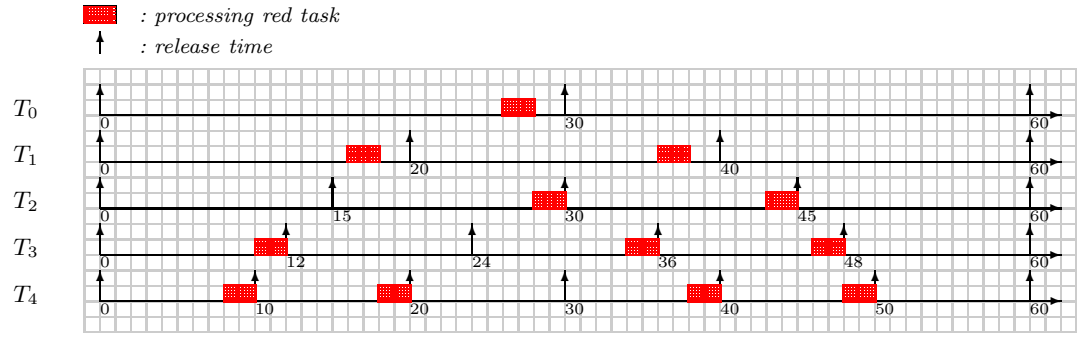

FIG. III.7 – Ordonnancement de $\mathcal{T}$ avec EDL avec une instance bleue à la fin

Au niveau des tâches T_1 et T_4, nous constatons qu'il n'y a aucun décalage et que la dernière instance de la sous-hyperpériode est bleue dans les deux cas. Ceci s'explique par le fait que la quantité $\lceil \frac{P}{s_i P_i} \rceil$ qui représente le nombre de séquences unitaires complètes (séquences d'instances constituées de $(s_i - 1)$ instances rouges et d'une instance bleue) dans P, est multiple de s_i. Pour T_0 et T_3, nous observons que la séquence des instances de tâches est en avance d'une période dans le temps vis-à-vis d'une séquence terminée par une instance bleue. En effet, nous pouvons remarquer que, pour la tâche T_3, la séquence d'instances observée sur l'intervalle $[24, 60[$ sur la figure III.6 est équivalente à celle observée sur l'intervalle $[12, 48[$ sur la figure III.7. Pour T_2 enfin, le décalage observé est de 2 périodes. Cela signifie qu'il manque 2 instances de tâches sur [0,P[pour former $\lceil \frac{P}{s_i P_i} \rceil$ séquences unitaires complètes.

Plus formellement, le décalage initial des instances peut être exprimé par un paramètre de positionnement calculé de la manière suivante :

Théorème 25 *Le paramètre de positionnement initial pos_i correspondant à l'avance de chaque tâche T_i au temps t=0, vis-à-vis d'une séquence dans laquelle la dernière instance de la première sous-hyperpériode P est bleue, est donnée par :*

$$pos_i = \lceil \frac{P}{s_i P_i} \rceil s_i - \frac{P}{P_i}. \tag{III.7}$$

Preuve :

L'avance d'une tâche T_i à l'instant $t = 0$ correspond au nombre d'instances manquantes sur la sous-hyperpériode P pour que toutes les séquences unitaires soient complètes. A partir de ce constat, il convient donc de quantifier ce nombre sur l'intervalle $[0, P[$. Les instances d'une tâche T_i considérées sur l'intervalle $[0, P[$, peuvent être regroupées en séquences unitaires complètes comportant chacune, par définition, s_i instances de tâches. Si la tâche présente un décalage, l'une des séquences unitaires sera incomplète souffrant d'un défaut de pos_i instances de tâches.

Dans l'intervalle $[0, P[$, le nombre de séquences unitaires complètes pour chaque tâche T_i est donné par $\lceil \frac{P}{s_i P_i} \rceil$, ce qui correspond à $\lceil \frac{P}{s_i P_i} \rceil s_i$ instances de tâches. Par ailleurs, le nombre total d'instances d'une tâche T_i sur l'intervalle $[0, P[$ est égal à $\frac{P}{P_i}$. Ainsi, la quantité $\lceil \frac{P}{s_i P_i} \rceil s_i - \frac{P}{P_i}$ figure le nombre d'instances de tâches manquantes sur $[0, P[$ pour que toutes les séquences unitaires soient complètes. Elle correspond de ce fait à l'avance de la tâche T_i sur la première sous-hyperpériode. □

Ce résultat peut être étendu à n'importe quelle sous-hyperpériode P de l'hyperpériode P'. Soit $pos_i(\tau)$ le décalage des instances à l'instant τ. Celui-ci est complètement défini par la formule énoncée dans le théorème 26.

Théorème 26 *Le paramètre de positionnement $pos_i(\tau)$ correspondant à l'avance de chaque tâche T_i à un temps $t = \tau$, vis-à-vis d'une séquence dans laquelle la dernière instance de la sous-hyperpériode courante contenant τ est bleue, est donné par :*

$$pos_i(\tau) = \lceil \frac{kP}{s_i P_i} \rceil s_i - \frac{kP}{P_i} \tag{III.8}$$

avec $k = (\lfloor \frac{\tau}{P} \rfloor + 1)$.

Preuve :

L'avance d'une tâche T_i à l'instant $t = \tau$ correspond au nombre d'instances manquantes sur la sous-hyperpériode courante pour que toutes les séquences unitaires soient complètes. A partir de ce constat, il convient donc de quantifier ce nombre sur l'intervalle $[0, kP[$, où $k = \lfloor \frac{\tau}{P} \rfloor + 1$. Les instances d'une tâche T_i considérées sur l'intervalle $[0, kP[$, peuvent être regroupées en séquences unitaires complètes comportant chacune, par définition, s_i instances de tâches. Si la tâche présente un décalage, l'une des séquences unitaires sera incomplète souffrant d'un défaut de $pos_i(\tau)$ instances de tâches.

Dans l'intervalle $[0, kP[$, le nombre de séquences unitaires complètes pour chaque tâche T_i est donné par $\lceil \frac{kP}{s_i P_i} \rceil$, ce qui correspond à $\lceil \frac{kP}{s_i P_i} \rceil s_i$ instances de tâches. Par ailleurs, le nombre total d'instances d'une tâche T_i sur l'intervalle $[0, kP[$ est égal à $\frac{kP}{P_i}$. Ainsi, la quantité $\lceil \frac{kP}{s_i P_i} \rceil s_i - \frac{kP}{P_i}$ figure le nombre d'instances de tâches manquantes sur $[0, kP[$ pour que toutes les séquences unitaires soient complètes. Elle correspond de ce fait à l'avance de la tâche T_i sur la sous-hyperpériode courante. □

Corollaire 1 *Le paramètre de positionnement $pos_i(\tau)$ correspondant à l'avance de chaque tâche T_i à un temps $t = \tau$, vis-à-vis d'une séquence dans laquelle la dernière instance de la sous-hyperpériode courante contenant τ est bleue, peut s'exprimer sous la forme suivante :*

$$pos_i(\tau) = (k.pos_i)\%s_i. \tag{III.9}$$

avec $k = (\lfloor \frac{\tau}{P} \rfloor + 1)$.

Preuve :

Considérons la division euclidienne qui, à deux entiers a et b, avec b strictement positif, associe un unique quotient q et un unique reste r, tous deux entiers, vérifiant l'équation $a = bq + r$. Soit $a = \frac{P}{Pi}$

et $b = s_i$, tous deux entiers. L'équation peut alors s'écrire sous la forme $\frac{P}{P_i} = s_i\lfloor\frac{P}{s_iP_i}\rfloor + (\frac{P}{P_i})\%s_i$. Par conséquent, $\lfloor\frac{P}{s_iP_i}\rfloor s_i = \frac{P}{P_i} - (\frac{P}{P_i})\%s_i$.

D'après le théorème 25, $pos_i = \lceil\frac{P}{s_iP_i}\rceil s_i - \frac{P}{P_i}$. Dans cette expression, si $\frac{P}{s_iP_i}$ est un entier alors $\frac{P}{P_i}\%s_i = 0$ et $pos_i = 0$. Si $\frac{P}{s_iP_i}$ n'est pas un entier, l'égalité suivante est vérifiée : $\lceil\frac{P}{s_iP_i}\rceil = \lfloor\frac{P}{s_iP_i}\rfloor + 1$ et donc $pos_i = (\lfloor\frac{P}{s_iP_i}\rfloor + 1)s_i - \frac{P}{P_i} = \lfloor\frac{P}{s_iP_i}\rfloor s_i - \frac{P}{P_i} + s_i$ soit encore $pos_i = -(\frac{P}{P_i})\%s_i + s_i$. Par ailleurs, $\frac{P}{P_i}\%s_i = \frac{P}{P_i} - ps_i$ avec p entier. Par conséquent, $pos_i = -\frac{P}{P_i} + ps_i + si$, soit $pos_i = -\frac{P}{P_i} + (p+1)s_i$.

Par un raisonnement similaire, nous obtenons que $pos_i(\tau) = -\frac{kP}{P_i} + (p'+1)s_i$ avec p' entier. En remplaçant $-\frac{P}{P_i}$ par $pos_i - (p+1)s_i$ dans cette dernière équation, nous obtenons la relation suivante : $pos_i(\tau) = k.pos_i + p''s_i$, avec p'' entier, à savoir $pos_i(\tau) = (k.pos_i)\%s_i$. □

Observons l'évolution du décalage des instances sur un exemple. Considérons individuellement la tâche T_0 de l'ensemble précédent. La succession temporelle des instances sur une hyperpériode est figurée sur le chronogramme suivant :

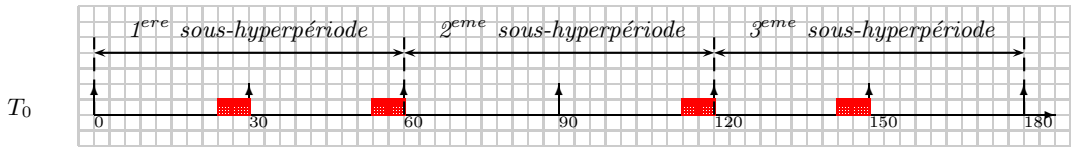

FIG. III.8 – Illustration de la succession des instances de T_0 sur une hyperpériode

Le décalage initial de T_0 calculé à partir du théorème 25, nous donne :

$$pos_0 = \lceil\frac{60}{3*30}\rceil * 3 - \frac{60}{30} = \lceil\frac{60}{90}\rceil * 3 - 2 = 3 - 2 = 1$$

Ce résultat est en accord avec ce que nous observons, à savoir que la séquence d'instances de T_0 sur la 1ère sous-hyperpériode est en avance d'une période $P_0 = 30$ vis-à-vis d'une séquence où la dernière instance est bleue. Calculons à présent les décalages dans les sous-hyperpériodes suivantes en considérant les instants $\tau = 100$ et $\tau = 150$:

$$pos_0(100) = ((\lfloor\frac{100}{60}\rfloor + 1) * 1)\%3 = ((1+1)*1)\%3 = 2\%3 = 2$$
$$pos_0(150) = ((\lfloor\frac{150}{60}\rfloor + 1) * 1)\%3 = ((2+1)*1)\%3 = 3\%3 = 0$$

La séquence d'instances est effectivement décalée de 2 périodes sur la 2ème hyperpériode tandis que nous vérifions le fait que, par définition, les instances sur la dernière sous-hyperpériode ne présentent pas de décalage.

Dans ce qui suit, nous présentons les nouvelles équations pour le calcul des temps creux par EDL pour des tâches RTO, sur une sous-hyperpériode P cette fois-ci. Celles-ci font bien sûr intervenir le décalage initial (pos_i) et courant ($pos_i(\tau)$) que nous avons établis précédemment.

2.1.2.1 Calcul du vecteur statique des échéances sur $[0, P[$. Le vecteur statique des échéances, noté $\mathcal{K}$, représente l'ensemble des instants considérés sur l'hyperpériode $[0, P[$, précédant un temps creux. Il se construit à partir des échéances distinctes des instances périodiques *rouges*. Pour chaque tâche T_i, on observera sur $[0, P[$, $\frac{mP}{s_i P_i}$ échéances d'instances rouges, $1 \leq i \leq n$. Par conséquent, les instants k_i sous RTO sont définis par le théorème suivant :

Théorème 27 *Les instants k_i du vecteur $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$ calculés sur $[0, P[$ pour des tâches RTO sont définis comme suit :*

$$k_i = (x.s_j + k).P_j \tag{III.10}$$

avec $x = 0, ..., (\frac{P}{s_j P_j} - 1)$, $k = 1, ..., m$, $k_i < k_{i+1}$, $k_0 = \min\{P_j; 1 \leq j \leq n\}$, *et* $k_q = P - \min\{P_j; 1 \leq j \leq n\}$

Preuve :

Par définition, dans le modèle RTO, chaque tâche T_j de période p_j et de paramètre de pertes s_j, présente sur l'intervalle $[0, s_j P_j]$, $(s_j - 1)$ requêtes d'exécution d'instances rouges suivies d'*une* requête d'exécution d'instance bleue. Ainsi, les échéances sur requêtes rouges observées sur $[0, s_j P_j]$ sont situées aux instants kP_j avec $1 \leq k \leq (s_j - 1)$. Cette séquence de requêtes d'exécution de $(s_j - 1)$ instances rouges et d'*une* instance bleue, est périodique de période $s_j P_j$. Par conséquent, pour chaque tâche T_j, cette séquence RTO se répète $\frac{P}{s_j P_j}$ fois sur $[0, P[$. On en déduit alors que les échéances sur requêtes rouges sur $[0, P[$ sont situées aux instants $xs_j P_j + kP_j$ soit encore $(xs_j + k)P_j$, avec $1 \leq k \leq (s_j - 1)$ et $0 \leq x \leq \frac{P}{s_j P_j} - 1$. □

2.1.2.2 Calcul du vecteur statique des temps creux sur $[0, P[$. Le vecteur statique des temps creux $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des échéances. Δ_i^* représente la longueur du temps creux débutant au temps k_i. Le théorème suivant fournit la formule de récurrence pour le calcul du vecteur $\mathcal{D}^*$ sous RTO, tenant compte des pertes s_j autorisées par chacune des tâches T_j :

Théorème 28 *Les durées des temps creux du vecteur $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$ calculés sur $[0, P[$ pour des tâches RTO sont définies comme suit :*

$$\Delta_i^* = \sup(0, F_i), \quad \textit{pour } i = q \textit{ à } 0 \tag{III.11}$$

$$\textit{avec} \quad F_i = (P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P - k_i}{P_j} \rceil - \lceil \frac{P - k_i + pos_j P_j}{s_j P_j} \rceil + \lceil \frac{pos_j P_j}{s_j P_j} \rceil) c_j - \sum_{k=i+1}^{q} \Delta_k^*$$

Preuve :

La séquence RTO observée sur $[k_i, P[$ est identique à une séquence RTO observée sur $[k_i\text{-}pos_j P_j, P - pos_j P_j]$ dans laquelle la dernière instance de la sous-hyperpériode P est bleue et où l'on tient compte du décalage pos_j, $0 \leq pos_j \leq (s_j - 1)$, associée à la tâche T_j. L'évaluation des pertes sur cet intervalle correspond au nombre d'instances bleues présentes dans la séquence RTO considérée

sur $[k_i - pos_j P_j, P - pos_j P_j]$. Cette quantité est égale au nombre de pertes observées sur l'intervalle $P-(k_i-pos_j P_j)$ auquel on soustrait le nombre de pertes observées sur l'intervalle $P-(P-pos_j P_j)$. Les pertes totales par tâche sur $[k_i, P[$ s'élèvent alors à $\lceil \frac{P-(k_i-pos_j P_j)}{s_j P_j} \rceil c_j - \lceil \frac{P-(P-pos_j P_j)}{s_j P_j} \rceil c_j$, soit après simplifications $(\lceil \frac{P-k_i+pos_j(\tau)P_j}{s_j P_j} \rceil - \lceil \frac{pos_j P_j}{s_j P_j} \rceil)c_j$.

Ainsi, la durée totale du temps creux sur $[k_i, P[$ en tenant compte des pertes est $(P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j P_j}{s_j P_j} \rceil + \lceil \frac{pos_j P_j}{s_j P_j} \rceil)c_j$. Comme la durée totale du temps creux sur $[k_{i+1}, P[$ est donnée par $\sum_{k=i+1}^{q} \Delta_k^*$, il vient que $\Delta_i^* = (P-k_i) - \sum_{j=1}^{n} (\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j P_j}{s_j P_j} \rceil + \lceil \frac{pos_j P_j}{s_j P_j} \rceil)c_j - \sum_{k=i+1}^{q} \Delta_k^*$. □

2.1.2.3 Calcul du vecteur dynamique des échéances sur $[\tau, kP[$.

Le vecteur dynamique des échéances $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$ représente les instants supérieurs ou égaux à τ (le temps courant) précédant un temps creux. Soit h l'index tel que $k_h = \sup\{d; d \in \mathcal{K} \text{ et } d < \tau\}$. Comme dans le cas statique, tous les instants k_i correspondent aux échéances distinctes des tâches *rouges*. Les instants k_i sous RTO sont définis par le théorème suivant.

Théorème 29 *Les instants k_i du vecteur $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$ calculés sur $[\tau, kP[$ pour des tâches RTO sont définis comme suit :*

$$k_i = (x.s_j + (k + pos_j(\tau))\%s_j).P_j \tag{III.12}$$

avec $x = 0, ..., (\frac{P}{s_j P_j} - 1)$, $j = 1, ..., n$ et $k = 1, ..., m$ où $m = s_j - 1$.

Preuve :

Sans décalage au niveau de la séquence RTO, les échéances sur requêtes rouges observées sur $[0, s_j P_j]$ sont situées aux instants kp_j, avec $1 \leq k \leq (s_j - 1)$. Soit $pos_j(\tau)$ le décalage dans le temps à l'instant τ associé à la tâche T_j, avec $0 \leq pos_j(\tau) \leq (s_j - 1)$. Chaque décalage dans la séquence des instances repousse les échéances de l'ensemble des requêtes dans le temps d'une quantité P_j. Ainsi, les échéances sur requêtes rouges sur $[0, s_j P_j]$ sont redéfinies aux instants $kP_j + pos_j(\tau)p_j$. Par conséquent, si l'on considère les $\frac{P}{s_j P_j}$ séquences unitaires identiques et consécutives sur $[0, P[$, on obtient $(x.s_j + k + pos_j(\tau)).P_j$ avec $0 \leq x \leq (\frac{P}{s_j P_j} - 1)$, $1 \leq k + pos_j \leq (s_j - 1)$. □

2.1.2.4 Calcul du vecteur dynamique des temps creux sur $[\tau, kP[$.

Le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ traduit les durées des temps creux associés aux instants consignés dans le vecteur $\mathcal{K}(\tau)$. $\Delta_i^*(\tau)$ représente la longueur du temps creux débutant au temps k_i. Par ailleurs, chaque tâche périodique T_i est caractérisée par la quantité de processeur $A_i(\tau)$ déjà allouée à son instance courante au temps τ. Cette instance possède également une échéance dynamique notée d_i. On note M, la plus grande échéance parmi toutes les instances de tâches périodiques actives et l'on figure dans le vecteur des temps creux l'indice f tel que $k_f = \min\{k_i; k_i > M\}$. $\mathcal{D}^*(\tau)$ est alors complètement défini par la relation de récurrence suivante décrite dans le théorème qui suit :

Théorème 30 *Les durées des temps creux du vecteur* $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ *calculés sur* $[\tau, kP[$ *pour des tâches RTO sont définies comme suit :*

$$\Delta_i^*(\tau) = \Delta_i^*, \quad pour\ i = q\ à\ f \tag{III.13}$$

$$\Delta_i^*(\tau) = \sup(0, F_i(\tau)), \quad pour\ i = f-1\ à\ h \tag{III.14}$$

$$avec \quad F_i(\tau) = (P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{pos_j(\tau)P_j}{s_jP_j} \rceil) c_j$$

$$+ \sum_{\substack{j=1 \\ d_j > k_i}}^{n} A_j(\tau) - \sum_{k=i+1}^{q} \Delta_k^*(\tau)$$

Preuve :

La séquence RTO observée sur $[k_i, P[$ est identique à la séquence RTO observée sur $[k_i\text{-}pos_j(\tau)P_j, P' - pos_j(\tau)P_j]$ dans laquelle la dernière instance de la sous-hyperpériode P est bleue et où l'on tient compte du décalage $pos_j(\tau)$, $0 \leq pos_j(\tau) \leq (s_j - 1)$, associée à la tâche T_j. L'évaluation des pertes sur cet intervalle correspond au nombre d'instances bleues présentes dans la séquence RTO considérée sur $[k_i - pos_j(\tau)P_j, P - pos_j(\tau)P_j]$. Cette quantité est égale au nombre de pertes observées sur l'intervalle $P - (k_i - pos_j(\tau)P_j)$ auquel on soustrait le nombre de pertes observées sur l'intervalle $P - (P - pos_j(\tau)P_j)$. Les pertes totales par tâche sur $[k_i, P[$ s'élèvent alors à $\lceil \frac{P-(k_i-pos_j(\tau)P_j)}{s_jP_j} \rceil c_j - \lceil \frac{P-(P-pos_j(\tau)P_j)}{s_jP_j} \rceil c_j$, soit après simplifications $(\lceil \frac{P-k_i+pos_j(\tau)P_j}{s_jP_j} \rceil - \lceil \frac{pos_j(\tau)P_j}{s_jP_j} \rceil) c_j$.

Si $d_j < k_i$, le temps total d'exécution requis par la tâche T_j sur $[k_i, P[$, en tenant compte des pertes, est donné par $(\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{pos_j(\tau)P_j}{s_jP_j} \rceil) c_j$. Sinon, il est donné par $(\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{pos_jP_j(\tau)}{s_jP_j} \rceil) c_j - A_j(\tau)$ puisque la requête courante de T_i a été partiellement exécutée. La sommation des $A_j(\tau)$ s'effectue sur les instances rouges qui ont entamé ou terminé leur exécution à l'instant τ, et uniquement sur les instances bleues qui ont terminé leur exécution à l'instant τ. En effet, étant donné que ces instances bleues ont réussi leur exécution, les instances suivantes seront également bleues. Ainsi, ces instances bleues décalées à $\tau + P_i$ sont considérées comme des instances rouges dans la séquence EDL simulée sur la base RTO à l'instant τ. Par conséquent, la durée totale des temps creux sur $[k_i, P[$ est donnée par :
$(P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+pos_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{pos_j(\tau)P_j}{s_jP_j} \rceil) c_j + \sum_{\substack{j=1, d_j > k_i \\ bleue\ reussie/rouge}}^{n} A_j(\tau)$. □

L'établissement optimisé de la séquence EDL statique pour un modèle de tâche RTO est obtenue en $O(Nn)$ opérations dans le pire-cas, où N représente le nombre de requêtes périodiques rouges sur l'intervalle $[0, P[$ avec $P = ppcm(P_1, P_2, ..., P_n)$. Cette complexité dépend uniquement des périodes des tâches périodiques. Rappelons que la méthode de calcul non optimisée exposée dans le paragraphe 2.1.1 présentait une complexité en $O(N'n)$ où N' représente le nombre de requêtes périodiques rouges sur l'intervalle $[0, P'[$ avec $P' = ppcm(s_1P_1, s_2P_2, ..., s_nP_n)$. Le gain, en termes du nombre d'opérations de la version optimisée par rapport à la version non optimisée est alors d'autant plus important que les pertes autorisées sont faibles (s_i grand). A titre illustratif, pour un ensemble de tâches définies avec un paramètre de pertes uniforme $s_i = 10$, le coût calculatoire relatif au calcul des temps creux sous la version optimisée est réduit d'un facteur 10 par rapport à la méthode non optimisée.

En ce qui concerne la complexité algorithmique du calcul du vecteur dynamique des temps creux, on observe que la détermination des temps creux avec un modèle de tâches RTO est la même que celle observée avec un modèle de tâches classiques, soit en $O(\lceil \frac{R}{p} \rceil n)$ où n désigne le nombre de tâches périodiques, R la plus grande échéance parmi les tâches actives et p la plus petite période.

Quant à la complexité spatiale, celle-ci est déterminée par la taille du vecteur statique des temps creux. Dans le pire-cas, ce dernier est en $O(N)$. Là encore, le gain calculatoire de la méthode optimisée par rapport à la méthode basique est significatif. Remarquons que pour des configurations de tâches périodiques possédant un paramètre de pertes uniforme, la complexité spatiale est réduite d'un facteur égal à s_i.

2.2 Application de l'algorithme EDL au modèle BWP

Dans cette partie nous exposons l'adaptation des équations de calcul des temps creux sous EDL d'un modèle de tâches périodiques sans pertes à un modèle de tâches BWP.

Le calcul des temps creux effectué en-ligne diffère de celui présenté pour RTO, dans la mesure où il doit prendre en compte la présence éventuelle d'instances de tâches bleues à un instant τ donné. Les équations pour le calcul en-ligne des temps creux sous BWP sont décrites ci-après. Seule la version optimisée du calcul est présentée.

2.2.1 Calcul du vecteur dynamique des échéances sur $[\tau, kP[$

La différence ici vis à vis du modèle RTO, réside dans le fait qu'à l'instant $t = \tau$, l'ensemble des instances réveillées n'est pas uniformément rouge. En effet, dans le modèle BWP, lorsqu'il n'y a aucune instance rouge à exécuter, l'ordonnanceur tente d'exécuter des instances bleues qui, par définition, peuvent être abandonnées à tout moment. De plus, si une instance bleue réussit, la prochaine instance de la tâche est encore bleue, introduisant un décalage par rapport à la séquence RTO d'origine. Ce décalage dans le temps (*move forward*) à l'instant τ est modélisé par un paramètre $m_i(\tau)$ associé à la tâche T_i, qui est incrémenté modulo s_i, à chaque fois qu'une instance bleue s'accomplit dans le respect de son échéance.

Le vecteur dynamique des échéances $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$ représente les instants supérieurs ou égaux à τ (le temps courant) précédant un temps creux. Soit h l'index tel que $k_h = \sup\{d; d \in \mathcal{K} \text{ et } d < \tau\}$. Comme dans le cas de RTO, tous les instants k_i correspondent aux échéances distinctes des tâches *rouges*. Les instants k_i sous BWP sont définis par le théorème suivant.

Théorème 31 *Les instants k_i du vecteur $\mathcal{K}(\tau) = (\tau, k_{h+1}, ..., k_i, ..., k_q)$ calculés sur $[\tau, kP[$ pour des tâches BWP sont définis comme suit :*

$$k_i = (x.s_j + (k + m_j(\tau) + pos_j(\tau))\%s_j).P_j \tag{III.15}$$

avec $x = 0, ..., (\frac{P}{s_j P_j} - 1)$, $j = 1, ..., n$ et $k = 1, ..., m$ où $m = s_j - 1$.

Preuve :

Sans décalage au niveau de la séquence BWP, les échéances sur requêtes rouges observées sur $[0, s_jP_j]$ sont situées aux instants kp_j, avec $1 \leq k \leq (s_j - 1)$. Soit $m_j(\tau) + pos_j(\tau)$ le décalage dans le temps à l'instant τ associé à la tâche T_j qui prend en compte non seulement le décalage lié aux succès des instances bleues mais aussi celui lié au calcul de EDL ramené sur P, avec $0 \leq m_j(\tau) \leq (s_j - 1)$ et $0 \leq pos_j(\tau) \leq (s_j - 1)$. Chaque décalage dans la séquence des instances repousse les échéances de l'ensemble des requêtes dans le temps d'une quantité P_j. Ainsi, les échéances sur requêtes rouges sur $[0, s_jP_j]$ sont redéfinies aux instants $kP_j + m_j(\tau) + pos_j(\tau)p_j$. Par conséquent, si l'on considère les $\frac{P}{s_jP_j}$ séquences unitaires identiques et consécutives sur $[0, P[$, on obtient $k_i = (x.s_j + k + m_j(\tau) + pos_j(\tau)).P_j$ avec $0 \leq x \leq (\frac{P}{s_jP_j} - 1)$, $1 \leq k + m_j(\tau) + pos_j \leq (s_j - 1)$. □

2.2.2 Calcul du vecteur dynamique des temps creux sur $[\tau, kP[$

Le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ traduit les durées des temps creux associés aux instants consignés dans le vecteur $\mathcal{K}(\tau)$. $\Delta_i^*(\tau)$ représente la longueur du temps creux débutant au temps k_i. Par ailleurs, chaque tâche périodique T_i est caractérisée par la quantité de processeur $A_i(\tau)$ déjà allouée à son instance courante au temps τ. Cette instance possède également une échéance dynamique notée d_i. $\mathcal{D}^*(\tau)$ est alors complètement défini par la relation de récurrence suivante décrite dans le théorème qui suit :

Théorème 32 *Les durées des temps creux du vecteur* $\mathcal{D}^*(\tau) = (\Delta_h^*(\tau), \Delta_{h+1}^*(\tau), ..., \Delta_i^*(\tau), ..., \Delta_q^*(\tau))$ *calculés sur* $[\tau, kP[$ *pour des tâches BWP sont définies comme suit :*

$$\Delta_i^*(\tau) = \sup(0, F_i(\tau)), \quad \textit{pour } i = q \textit{ à } \tau \tag{III.16}$$

$$\textit{avec} \quad F_i(\tau) = (P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P - k_i}{P_j} \rceil - \lceil \frac{P - k_i + shift_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{shift_j(\tau)P_j}{s_jP_j} \rceil) c_j$$

$$+ \sum_{\substack{j=1 \\ d_j > k_i}}^{n} A_j(\tau) - \sum_{k=i+1}^{q} \Delta_k^*(\tau)$$

$$et \quad shift_j(\tau) = (m_j(\tau) + pos_j(\tau))\%s_i.$$

Preuve :

La séquence BWP observée sur $[k_i, P[$ est identique à la séquence RTO observée sur $[k_i - shift_j(\tau)P_j,\ P' - shift_j(\tau)P_j]$ dans laquelle la dernière instance de la sous-hyperpériode P est bleue et où l'on tient compte du décalage à la fois du décalage $m_j(\tau)$ lié aux succès des instances bleues, mais aussi celui lié au calcul de EDL ramené sur P à savoir $pos_j(\tau)$, avec $0 \leq m_j(\tau) \leq (s_i - 1)$, $0 \leq pos_j(\tau) \leq (s_i - 1)$ et $shift_j(\tau) = (m_j(\tau) + pos_j(\tau))\%s_i$. L'évaluation des pertes sur cet intervalle correspond au nombre d'instances bleues présentes dans la séquence RTO considérée sur $[k_i - shift_j(\tau)P_j, P - shift_j(\tau)P_j]$. Cette quantité est égale au nombre de pertes observées sur l'intervalle $P - (k_i - shift_j(\tau)P_j)$ auquel on soustrait le nombre de pertes observées sur l'intervalle $P - (P - shift_j(\tau)P_j)$. Les pertes totales par tâche sur $[k_i, P[$ s'élèvent alors

à $\lceil \frac{P-(k_i-shift_j(\tau)P_j)}{s_jP_j} \rceil c_j - \lceil \frac{P-(P-shift_j(\tau)P_j)}{s_jP_j} \rceil c_j$, soit après simplifications $(\lceil \frac{P-k_i+shift_j(\tau)P_j}{s_jP_j} \rceil - \lceil \frac{shift_j(\tau)P_j}{s_jP_j} \rceil)c_j$.

Si $d_j < k_i$, le temps total d'exécution requis par la tâche T_j sur $[k_i, P[$, en tenant compte des pertes, est donné par $(\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+shift_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{shift_j(\tau)P_j}{s_jP_j} \rceil)c_j$. Sinon, il est donné par $(\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+shift_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{shift_jP_j(\tau)}{s_jP_j} \rceil)c_j - A_j(\tau)$ puisque la requête courante de T_i a été partiellement exécutée. La sommation des $A_j(\tau)$ s'effectue sur les instances rouges qui ont entamé ou terminé leur exécution à l'instant τ, et uniquement sur les instances bleues qui ont terminé leur exécution à l'instant τ. En effet, étant donné que ces instances bleues ont réussi leur exécution, les instances suivantes seront également bleues. Ainsi, ces instances bleues décalées à $\tau + P_i$ sont considérées comme des instances rouges dans la séquence EDL simulée sur la base RTO à l'instant τ. Par conséquent, la durée totale des temps creux sur $[k_i, P[$ est donnée par :
$(P - k_i) - \sum_{j=1}^{n} (\lceil \frac{P-k_i}{P_j} \rceil - \lceil \frac{P-k_i+shift_j(\tau)P_j}{s_jP_j} \rceil + \lceil \frac{shift_j(\tau)P_j}{s_jP_j} \rceil)c_j + \sum^{n}_{\substack{j=1, d_j > k_i \\ bleue\ reussie/rouge}} A_j(\tau)$. □

2.2.3 Illustration du calcul en-ligne de f^{EDL} pour des tâches BWP

Nous pouvons à présent appliquer ces résultats à l'ensemble $\mathcal{T} = \{T_1(3, 10, 2), T_2(3, 6, 2)\}$ avec $s_1 = s_2 = 2$. Cet ensemble de tâches périodiques BWP est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 13$. Supposons que l'on souhaite calculer le maximum de temps processeur libre dans l'intervalle $[13, 30[$ relativement à l'ensemble $\mathcal{T}$. On note qu'à $\tau = 13$, la tâche T_1 n'est pas encore terminée, ce qui implique que la requête suivante est rouge. A partir des formules III.15 et III.16, on extrait le vecteur dynamique des échéances $\mathcal{K}(\tau) = (15, 18)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (3, 6)$, comme l'illustre la figure III.9.

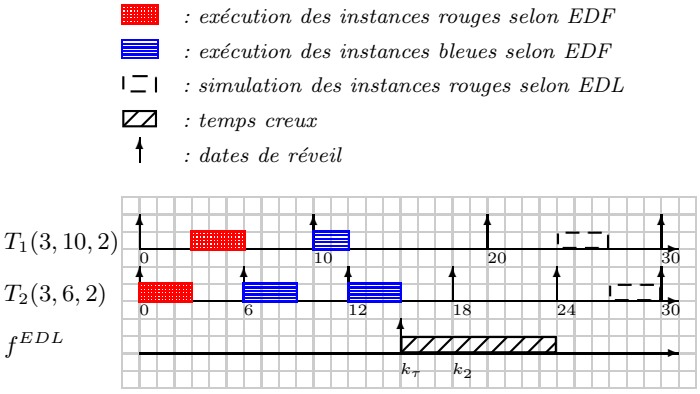

FIG. III.9 – Calcul optimisé des temps creux dynamiques à l'instant $\tau = 15$ sous BWP

En ce qui concerne la complexité algorithmique du calcul du vecteur dynamique des temps creux, on observe que la détermination des temps creux avec un modèle de tâches BWP est en $O(N)$ où N représente le nombre de requêtes périodiques rouges sur l'intervalle $[\tau, kP[$. Quant à la complexité spatiale, celle-ci est déterminée par la taille du

vecteur statique des temps creux. Dans le pire-cas, ce dernier est en $O(N)$.

Dans la section suivante, nous introduisons deux nouveaux algorithmes d'ordonnancement avec pertes au sens Skip-Over. Ceux-ci constituent une amélioration des ordonnanceurs de base RTO et BWP, au niveau de la QoS globale observée pour les tâches périodiques. Le fonctionnement de ces 2 algorithmes est basé sur un ordonnancement EDL adapté au modèle de tâches Skip-Over, tel que nous avons pu le décrire précédemment.

3 Amélioration des ordonnanceurs Skip-Over

3.1 L'algorithme RLP

Le principal inconvénient de BWP réside dans le fait que les tâches bleues sont exécutées en tâches de fond des tâches rouges et par conséquent avec une priorité inférieure. Ceci conduit ainsi à abandonner des tâches bleues partiellement ou quasi-totalement exécutées, et donc à gaspiller du temps processeur et ce, au bénéfice de tâches rouges éventuellement moins urgentes.

L'objectif de l'algorithme *Red tasks as Late as Possible* (*RLP*) que nous proposons consiste à supprimer l'inconvénient de BWP en retardant au maximum l'exécution des instances rouges, de manière à minimiser le nombre d'instances de tâches bleues abandonnées. En effet, sur la figure III.2 illustrant le comportement de BWP, nous pouvons observer que du temps processeur est parfois gaspillé comme c'est le cas de la seconde instance de la tâche T_3 où 5 unités de temps sont effectivement perdues. Le but principal est donc d'améliorer la QoS (*i.e.,* le nombre total d'instances réussies) observée au niveau des tâches périodiques, en avançant autant que possible l'exécution des instances bleues. Dans cette perspective, la spécification du comportement de l'algorithme d'ordonnancement *RLP*, qui est un algorithme d'ordonnancement dynamique, est la suivante :

(1) s'il n'y a pas d'instances de tâches bleues dans le système, les instances de tâches rouges sont ordonnancées au plus tôt selon l'algorithme *EDF*,

(2) si des instances de tâches bleues sont présentes dans le système, celles-ci sont ordonnancées au plus tôt selon l'algorithme *EDF* (notons que cet ordonnancement pourrait être effectué selon une autre heuristique), tandis que les instances de tâches rouges sont exécutées au plus tard selon l'algorithme EDL.

Les conflits d'échéances sont systématiquement résolus en faveur de la tâche possédant la date de réveil la plus ancienne. L'idée principale de cette approche est de tirer profit de la laxité des instances de tâches rouges. La détermination de l'instant de début d'exécution au plus tard de chacune des requêtes rouges d'un ensemble de tâches périodiques, nécessite une construction préliminaire de la séquence en utilisant une variante de l'algorithme EDL qui prend en compte les pertes autorisées au niveau des tâches, comme décrit dans le paragraphe précédent. L'établissement de cette séquence a été présenté en détails dans la section 2.2. Dans l'ordonnancement EDL établi au temps

t, nous supposons que l'instance suivant immédiatement une instance bleue active à l'instant t, est rouge. En effet, aucune garantie ne peut être apposée à une instance de tâche bleue concernant le fait qu'elle s'accomplisse dans le respect de son échéance. Nous illustrons notre propos sur l'exemple suivant : soit une configuration de tâches $\mathcal{T} = \{T_1(6,10), T_2(3,6)\}$, la séquence EDL établie à l'instant $t = 12$ correspondant à l'occurrence d'une instance de tâche bleue pour la tâche T_2, est représentée sur la figure III.10.

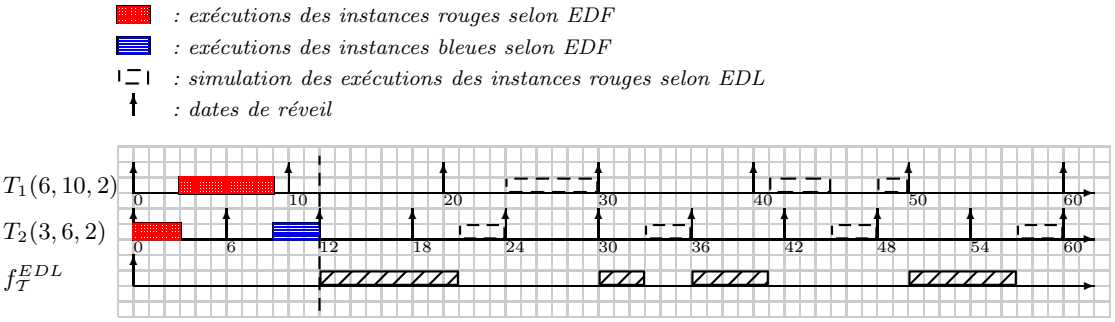

FIG. III.10 – Calcul des temps creux sous RLP au temps $t = 12$

Dans la séquence EDL calculée, les temps creux matérialisés par la fonction $f_{\mathcal{T}}^{EDL}$ correspondent aux instants disponibles pour l'exécution des instances bleues. Silly dans [Sil99] a prouvé que le calcul en-ligne des temps creux est uniquement requis aux instants correspondant à l'occurrence d'une requête alors qu'il n'y en avait pas au préalable dans le système. Dans notre cas, la séquence EDL est construite non seulement lorsqu'une instance de tâche bleue est réveillée (et qu'il n'y en avait aucune au préalable) mais aussi suite à la terminaison correcte d'une instance de tâche bleue. En effet, l'instance d'une tâche suivant une instance bleue réussie est également bleue, ce qui entraîne un décalage de la séquence des instances rouges simulées pour la tâche et donc un recalcul de la séquence EDL.

Notons que les tâches bleues sont exécutées dans les temps creux calculés par EDL et qu'elles possèdent désormais la même importance que les tâches rouges. Nous nous affranchissons ici du modèle de hiérarchisation des types d'instances présent dans l'algorithme BWP. Rappelons que ce dernier assigne toujours la priorité la plus élevée aux tâches rouges.

3.1.1 Description algorithmique

L'algorithme d'ordonnancement RLP, tout comme BWP, est chargé de gérer 3 listes de tâches : une liste de tâches en attente et 2 listes de tâches prêtes correspondant respectivement aux types de tâches rouges et bleues.

L'algorithme RLP fonctionne en 3 phases. Dans une première phase, il examine la liste des tâches bleues prêtes dans le but d'abandonner, la ou les tâches bleues dont l'échéance absolue est supérieure ou égale au temps courant.

La seconde phase de l'algorithme consiste à parcourir la liste des tâches en attente et à réveiller, le cas échéant, la ou les tâches dont la date de réveil est inférieure ou égale au temps courant. Les tâches rouges sont placées dans la liste des tâches rouges prêtes lorsqu'il n'y a pas de temps creux au temps courant, contrairement aux tâches bleues qui sont placées dans la liste des tâches bleues prêtes, uniquement lorsqu'il y a un temps creux.

Enfin, l'exécution de la dernière phase dépend de l'état de la liste des tâches bleues prêtes. Si cette liste est non vide et si le temps courant correspond à un temps creux, alors toutes les tâches rouges prêtes sont suspendues et replacées dans la liste des tâches en attente.

La description algorithmique de RLP est fournie ci-dessous dans l'algorithme 3.

Algorithme 3
RLP_schedule*(t : temps courant)*
début
*/*Recherche des abandons de tâches bleues associés au temps courant*/*
tantque *(liste de tâches bleues prêtes=***not***(∅))*
si *(tâche→ r_i +tâche→ D_i < t)*
break
finsi
Retirer la tâche de la liste des tâches bleues prêtes
tâche→ r_i = tâche→ r_i + tâche→ P_i
tâche→ s_{i_dyn} = 1
Insérer la tâche dans la liste des tâches en attente
fintantque
*/*Recherche des réveils de tâches associés au temps courant*/*
tantque *(liste de tâches en attente=***not***(∅))*
si *(tâche→ r_i > t)*
break
finsi
si *((tâche→ s_{i_dyn} < tâche→ s_i)* **and** *(f_EDL(t)=0))*
Retirer la tâche de la liste des tâches en attente
Insérer la tâche dans la liste des tâches rouges prêtes
sinon
si *(liste des tâches bleues prêtes=∅)*
Calculer f_EDL
finsi
si *(f_EDL(t) !=0)*
Retirer la tâche de la liste des tâches en attente
Insérer la tâche dans la liste des tâches bleues prêtes
finsi

finsi
tâche$\rightarrow s_{i\_dyn}$ =*tâche*$\rightarrow s_{i\_dyn} + 1$
fintantque
si *((liste des tâches bleues prêtes=***not***(∅)) and (f_EDL(t) !=0))*
*/*Suspension des tâches rouges prêtes*/*
tantque *(liste des tâches rouges prêtes=***not***(∅))*
Retirer la tâche de la liste des tâches rouges prêtes
Insérer la tâche dans la liste des tâches en attente
fintantque
finsi
fin

3.1.2 Illustration

Considérons de nouveau l'ensemble de tâches $\mathcal{T}$ défini dans la Table III.1. L'ordonnancement de cette configuration de tâches par RLP est illustré sur la figure III.11.

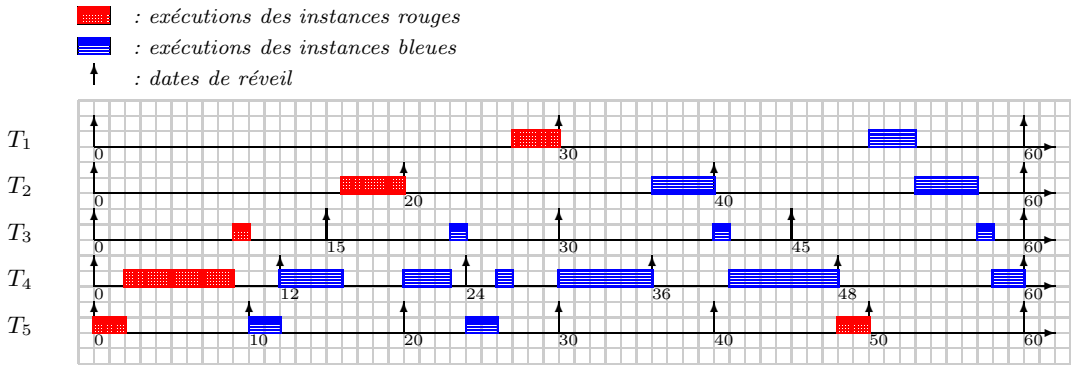

FIG. III.11 – Illustration de l'algorithme d'ordonnancement RLP

Nous pouvons observer dans cet exemple que, grâce à l'algorithme RLP, le nombre de violations d'échéances relatives à des instances de tâches bleues a été ramené à 3. Celles-ci surviennent aux instants $t = 40$ (tâche T_5), et $t = 60$ (tâches T_4 et T_5). Observons que la première instance de T_4 qui ne réussissait pas à s'exécuter dans le respect de son échéance sous BWP (cf. figure III.2), dispose d'assez de temps pour s'achever avec succès sous RLP où les exécutions des premières instances rouges de T_2 et T_1 sont différées dans le temps.

Jusqu'au temps $t = 10$, les instances de tâches rouges sont ordonnancées au plus tôt. A partir de l'instant $t = 10$ jusqu'à la fin de l'hyperpériode, les instances de tâches rouges s'exécutent effectivement au plus tard, du fait de la présence d'instances de tâches bleues dans le système. Grâce à ce nouveau mode d'ordonnancement, la QoS des tâches

périodiques est améliorée. Cette amélioration sera quantifiée par une étude de simulation reportée plus loin.

3.2 L'algorithme RLP/T

3.2.1 Description

L'inconvénient de RLP réside dans le fait que l'algorithme tente d'exécuter au plus tôt les tâches bleues au risque qu'elles soient interrompues avant d'avoir pu terminer leur exécution avant échéance, engendrant ainsi un gaspillage de la capacité de traitement. Ce constat nous a conduit à proposer un nouvel algorithme nommé *RLP/T* (*Red tasks as Late as Possible with blue acceptance Test*) visant à pallier à cet inconvénient.

L'algorithme RLP/T a également été conçu pour maximiser la QoS observée au niveau d'ensembles de tâches périodiques avec des contraintes de pertes. Il fonctionne de la manière suivante : à leur arrivée, les instances de tâches rouges entrent directement dans le système tandis que les instances de tâches bleues intègrent le système sur test d'acceptation. Une fois qu'elles ont été acceptées, les instances de tâches bleues sont ordonnancées de façon conjointe avec les instances de tâches rouges selon EDF. Les conflits d'échéances sont systématiquement résolus en faveur de la tâche possédant la date de réveil la plus ancienne vis-à-vis du temps courant.

A chaque fois qu'une nouvelle instance de tâche bleue entre dans le système, les temps creux d'un ordonnanceur EDL sont calculés. Dans l'ordonnancement EDL établi au temps t, nous supposons que l'instance suivant immédiatement une instance bleue active à l'instant t, est également bleue. En effet, toutes les instances de tâches bleues préalablement acceptées au sein du système, sont garanties de s'exécuter dans le respect de leur échéance du fait du test d'acceptation mis en œuvre. Nous illustrons notre propos sur l'exemple suivant : soit une configuration de tâches $\mathcal{T} = \{T_1(6,10), T_2(3,6)\}$, la séquence EDL établie à l'instant $t = 12$ correspondant à l'occurrence d'une instance de tâche bleue pour la tâche T_2 est représentée sur la figure III.12.

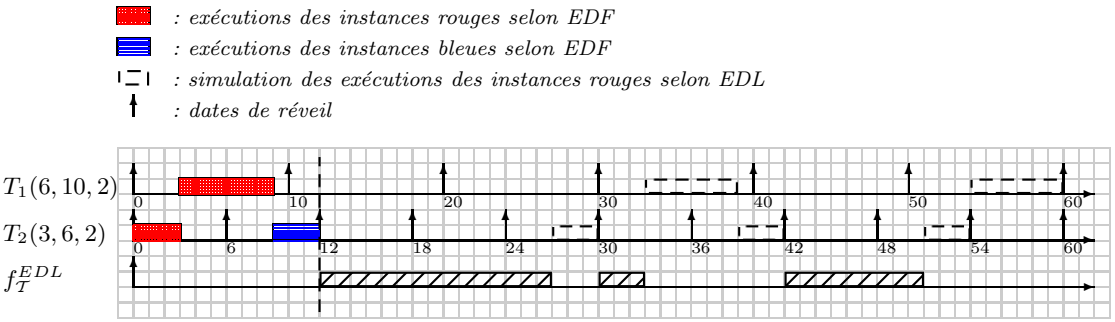

FIG. III.12 – Calcul des temps creux sous RLP/T au temps $t = 12$

Dans la séquence EDL calculée, nous pouvons remarquer que l'instance de tâche de T_1 réveillée à l'instant $t = 20$ est supposée bleue. Ceci s'explique par le fait qu'étant donné que l'instance de tâche de T_1 réveillée au temps $t = 10$ était bleue et avait été acceptée (donc garantie), l'instance suivante, selon la définition du modèle Skip-Over, est également bleue. Un raisonnement identique peut être appliqué à l'instance de T_2 occurrente à $t = 12$. Nous faisons l'hypothèse a priori de son acceptation, c'est pour cette raison que l'instance de tâche de T_2 réveillée à l'instant $t = 18$ est supposée bleue. La séquence EDL est construite sur la base d'une succession d'instances équivalente à un ordonnancement RTO dans lequel les tâches bleues sont systématiquement rejetées.

3.2.2 Test d'acceptation des instances bleues sous RLP/T

Nous présentons ici le nouvel algorithme de test pour l'acceptation des tâches bleues selon le schéma RLP/P qui, étant donné n'importe quelle instance B de tâche bleue est capable de répondre à la question "B peut-elle être acceptée ?". Notons que B sera acceptée *si et seulement si* il existe une séquence valide, c'est-à-dire un ordonnancement dans lequel B s'exécutera dans le respect de son échéance tandis que les instances de tâches rouges et les instances de tâches bleues préalablement acceptées, respecteront elles-aussi leurs échéances. Ce problème est similaire au problème de l'acceptation d'une tâche apériodique critique parmi un ensemble de tâches périodiques et de tâches apériodiques préalablement acceptées [CC89].

Soit τ le temps courant coïncidant avec l'arrivée d'une instance B de tâche bleue. A son arrivée, l'instance de tâche $B(r, c, d)$ est caractérisée par sa date de réveil r, sa durée d'exécution au pire-cas c, et son échéance d, avec $r + c \leq d$. Le système peut présenter en son sein plusieurs tâches bleues à l'instant τ ; chacune d'elles ayant été acceptée avant τ et n'ayant pas terminé son exécution au temps τ. Soit $\mathcal{B}(\tau) = \{B_i(c_i(\tau), d_i), i = 1 \text{ à } bleue(\tau)\}$ l'ensemble des tâches bleues supportées par le système au temps τ. Le paramètre $c_i(\tau)$ est appelé *durée d'exécution dynamique* et représente la durée d'exécution restante de B_i à τ. Nous supposons de plus que $\mathcal{B}(\tau)$ est ordonné de telle manière que $i < j$ implique $d_i \leq d_j$.

Le test d'acceptation des tâches bleues au sein d'un système ordonnancé selon RLP présenté ci-dessous dans le théorème 33, est basé sur celui établi par Silly et al. [SCE90] pour l'acceptation de requêtes apériodiques critiques dans un système gérant un ensemble de tâches périodiques basiques, c'est-à-dire sans pertes (cf. Chapitre I. Section 3.3.1.4).

Théorème 33 *$B(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $B_i \in \mathcal{B}(\tau) \cup \{B\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{III.17}$$

$\delta_i(\tau)$, appelée laxité de la tâche R_i au temps τ, représente la longueur de l'intervalle de temps qui sépare sa fin d'exécution au plus tôt de son échéance. $\Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i)$ représente la somme totale des temps creux calculée sur l'intervalle $[\tau, d_i]$. La sommation $\sum_{j=1}^{i} c_j(\tau)$

traduit la somme des exécutions restantes à l'instant τ sur l'ensemble des tâches bleues dont l'échéance est inférieure ou égale à d_i.

La procédure implémentant le test d'acceptation fait appel à l'algorithme EDL adapté à des tâches Skip-Over, pour le calcul dynamique de la somme totale des temps creux. Celle-ci est utilisée pour établir la laxité des tâches bleues qui est ensuite comparée à zéro. Par conséquent, le test d'acceptation proposé a une complexité en $O(\lfloor \frac{R}{p} \rfloor n + bleue(\tau))$ dans le pire-cas, où n représente le nombre de tâches périodiques, R la plus grande échéance, p la plus petite période, et $bleue(\tau)$ le nombre de tâches bleues actives à l'instant τ dont l'échéance est supérieure ou égale à l'échéance de la tâche bleue occurrente. Soulignons que ce test d'acceptation pourrait être implémenté en $O(n + bleue(\tau))$ en considérant des structures de données plus complexes basées sur la mise à jour de tables de laxités, comme cela a pu être démontré dans [TLS+94].

Le pseudo-code du test d'acceptation mis en œuvre par l'algorithme RLP/T est fournie ci-dessous dans l'algorithme 9.

Algorithme 4
RLP/T_blue_acceptance_test*(t : temps courant)*
début
soit $\mathcal{B}(t)$ l'ensemble des tâches bleues actives au temps t
tantque *(une tâche bleue B survient au temps t)* **faire**
ordonnançable =1
Choisir la tâche bleue B_j de $\mathcal{B}(t) \cup \{B\}$ possédant la plus grande échéance
Calculer $f^{EDL}(t, d_j)$
pour *toutes les tâches bleues B_i de $\mathcal{B}(t) \cup \{B\}$ telles que $d_i \geq d$* **faire**
Calculer la laxité $\delta_i(t)$
si *($\delta_i(t) < 0$)*
ordonnançable = 0 et STOP
finsi
finpour
return ordonnançable
fintantque
fin

Appliquons cet algorithme sur l'exemple de la figure III.12 en calculant les laxités des instances bleues de T_1 et T_2 au temps $t = 12$ correspondant à l'occurrence d'une instance de tâche bleue pour la tâche T_2. La somme des temps creux dans les intervalles $[12, 18]$ et $[12, 20]$ est donné par $\Omega^{EDL}_{\mathcal{T}(12)}(12, 18) = 6$ et $\Omega^{EDL}_{\mathcal{T}(12)}(12, 20) = 8$ respectivement. La durée d'exécution requise par les tâches bleues sur l'intervalle $[12, 18]$ est égale à $\sum_{j=1}^{i} c_j(12) = 3$ (l'instance bleue de T_2 réveillée à $t = 12$ a besoin de 3 unités de temps pour finir son exécution). Sur l'intervalle $[12, 20]$, cette quantité est égale à $\sum_{j=1}^{i} c_j(12) = 9$ (les instances bleues de T_1 et T_2 réveillées aux temps $t = 10$ et $t = 12$ ont besoin de 6 et 3 unités de temps respectivement pour terminer leur exécution). Par conséquent, la laxité de l'instance de tâche bleue de T_2 au temps $t = 12$ est égale à $\delta_i(12) = \Omega^{EDL}_{\mathcal{T}(12)}(12, 18) -$

$\sum_{j=1}^{i} c_j(12) = 6-3 = 3$, ce qui est acceptable. Par contre, la laxité de l'instance de tâche bleue de T_1 au temps $t = 12$ est égale à $\delta_i(12) = \Omega^{EDL}_{\mathcal{T}(12)}(12, 20) - \sum_{j=1}^{i} c_j(12) = 8 - 9 = -1$. Cette valeur négative qui indique qu'il manque une unité de temps à l'instance bleue de T_1 pour s'accomplir dans le respect de son échéance, implique que l'instance bleue occurrente (instance de T_2) doit être rejetée au temps $t = 12$.

3.2.3 Illustration

L'algorithme d'ordonnancement RLP/T est illustré sur la figure III.13 avec le même ensemble de tâches $\mathcal{T}$ que celui utilisé pour présenter les algorithmes RTO, BWP et RLP. Ce dernier est décrit précédemment dans la Table III.1.

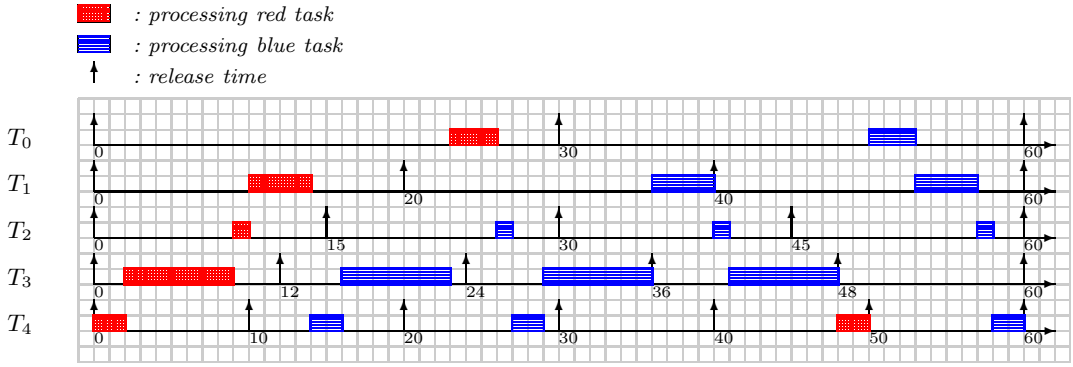

FIG. III.13 – RLP/T scheduling algorithm ($s_i = 2$)

RLP/T apparaît ici meilleur que RLP et BWP en termes de qualité de service offerte au niveau des tâches périodiques. Seulement 2 violations d'échéances relatives à des instances de tâches bleues sont observées : au temps $t = 40$ (tâche T_4) et $t = 60$ (tâche T_3). Le test d'acceptation contribue à sauvegarder le temps processeur perdu à débuter l'exécution de tâches bleues qui ne sont pas capables de s'exécuter dans le respect de leur échéance. Comme nous pouvons l'observer, dans le cas de RLP (cf. figure III.11), l'instance bleue de T_3 réveillée au temps $t = 48$ est abandonnée au temps $t = 60$ (2 unités de temps sont ainsi gaspillées). Notons que le rejet de cette même instance bleue, effectué avec RLP/T, contribue à sauver du temps CPU utilisé pour l'exécution réussie de l'instance bleue de T_4 réveillée au temps $t = 50$.

Dans la section suivante, nous quantifions de manière plus précise le gain de performance de l'algorithme RLP/T vis-à-vis des algorithmes RLP, BWP et RTO.

4 Résultats de simulation

L'objectif de ce paragraphe est de décrire les résultats de simulations effectuées dans le but d'évaluer comparativement la performance des stratégies précédemment décrites. L'objectif est toujours de maximiser le niveau de qualité de service des tâches périodiques, c'est-à-dire la proportion des tâches périodiques qui s'exécutent dans le respect de leur échéance. Les expérimentations en simulation tendent également à évaluer l'impact du paramètre de pertes sur les performances relatives des différentes stratégies.

4.1 Environnement de simulation

L'environnement de simulation consiste en un noyau de simulation (*l'ordonnanceur*) accompagné d'un certain nombre de composantes impliquées dans la gestion et l'analyse des simulations. Le programme de simulation a été implémenté en langage C. L'architecture fonctionnelle du simulateur est représentée sur la figure III.14.

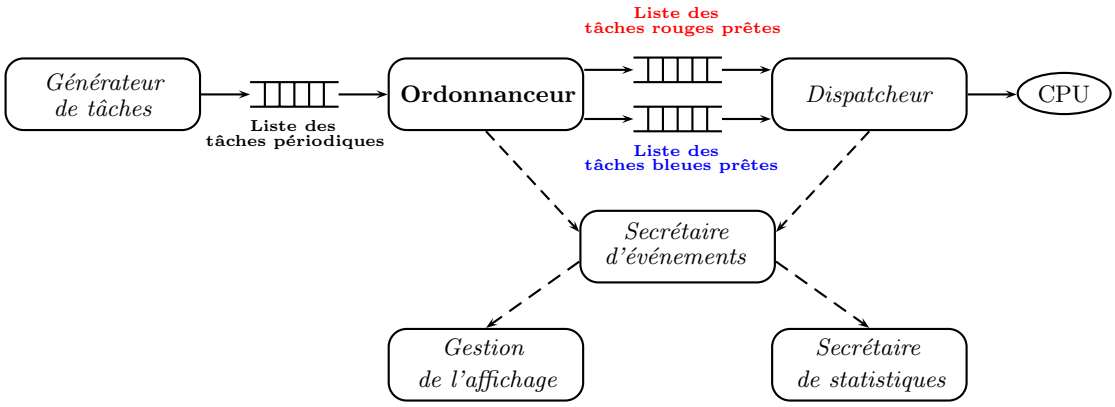

FIG. III.14 – Architecture fonctionnelle du simulateur

Un *générateur de configuration de tâches périodiques* a été conçu sur la base de celui décrit par Martineau dans [Mar94]. Il accepte en entrée plusieurs paramètres : le nombre de tâches n souhaité pour la configuration, le *ppcm* des périodes des tâches, la charge périodique U_p et la valeur s_i des pertes uniformes autorisées au niveau des tâches. En sortie, on obtient une configuration de tâches avec pertes $\mathcal{T} = \{T_i(C_i, P_i, s_i), i = 1 \text{ à } n\}$. Les périodes des tâches sont harmoniques entre elles. Les durées d'exécution des tâches sont générées aléatoirement et sont telles que $\sum_{i=1}^{n} \frac{C_i}{P_i} = U_p$. La valeur du facteur équivalent du processeur U_p^* correspondant à la configuration de tâches établie est également fournie en sortie, de manière à vérifier la faisabilité de la configuration du point de vue de l'exécution des instances rouges uniquement (cf. Théorème 21, Chapitre II., Section 5.1).

Après la génération d'une configuration de tâches, le simulateur ordonnance cette dernière en-ligne pour chaque algorithme d'ordonnancement, ici RTO, BWP, RLP et RLP/T.

Le *dispatcher* lance l'exécution des instances de tâches prêtes placées en tête de liste par le module d'ordonnancement. L'exécution des tâches sur le processeur est une simple abstraction. Le temps est relié à une horloge logique et l'exécution d'une tâche consiste uniquement à décrémenter son compteur d'exécution.

L'analyse de la simulation s'effectue grâce à un *secrétaire d'événements* chargé de consigner dans un fichier les principaux événements intervenus lors de la simulation (réveil/terminaison de tâches, violation d'échéance, etc.). Un *secrétaire de statistiques* rassemble par ailleurs les statistiques d'exécution de la séquence des tâches (taux de respect, taux de préemption, etc.) dans un fichier indépendant.

Enfin, le simulateur propose un module optionnel d'*affichage graphique*. Ce dernier permet de présenter la séquence simulée en figurant le comportement des différentes tâches (réveils, exécutions, préemptions, etc.).

Pour chaque stratégie d'ordonnancement, 50 configurations de tâches différentes ont été générées, chacune d'elles comprenant 10 tâches avec un *ppcm* de périodes égal à 3360. Les simulations ont été effectuées sur 10 hyperpériodes. Les performances de chacun des algorithmes d'ordonnancement ont été évaluées en faisant varier la charge périodique du système ainsi que le facteur de pertes appliqué aux tâches.

4.2 Critères mesurés

Quatre paramètres ont servi à comparer les performances des ordonnanceurs étudiés :

4.2.1 Le taux de respect

Il s'agit du nombre de requêtes dont l'exécution s'est achevée avant son échéance. Ce paramètre représente la qualité de service observée globalement au niveau du système.

$$T_{QoS} = \frac{\text{nombre d'échéances satisfaites}}{\text{nombre total de requêtes}}$$

4.2.2 Le taux de préemption

C'est la moyenne du rapport entre le nombre de préemptions et le nombre de requêtes traitées. Ce facteur permet d'évaluer les surcoûts dus aux changements de contextes. Par définition, il y a préemption lorsque l'exécution d'une tâche est interrompue au profit d'une tâche plus prioritaire. L'exécution de la tâche préemptée est alors reprise plus tard dans le temps.

$$T_{preemption} = \frac{\text{nombre de préemptions}}{\text{nombre total de requêtes}}$$

4.2.3 Le taux de CPU gaspillé

C'est le coût de la tolérance aux pertes, c'est-à-dire le pourcentage de temps passé par le processeur à exécuter des instances de tâches qui n'ont pas servi. Le temps CPU gaspillé se limite au temps passé à exécuter des instances bleues qui ont été abandonnées.

$$T_{CPU\_gaspille} = \frac{\text{temps processeur alloué à des instances bleues abandonnées}}{\text{temps total de simulation}}$$

4.2.4 Le taux d'oisiveté

Il s'agit du pourcentage de temps pendant lequel le processeur est inactif (*idle*).

$$T_{temps\_creux} = \frac{\text{temps processeur inactif}}{\text{temps total de simulation}}$$

4.3 Evaluation du taux de respect

L'évaluation a été effectuée en deux phases. Nous avons tout d'abord comparé les performances des différents algorithmes d'ordonnancement en supposant que toutes les tâches s'exécutent exactement selon leur durée d'exécution au pire-cas (WCET). Puis, nous avons considéré un cadre plus proche de la réalité dans lequel les tâches s'exécutent selon une durée d'exécution effective (Average Case Execution Time notée ACET) inférieure à leur durée d'exécution au pire-cas.

4.3.1 Exécutions sur la base du WCET

Les mesures reposent sur la fraction de tâches périodiques qui s'accomplissent dans le respect de leur échéance. L'évaluation est menée en fonction de la charge périodique U_p appliquée au système. Les résultats obtenus pour $s_i = 2$ (une instance sur deux peut être abandonnée) et $s_i = 6$ (une instance sur six peut être abandonnée) sont décrits sur les figures III.15 et III.16 respectivement.

Les courbes nous montrent que les algorithmes BWP, RLP et RLP/T offrent de meilleures performances que l'algorithme RTO pour lequel le niveau de QoS offert est toujours le même, quelle que soit la charge périodique appliquée. Pour $s_i = 6$, la QoS reste ainsi constante à un taux de $5/6 = 83\%$. RTO peut ainsi servir de référence pour l'évaluation des autres stratégies puisqu'il fournit systématiquement une QoS minimale.

L'avantage de RLP vis-à-vis de BWP est peu important pour de fortes pertes autorisées, et devient davantage significatif pour des pertes plus limitées. Notons que la performance de BWP et RLP est nettement moins bonne que celle de RLP/T. Ce résultat était prévisible dans la mesure où BWP et RLP tentent tous les deux d'exécuter des instances bleues qui n'auront pas forcément assez de temps pour s'accomplir dans le respect de leur échéance. Ce temps CPU gaspillé impute donc nécessairement la qualité de service globale du système. Au contraire, dans le cas de RLP/T, le test d'acceptation

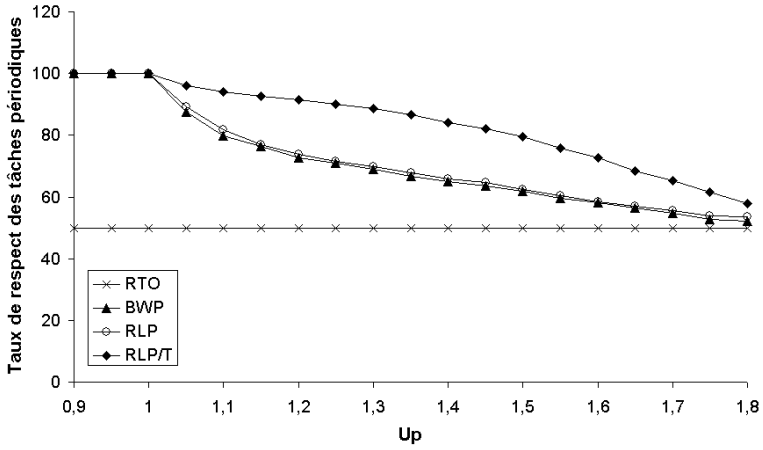

FIG. III.15 – Taux de respect en fonction de U_p ($s_i = 2$)

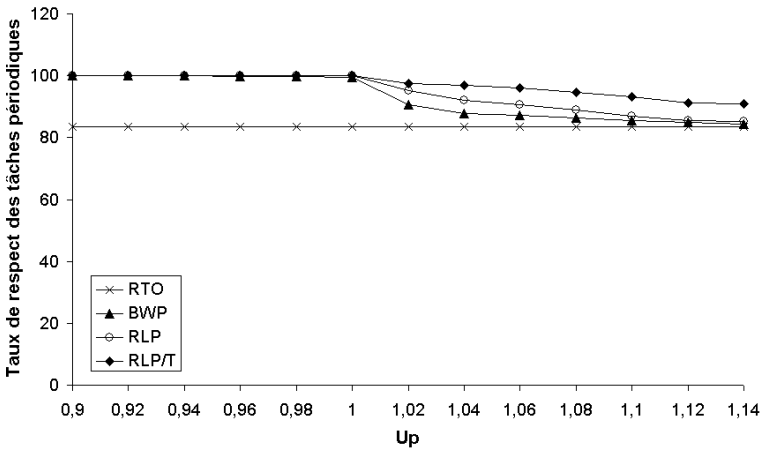

FIG. III.16 – Taux de respect en fonction de U_p ($s_i = 6$)

des bleues mis en œuvre par l'algorithme permet de sauvegarder du temps CPU dont peuvent bénéficier d'autres instances de tâches garanties.

Nous pouvons également observer que le gain de performance de RLP/T sur les autres algorithmes d'ordonnancement est d'autant plus important que la charge périodique U_p est élevée ($U_p < 150\%$). Par exemple, sur la figure III.15, pour une charge $U_p \geq 120\%$, RLP/T surpasse BWP d'un facteur 1/4. De plus, nous pouvons remarquer qu'un faible gradient est associé à la courbe de RLP/T, ce qui n'est pas le cas pour les autres modèles. Pour $U_p = 150\%$ et $s_i = 2$, les niveaux de QoS offerts par RTO et RLP/T sont respectivement égaux à 50% et 84%, ce qui figure la prédominance de RLP/T sur RTO.

La variation du paramètre de pertes nous montre que pour des charges élevées, la QoS des tâches périodiques est d'autant plus améliorée avec RLP/T que le paramètre s_i est petit. A titre illustratif, pour $U_p = 110\%$, l'écart de performance de RLP/T vis-à-vis de BWP dans le cas de tâches périodiques où $s_i = 2$ (cf. figure III.15) est double de celui observé avec des tâches périodiques où $s_i = 6$ (cf. figure III.16). Comme nous pouvons le voir, la plus grande différence de performance entre RLP/T et BWP apparaît non seulement pour des charges élevées mais aussi pour de fortes pertes autorisées.

4.3.2 Exécutions sur la base du ACET

Nous nous plaçons à présent dans le cas où les tâches s'exécutent non plus sur une durée égale à leur WCET mais sur une durée, appelée ACET, inférieure à leur durée d'exécution au pire-cas. Cette durée est définie lors de la phase d'initialisation et est telle que $ACET = k.WCET$ avec $0 < k < 1$. Nous l'avons volontairement considérée identique pour toutes les instances d'une même tâche, de manière à souligner l'impact de ce paramètre sur les performances des ordonnanceurs.

Nous rapportons ci-dessous les résultats de l'évaluation du taux de respect des tâches périodiques en fonction de la charge périodique U_p appliquée au système. Comme précédemment, les mesures ont été effectuées pour des paramètres de pertes distincts ($s_i = 2$ et $s_i = 6$). Pour chaque configuration de pertes, nous présentons les résultats obtenus (cf. figures III.17, III.18, III.19, III.20) avec des durées moyennes d'exécution de 90% et 75% vis-à-vis du WCET.

Nous observons que pour les algorithmes d'ordonnancement BWP, RLP et RLP/T, à charge périodique équivalente, le taux de respect observé ici est meilleur que dans le cas où les tâches s'exécutent selon leur WCET. Ceci s'explique par le fait que la quantité de temps WCET-ACET non utilisé par chaque instance de tâche, représente du temps CPU additionnel utilisé pour exécuter un plus grand nombre d'instances de tâches dans le respect de leurs échéances.

Par ailleurs, il est intéressant de constater qu'à faibles surcharges et dans le cas de fortes pertes autorisées, les algorithmes BWP et RLP offre de meilleures performances que RLP/T. Ceci est dû au fait que le test d'acceptation des bleues sous RLP/T se base sur le WCET des tâches pour vérifier l'ordonnançabilité de l'ensemble courant, la grandeur ACET n'étant bien sûr pas *a priori* connue. Cependant, en réalité les tâches s'exécutant généralement durant un temps inférieur à leur WCET, RLP/T rejette nécessairement des tâches qui pourraient finalement être acceptées si le test d'ac-

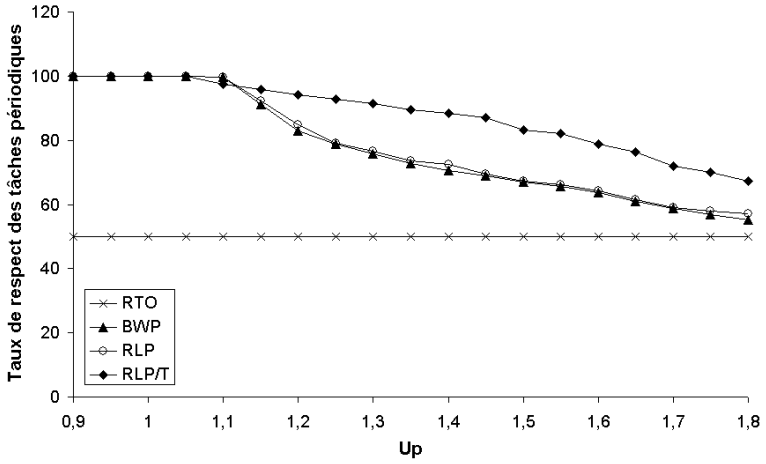

FIG. III.17 – Taux de respect en fonction de U_p ($s_i = 2$, ACET=0.90 WCET)

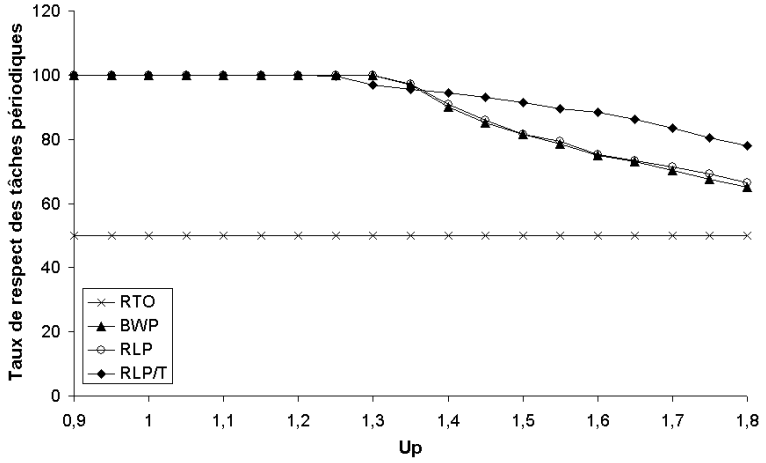

FIG. III.18 – Taux de respect en fonction de U_p ($s_i = 2$, ACET=0.75 WCET)

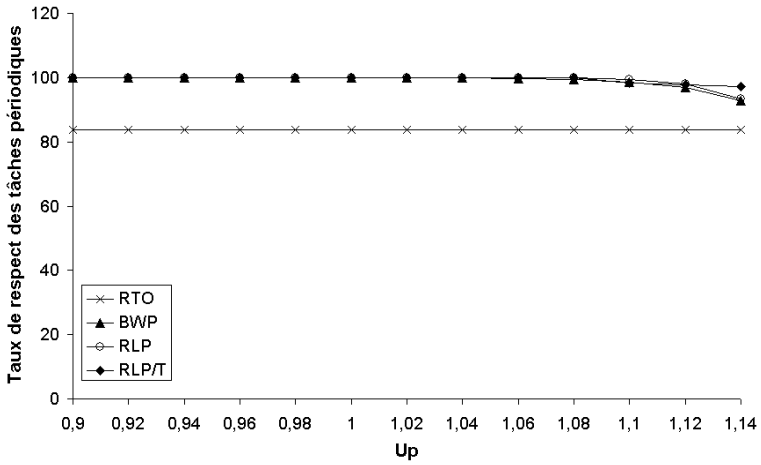

FIG. III.19 – Taux de respect en fonction de U_p ($s_i = 6$, ACET=0.90 WCET)

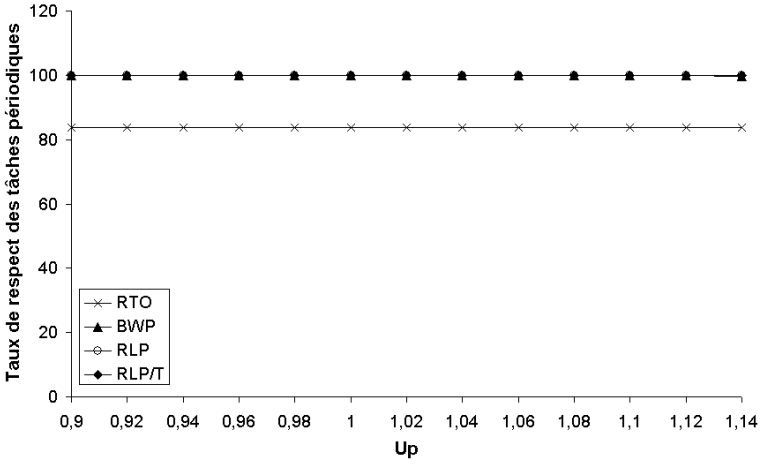

FIG. III.20 – Taux de respect en fonction de U_p ($s_i = 6$, ACET=0.75 WCET)

ceptation s'effectuait sur la base des exécutions réelles des tâches. C'est exactement ce que nous observons sur les figures III.17 et III.18 : pour des charges périodiques égales à 110% et 130% respectivement, RLP/T offre momentanément des performances légèrement inférieures à celles fournies par BWP et RLP. Ce phénomène n'est plus observable dès lors que l'on autorise peu de pertes au niveau des tâches périodiques. En effet, pour $s_i = 6$, les performances de BWP, RLP et BWP sont identiques. Néanmoins pour $s_i = 2$, notons que globalement (toutes charges périodiques confondues), RLP/T est quasiment toujours plus performant vis-à-vis des autres algorithmes d'ordonnancement.

En résumé, nous pouvons dire que l'influence de la durée d'exécution réelle de la tâche sur les performances des algorithmes étudiés est significative. La prévalence de RLP/T sur BWP et RLP est d'autant plus importante que d'une part, les tâches s'accomplissent avec des durées d'exécution proches de leur WCET, et que d'autre part, les pertes autorisées au niveau des tâches sont importantes.

4.4 Evaluation du taux de préemption

Il s'est avéré intéressant de comparer le nombre de préemptions engendrées par les différents algorithmes, de manière à évaluer correctement les surcoûts temporels relatifs à leur implémentation. En effet, les résultats précédents deviendraient inexploitables s'ils étaient obtenus au prix de surcoûts importants lors de changements de contexte trop nombreux. Les simulations présentées ci-après évaluent le taux de préemption engendré par les algorithmes RTO, BWP, RLP et RLP/T en fonction de la charge périodique U_p appliquée au système, pour deux paramètres de tolérance aux pertes : $s_i = 2$ (cf. figure III.21) et $s_i = 6$ (cf. figure III.22).

Au vu des résultats, nous pouvons dire que globalement, les algorithmes Skip-Over classiques, à savoir RTO et BWP, présentent un taux de préemption qui tend à croître en fonction de la charge périodique U_p. Le taux observé sous BWP étant d'autant plus important vis-à-vis de celui affiché sous RTO, que les pertes autorisées sont faibles d'une part, et que la charge appliquée au système est élevée d'autre part.

Concernant l'algorithme RLP, nous constatons qu'il induit le plus grand nombre de préemptions, tous algorithmes confondus, notamment lorsque le système est moyennement chargé. A titre illustratif, pour $s_i = 2$ et $U_p = 130\%$, le taux de préemption moyen engendré par l'algorithme RLP atteint une valeur maximale égale à 0.7%. Au-delà de cette charge périodique, la courbe de RLP subit une décroissance. Celle-ci est due au fait que plus le système est surchargé, moins il y a d'instances bleues, donc de préemptions liées à l'exécution de ce type d'instances.

Enfin, au niveau de l'algorithme RLP/T, nous observons que le taux de préemption semble indépendant de la charge périodique appliquée au système. RLP/T est l'algorithme le plus compétitif en termes de taux de préemption dès lors que le système est faiblement surchargé pour $s_i = 6$ ($U_p \approx 100\%$), ou moyennement surchargé dans le cas de $s_i = 2$ ($U_p > 140\%$). Nous remarquons également que le paramètre de pertes influe sur les résultats de mesures : plus les pertes autorisées au niveau des tâches sont faibles, plus l'écart de performance de RLP/T est important vis-à-vis des trois autres algorithmes étudiés.

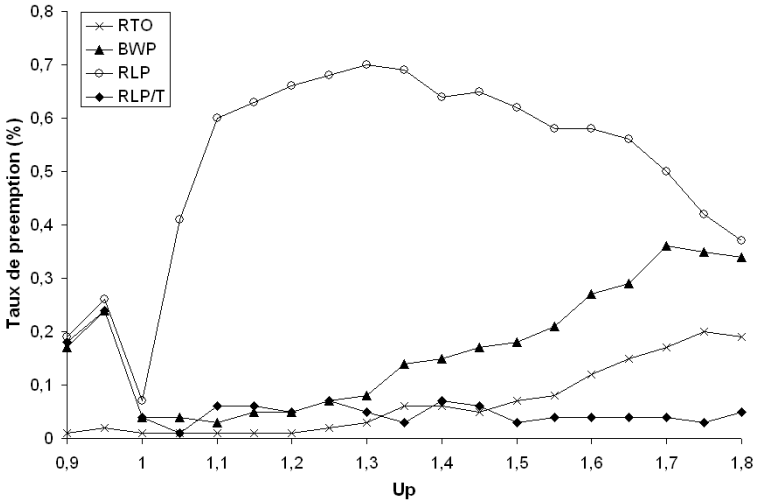

FIG. III.21 – Taux de préemption en fonction de U_p ($s_i = 2$)

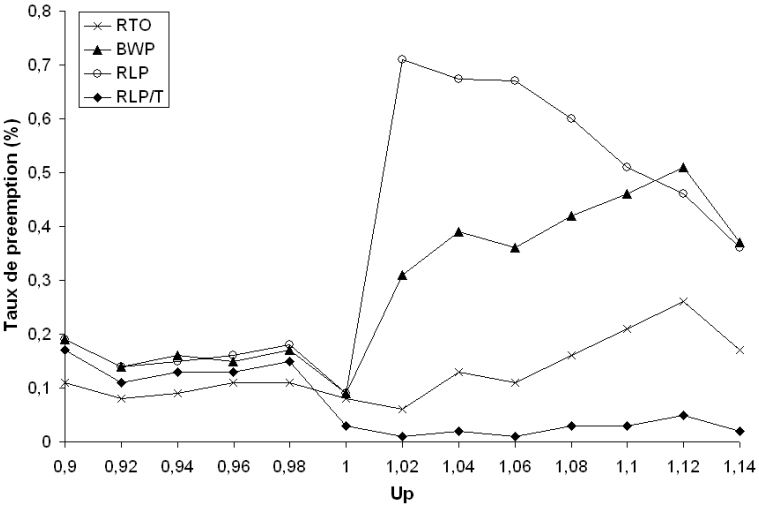

FIG. III.22 – Taux de preemption en fonction de U_p ($s_i = 6$)

4.5 Evaluation du taux de CPU gaspillé

Il est également apparu judicieux d'effectuer des mesures concernant le temps CPU gaspillé dans l'exécution d'instances bleues non terminées. Les résultats de simulation pour $s_i = 2$ et $s_i = 6$ sont représentés sur les figures III.23 et III.24 respectivement.

Le temps CPU gaspillé sous RTO est nul puisque l'algorithme ordonnance uniquement des instances rouges. Pour RLP/T, le temps CPU gaspillé est aussi égal à zéro quelle que soit la charge périodique appliquée au système. Dans ce cas, ce résultat attendu provient de la mise en œuvre du test d'acceptation des instances bleues qui écarte tout abandon de ce type de tâches. Une tâche bleue n'est acceptée que si celle-ci peut s'accomplir dans le respect de son échéance, tout en ne compromettant pas l'exécution des autres instances de tâches présentes dans le système.

Au niveau des algorithmes BWP et RLP, le temps CPU gaspillé est toujours positif dès lors que le système souffre d'une surcharge ($U_p > 1$). Pour $U_p = 115\%$, les algorithmes BWP et RLP induisent la quantité de temps CPU gaspillé la plus importante, à savoir un taux de 24% et 26% respectivement. Les résultats de l'évaluation du temps CPU gaspillé mettent en évidence le fait que RLP passe globalement plus de temps à exécuter des instances bleues en échec que BWP. Sous RLP, le fait d'ordonnancer les instances rouges au plus tard libère davantage de temps CPU utilisé pour tenter l'exécution d'instances bleues que sous BWP. Par conséquent, ceci s'assortit d'une augmentation globale du temps CPU gaspillé dans l'exécution d'instances bleues tentées puis abandonnées. Toutefois, comme nous avons pu le voir dans la section précédente, le nombre global de requêtes bleues réussies sous RLP reste supérieur à celui observé sous BWP.

Par ailleurs, l'allure générale des courbes relatives à BWP et RLP mérite quelques commentaires. Nous pouvons remarquer que le temps CPU gaspillé le plus important est observé lorsque le système est faiblement surchargé ($U_p < 125\%$ pour $s_i = 2$, et $U_p < 106\%$ pour $s_i = 6$). Au-delà de ces charges périodiques, les courbes obtenues pour BWP et RLP subissent une décroissance : plus le système est surchargé, plus il y a d'instances rouges à traiter, donc moins il y a de temps CPU disponible pour l'exécution des instances bleues.

Enfin, la valeur du paramètre de pertes est directement liée au pourcentage de temps CPU gaspillé dans l'exécution d'instances bleues abandonnées. Les résultats de simulation montrent que le temps CPU gaspillé est d'autant plus faible que les pertes autorisées au niveau des tâches sont peu importantes. Ainsi, pour $U_p = 110\%$, le temps CPU gaspillé sous BWP est 4 fois moindre dans le cas où $s_i = 6$ que dans celui où $s_i = 2$.

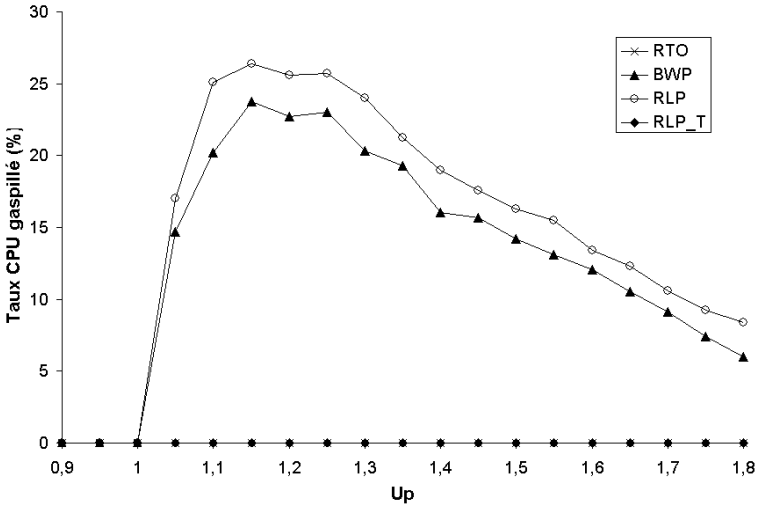

FIG. III.23 – Taux de temps CPU gaspillé en fonction de U_p ($s_i = 2$)

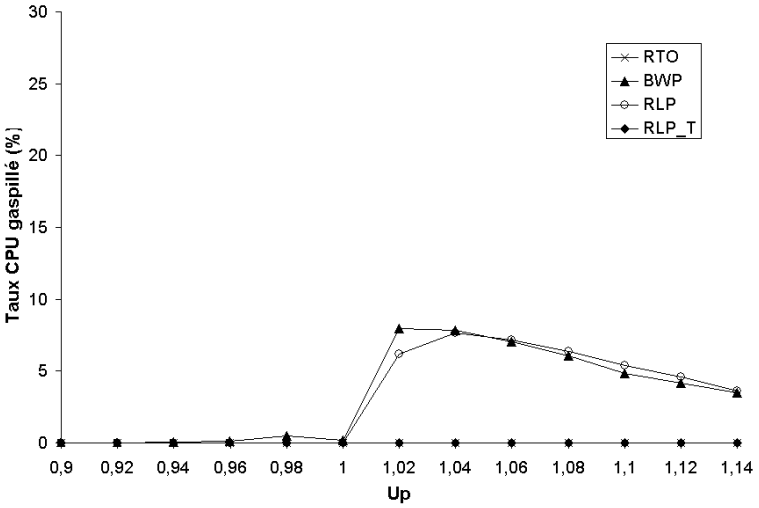

FIG. III.24 – Taux de temps CPU gaspillé en fonction de U_p ($s_i = 6$)

4.6 Evaluation du taux d'oisiveté

Nous avons enfin comparé les performances des différents algorithmes d'ordonnancement sur la base d'un dernier critère : le taux d'oisiveté. Celui-ci représente le pourcentage de temps durant lequel le processeur est inactif, c'est-à-dire durant lequel il n'exécute aucune tâche. Il est intéressant de mesurer cette grandeur qui représente la capacité du système à faire face à des situations d'accroissement temporaire de la charge de traitement liée à l'occurrence d'une tâche apériodique par exemple. Les résultats de simulation pour $s_i = 2$ et $s_i = 6$ sont représentés sur les figures III.25 et III.26 respectivement.

Nous remarquons tout d'abord que le taux d'oisiveté sous RTO est le plus important. Celui-ci décroît linéairement en fonction de la charge périodique U_p appliquée au système, variant dans le cas de $s_i = 2$, de 55% pour $U_p = 90\%$, à 10% pour $U_p = 180\%$. Notons les points singuliers des courbes $s_i = 2$ et $s_i = 6$: lorsque $U_p = 100\%$, les taux d'oisiveté sont respectivement égaux à $\frac{1}{2} = 50\%$ et $\frac{1}{6} = 16.7\%$, ces valeurs correspondant exactement aux taux de pertes autorisées.

En ce qui concerne les algorithmes BWP, RLP et RLP/T, les taux d'oisiveté observés pour ces algorithmes sont identiques lorsque le système n'est pas surchargé ($U_p < 100\%$). Ceux-ci sont positifs ($T_{temps\_creux} = 10\%$ pour $U_p = 90\%$) et décroissent de manière linéaire pour atteindre une valeur nulle pour $U_p = 100\%$.

Cependant, lorsque le système est en surcharge ($U_p > 100\%$), les résultats obtenus par les différents algorithmes diffèrent. RLP n'induit aucun temps oisif quelles que soient les pertes autorisées au niveau du système. BWP présente un faible taux d'oisiveté uniquement pour de faibles pertes autorisées. Quant à RLP/T, cet algorithme paraît le plus intéressant conservant un taux d'oisiveté non négligeable lorsque le système est faiblement surchargé. En effet, le taux d'oisiveté sous RLP/T pour $s_i = 2$ et $U_p = 115\%$ est égal à 9%. De plus, même lorsque le système est fortement surchargé, RLP/T induit un pourcentage de temps oisifs certes faible mais toujours positif.

En conclusion, l'algorithme RLP/T semble le plus apte à répondre à des surcharges temporaires de traitement comme nous pourrons le vérifier dans le chapitre IV.

5 Synthèse

Le but de ces simulations a été de nous permettre d'évaluer les performances des 2 algorithmes proposés dans ce chapitre, à savoir RLP et RLP/T, par rapport à celles des algorithmes RTO et BWP introduits par le passé.

Au vu d'une part des descriptions algorithmiques, et d'autre part des résultats de simulation, nous pouvons dresser le tableau récapitulatif suivant (cf. III.3) résumant les performances des différents ordonnanceurs en termes de performances et de complexités. Dans ce tableau, les '★' représentent les bonnes performances de l'algorithme d'ordonnancement au niveau d'un critère donné.

Ces résultats mettent en avant l'existence d'un compromis performance-complexité. En effet, cette classification nous montre la performance assez médiocre des algorithmes Skip-Over de base (RTO et BWP). Celle-ci s'accompagne d'une facilité de mise en œuvre

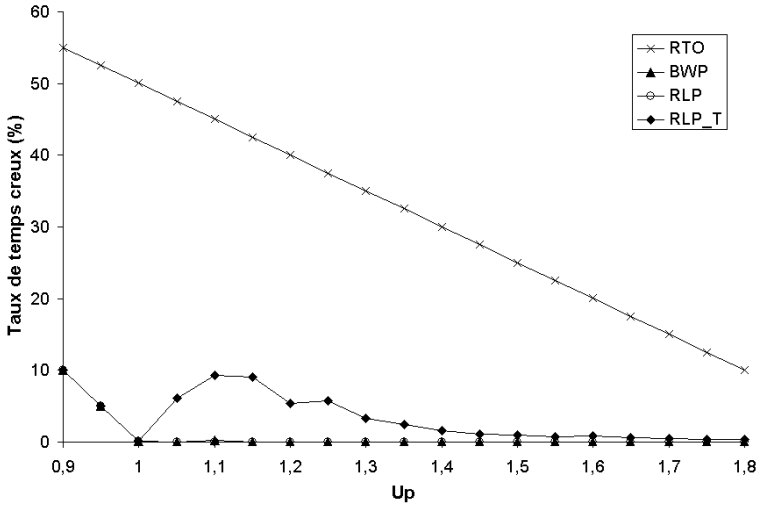

FIG. III.25 – Taux d'oisiveté en fonction de U_p ($s_i = 2$)

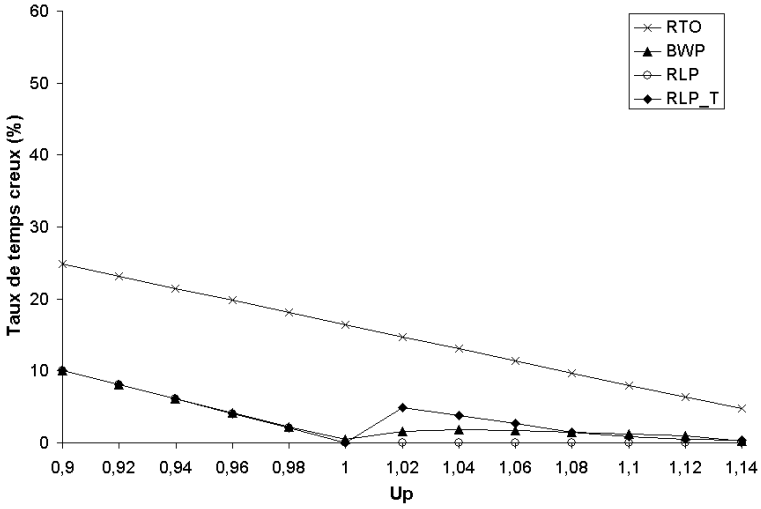

FIG. III.26 – Taux d'oisiveté en fonction de U_p ($s_i = 6$)

Algorithmes Skip-Over	Performance QoS	Complexité algorithmique	Besoins mémoire	Complexité d'implémentation
RTO	★	★★★★	★★★★	★★★★
BWP	★★	★★★	★★★	★★★
RLP	★★★	★★	★	★★
RLP/T	★★★★	★	★	★

TAB. III.3 – Synthèse des performances des ordonnanceurs sous contraintes de QoS

et d'implémentation. En contrepartie, ce tableau souligne la prédominance de performance de l'algorithme RLP/T, qui est cependant acquise au détriment d'une augmentation de la complexité de calcul et d'implémentation.

6 Conclusion

A la fin de ce chapitre, nous disposons de deux nouveaux ordonnanceurs monoprocesseur, à savoir RLP et RLP/T, capables de prendre en compte des configurations de tâches composées de tâches périodiques munies de contraintes de pertes.

Nous avons entrepris des simulations dans le but de comparer les comportements des différents algorithmes Skip-Over que nous présentons en fonction de la charge périodique appliquée au système. Celles-ci ont permis de comparer les performances de ces algorithmes suivant les critères d'efficacité classiques comme le taux d'échéances satisfaites, le taux de préemption, etc. Les simulations permettent de confirmer le très bon comportement de l'algorithme RLP/T au niveau de l'ensemble des critères évalués.

L'objectif du chapitre suivant va consister à étendre cette approche de l'ordonnancement dynamique avec contraintes de QoS à la gestion conjointe de tâches apériodiques.

Chapitre IV

Serveurs de tâches apériodiques sous contraintes de QoS

L'objet de ce chapitre est d'étudier l'ordonnancement d'un ensemble hybride de tâches constitué de tâches périodiques définies sous contraintes de QoS et de tâches apériodiques. Nous considérons les cas distincts relatifs à l'occurrence de requêtes apériodiques critiques et non critiques. Dans les deux premières parties nous décrivons successivement l'utilisation des serveurs de tâches apériodiques BG, TBS, TB et EDL avec les différents ordonnanceurs Skip-Over basés sur EDF, à savoir RTO et BWP. Ensuite, notre contribution se focalise sur l'utilisation du serveur optimal EDL avec des tâches périodiques ordonnancées selon les stratégies introduites dans le chapitre précédent, à savoir RLP et RLP/T. La dernière section du chapitre présente l'analyse des performances des différents modèles d'ordonnancement envisagés.*

Les travaux présentés ici visent à exploiter les pertes autorisées au niveau des tâches périodiques de manière à servir au mieux les tâches apériodiques non critiques (minimisation de leur temps de réponse) et les tâches apériodiques critiques (maximisation de leur taux d'acceptation). Le but est également de minimiser la dégradation de QoS (maximisation du nombre d'instances de tâches réussies) observée au niveau des tâches périodiques, causée par l'occurrence des requêtes apériodiques.

1 Ordonnancement des tâches apériodiques sous RTO

Considérons que les tâches périodiques avec contraintes de QoS sont ordonnancées selon l'algorithme RTO. La prise en compte des différents types d'instances de tâches (rouges, bleues, apériodiques critiques et apériodiques non critiques) peut alors être représentée par le schéma décrit sur la figure IV.1.

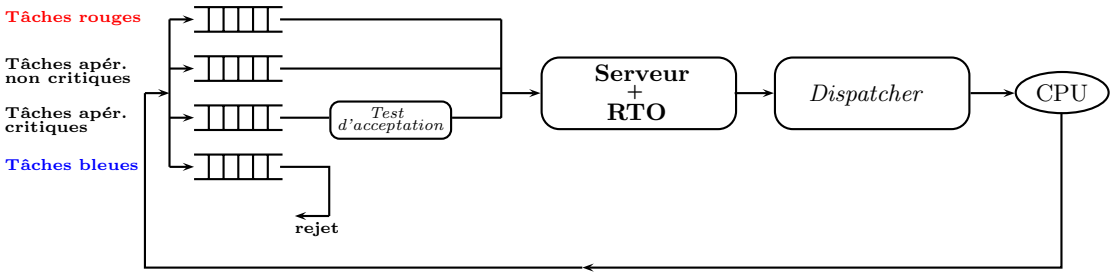

FIG. IV.1 – Schéma d'ordonnancement des tâches apériodiques sous RTO

Les instances de tâches occurrentes sont indiquées sur la partie gauche du schéma où apparaissent également les listes des tâches en attente ; chaque type d'instance étant stockée dans une liste de tâches différente, dans l'attente de son traitement par le *modèle d'ordonnancement* dénommé ici "Serveur + RTO". Par *modèle d'ordonnancement*, on entend l'association combinée d'un serveur de tâches apériodiques et d'un ordonnanceur de tâches périodiques, assurant l'ordonnancement conjoint des différents types de tâches occurrentes. Le *dispatcher* choisit ensuite la tâche placée en tête de liste par le modèle d'ordonnancement pour l'exécuter sur le processeur. Dans la suite, nous nous attachons à spécifier le comportement du modèle d'ordonnancement selon la présence ou non de tâches apériodiques critiques ou non critiques.

– **Cas n°1 : pas de tâches apériodiques critiques ou non critiques**

  Cette situation correspond exactement à la définition du comportement de l'ordonnanceur RTO. Les tâches rouges occurrentes entrent directement dans le système tandis que les tâches bleues occurrentes sont systématiquement rejetées. Les tâches rouges prêtes sont ensuite exécutées au plus tôt selon l'algorithme EDF.

– **Cas n°2 : présence de tâches apériodiques non critiques uniquement**

  Dans le cas où des tâches apériodiques non critiques surviennent, celles-ci sont acceptées sans condition aucune au sein du système et sont exécutées conjointement avec les tâches rouges selon la politique du serveur de tâches apériodiques sélectionné. En présence d'au moins une tâche apériodique non critique prête et non entièrement exécutée, toute instance bleue occurrente est rejetée.

– **Cas n°3 : présence de tâches apériodiques critiques uniquement**

  Les tâches apériodiques critiques entrent dans le système sur test d'acceptation. Leur acceptation ne doit pas remettre en cause le respect des échéances des tâches rouges présentes dans le système. Le test d'acceptation mis en œuvre dépend du serveur de tâches apériodiques utilisé. Une fois acceptée, toute tâche apériodique

critique est ordonnancée conjointement avec les tâches rouges et les tâches apériodiques critiques préalablement acceptées, selon l'algorithme EDF.

– **Cas n°4 : présence de tâches apériodiques critiques et non critiques**

Là encore, parmi les tâches périodiques, seules les instances de tâches rouges sont ordonnancées, les instances de tâches bleues étant systématiquement écartées en présence d'au moins une tâche apériodique non critique non terminée. Les tâches apériodiques non critiques entrent directement dans le système tandis que les tâches apériodiques critiques subissent un test d'acceptation pour intégrer le système. Ensuite, l'ordonnancement conjoint des tâches rouges, des tâches apériodiques critiques et non critiques est lié au serveur de tâches apériodiques mis en œuvre.

Dans ce qui suit, nous détaillons le fonctionnement des différents modèles d'ordonnancement étudiés, à savoir BG-RTO, TBS-RTO, TB*-RTO et EDL-RTO. Pour chaque modèle, nous décrivons successivement la gestion des tâches apériodiques non critiques puis celle des tâches apériodiques critiques.

1.1 Le modèle d'ordonnancement BG-RTO

1.1.1 Gestion des tâches apériodiques non critiques

1.1.1.1 Description. Le serveur de tâches apériodiques Background (BG) ordonnance les tâches apériodiques lorsqu'il n'y aucune activité périodique (c'est-à-dire, lorsqu'il n'y a aucune tâche périodique à l'état prêt) au sein du système. Dans le cas de tâches périodiques basées sur le modèle RTO, le serveur BG ordonnance les tâches apériodiques lorsqu'il n'y a aucune tâche *rouge* active dans le système.

1.1.1.2 Illustration. Considérons un ensemble $\mathcal{T}$ constitué de deux tâches périodiques $T_1(6, 10, 2)$ et $T_2(3, 6, 2)$. Ce système n'est pas ordonnançable avec un ensemble de tâches périodiques définies selon le modèle classique, car le facteur d'utilisation du processeur, $\sum_{i=1}^{n} \frac{c_i}{p_i} = \frac{6}{10} + \frac{3}{6}$ dans ce cas, est strictement supérieur à 1. Le facteur d'utilisation des tâches en prenant en compte les pertes est quant à lui égal à $U_p^* = 90\%$ (cf. Chap II. Définition 9), soit un facteur d'utilisation du processeur inférieur à 1. On suppose à présent qu'une requête apériodique de durée 10 unités de temps survient au temps $\tau = 20$. Nous avons vu précédemment que les temps creux libérés grâce aux pertes introduites au niveau des tâches périodiques, pouvaient être exploitées pour exécuter des tâches supplémentaires et plus particulièrement pour ordonnancer des requêtes apériodiques. Les tâches dans cet exemple (cf. figure IV.2) sont définies selon le modèle RTO et sont toujours ordonnancées au plus tôt selon l'agorithme EDF. La requête apériodique est exécutée dès lors qu'il n'y a plus de tâches rouges à l'état prêt. Le temps de réponse observé pour la tâche apériodique avec le modèle d'ordonnancement BG-RTO est alors de 28 unités de temps.

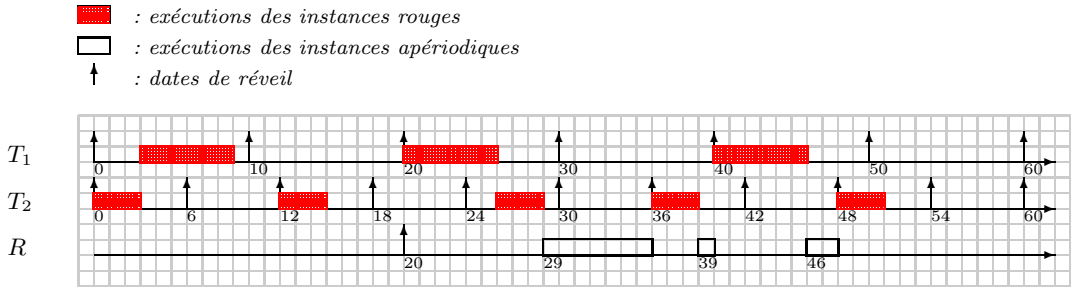

FIG. IV.2 – Gestion des apériodiques non critiques sous BG-RTO

1.1.2 Gestion des tâches apériodiques critiques

L'acceptation des tâches apériodiques sous BG-RTO nécessite la construction préliminaire de la fonction de disponibilité f^{EDS} appliquée à un modèle de tâches RTO. Celle-ci représente l'ensemble des temps creux disponibles lorsque les tâches périodiques rouges sont exécutées au plus tôt. La séquence EDS-RTO doit en effet être déterminée pour pouvoir calculer la somme des temps creux entre l'instant d'occurrence d'une tâche apériodique critique et son échéance, de manière à valider ou non l'acceptation de cette nouvelle tâche au sein du système. Les équations de calcul de la séquence EDS (Earliest Deadline as Soon as possible) présentées ci-après pour le modèle de tâches RTO, sont dérivées de celles établies par Chetto et Chetto pour un modèle de tâches sans pertes [CC90].

1.1.2.1 Calcul des temps creux selon EDS sous le modèle RTO

Calcul du vecteur statique des dates de réveil. Soit l'hyperpériode $P' = ppcm(s_1p_1,\ s_2p_2, ..., s_np_n)$ définie sur un ensemble de tâches avec pertes. Le vecteur statique des dates de réveil, noté $\mathcal{E} = (e_0, e_1, ..., e_i, e_{i+1}, ..., e_q)$, représente l'ensemble des instants considérés sur l'hyperpériode $[0, P']$, qui marque la fin d'un temps creux. Il se construit à partir des dates de réveil distinctes des instances périodiques *rouges* et *bleues* confondues. Par conséquent, les instants e_i du vecteur $\mathcal{E}$ sont définis par le théorème suivant :

Théorème 34 *Les instants e_i du vecteur $\mathcal{E} = (e_0, e_1, ..., e_i, e_{i+1}, ..., e_q)$ calculés sur $0, P'[$ pour des tâches RTO sont définis comme suit :*

$$e_i = x.p_j \tag{IV.1}$$

avec $x = 1, ..., \frac{P}{p_j}$, $e_i < e_{i+1}$, $e_0 = \min\{p_j; 1 \leq j \leq n\}$ et $e_q = P$

Calcul du vecteur statique des temps creux. Le vecteur statique des temps creux $\mathcal{D} = (\Delta_0, \Delta_1, ..., \Delta_i, \Delta_{i+1}, ..., \Delta_q)$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des dates de réveil. Δ_i représente la longueur

du temps creux se terminant au temps k_i. Le théorème suivant fournit la formule de récurrence pour le calcul du vecteur $\mathcal{D}$ sous RTO, tenant compte des pertes s_j autorisées par chacune des tâches T_j :

Théorème 35 *Les durées des temps creux du vecteur* $\mathcal{D} = (\Delta_0, \Delta_1, ..., \Delta_i, \Delta_{i+1}, ..., \Delta_q)$ *calculées sur* $[0, P'[$ *pour des tâches RTO sont définies comme suit :*

$$\Delta_i = \sup(0, F_i) \quad \textit{pour } i = 0 \textit{ à } q \tag{IV.2}$$

$$\textit{avec} \quad F_i = e_i - \sum_{j=1}^{n} (\lceil \frac{e_i}{p_j} \rceil - \sup(0, \lceil \frac{e_i - (s_j - 1)p_j}{p_j s_j} \rceil))c_j - \sum_{k=0}^{i-1} \Delta_k$$

Preuve :

Pour une tâche T_j, la durée d'une séquence RTO est égale à une pseudo-période $s_j p_j$. Par définition du modèle RTO, chaque tâche T_j présente sur chaque pseudo-période, $(s_j - 1)$ requêtes d'exécution rouges suivies d'*une* requête d'exécution bleue. La tâche T_j présente donc une seule perte tous les $s_j p_j$ pseudo-périodes. Par ailleurs, l'instance bleue de la pseudo-période $s_j p_j$ contenant l'instant e_i, ne doit être comptabilisée comme perte seulement dans le cas où l'instant e_i considéré, est strictement supérieur à la date de réveil de cette instance bleue, soit strictement supérieur à l'instant $(s_j - 1)p_j$ relatif au début de la pseudo-période $s_j p_j$. Ainsi, le nombre total de pertes observées sur l'intervalle $[0, e_i]$ s'élève à $\lceil \frac{e_i - (s_j-1)p_j}{p_j s_j} \rceil$. Dans le cas où l'instant e_i apparaît au cours de la première pseudo-période, le calcul $(e_i - (s_j - 1)p_j)$, pouvant induire induire un résultat négatif, on considérera dans le cas général la formulation $\sup(0, \lceil \frac{e_i - (s_j-1)p_j}{p_j s_j} \rceil)$. □

Illustration du calcul de $f^{EDS-RTO}$. Considérons l'ensemble de tâches $\mathcal{T} = \{T_1, T_2\}$ constitué de deux tâches périodiques RTO, $T_1(6, 10, 2)$ et $T_2(3, 6, 2)$ avec $s_1 = s_2 = 2$. L'application des équations décrites dans les théorèmes 34 et 35 nous fournit la localisation et la durée de l'ensemble des temps creux résultant de l'ordonnancement de l'ensemble $\mathcal{T}$ par EDS. Les valeurs obtenues sont résumées dans la Table IV.1.

e_i	0	6	10	12	18	20	24	30	36	40	42	48	50	54	60
Δ_i	0	0	1	2	3	2	0	1	6	1	0	2	0	3	6
F_i	0	-3	1	2	3	2	-2	1	6	1	-4	2	-1	3	6
$\sum_k \Delta_k$	0	0	1	3	6	8	8	9	15	16	16	18	18	21	27

TAB. IV.1 – Dates et durées des temps creux calculées par l'algorithme EDL

Le calcul des intervalles de temps creux fournit la séquence EDS-RTO représentée sur la figure IV.3.

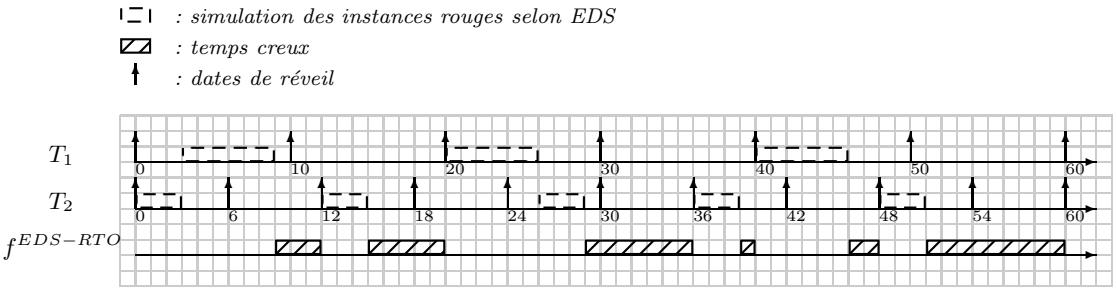

FIG. IV.3 – Calcul des temps creux selon EDS sous RTO

1.1.2.2 Test d'acceptation des apériodiques critiques sous BG-RTO. Le test d'acceptation dans le cas de tâches avec pertes sous RTO s'appuie sur celui énoncé précédemment (cf. Chapitre I Théorème10) dans le cas de tâches sans pertes.

Théorème 36 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDS-RTO}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{IV.3}$$

Dans ce cas, seule la somme des temps creux $\Omega^{EDS-RTO}_{\mathcal{T}(\tau)}(\tau, d_i)$ calculée entre τ et d_i sur la base d'une séquence EDS-RTO, diffère vis-à-vis du test d'acceptation sous un modèle de tâches classique sans pertes.

Notons qu'une fois acceptées, les tâches apériodiques critiques sont ordonnancées au plus tôt selon l'algorithme EDF, conjointement avec les tâches périodiques rouges.

1.2 Le modèle d'ordonnancement TBS-RTO

1.2.1 Gestion des tâches apériodiques non critiques

1.2.1.1 Description. La formule utilisée par le serveur TBS pour assigner une échéance fictive aux requêtes apériodiques occurrentes dans le cas de tâches Skip-Over est identique à celle utilisée dans le cas de tâches sans pertes. La différence réside dans le valeur du paramètre U_s assignée au serveur. Aussi, lorsque la k-ième requête apériodique arrive au temps $t = r_k$, elle reçoit une échéance :

$$d_k = \max(r_k, d_{k-1}) + \frac{C^a_k}{U_s} \tag{IV.4}$$

où C^a_k est la durée d'exécution de la requête et U_s correspond au facteur d'utilisation du serveur (c'est-à-dire sa largeur de bande). Caccamo et Buttazzo [CB97] ont montré que la largeur de bande CPU induite par les sauts d'instances bleues peut être exploitée par l'algorithme TBS pour avancer l'exécution des tâches apériodiques. Il convient alors

d'adapter la valeur du paramètre U_s du serveur pour exploiter au maximum les pertes autorisées au niveau des tâches périodiques. Les auteurs ont donc mené une analyse hors-ligne du fonctionnement de TBS avec des tâches périodiques définies au sens Skip-Over, qui les a conduits à dériver de nouvelles bornes d'ordonnançabilité. Celles-ci sont énoncées et justifiées ci-après.

1.2.1.2 Conditions d'ordonnançabilité. Le théorème suivant [CB97] fournit une condition *suffisante* garantissant l'ordonnançabilité d'un ensemble hybride de tâches.

Théorème 37 *Etant donnés un ensemble de n tâches périodiques autorisant des pertes et ayant un facteur d'utilisation équivalent du processeur U_p^*, et un ensemble de tâches apériodiques servies par un serveur TBS ayant un facteur d'utilisation du processeur U_s, le système entier est ordonnançable par RTO ou BWP si*

$$U_p^* + U_s \leq 1. \tag{IV.5}$$

Cette condition entraîne une largeur de bande CPU garantie, notée $U_{s_{min}} = 1 - U_p^*$, pour les tâches apériodiques. Celle-ci découle de la distribution *granulaire* des temps creux produits par les pertes affichées par les tâches périodiques. En fait, une fraction de ces temps creux est uniformément distribué tout au long de la séquence d'ordonnancement des périodiques et peut être utilisée en tant que largeur de bande CPU additionnelle ($U_{skip} = U_p - U_p^*$) disponible pour le service des tâches apériodiques. Le reste des temps creux induits par les pertes est discontinu, et crée des "trous" dans la séquence d'ordonnancement, qui ne peuvent pas être utilisés à tout moment malheureusement. Lorsqu'une requête apériodique survient dans un de ces trous, elle peut exploiter une largeur de bande CPU supérieure à $1 - U_p^*$. En effet, il est facile d'extraire des exemples pour lesquels l'ensemble des tâches périodiques est ordonnançable en assignant au serveur TBS, une largeur de bande U_s supérieure à $1 - U_p^*$. Le théorème 38 [CB97] fournit une largeur de bande maximale $U_{s_{max}}$ au-delà de laquelle la séquence n'est cependant plus ordonnançable.

Théorème 38 *Etant donnés un ensemble de n tâches périodiques autorisant des pertes et ayant un facteur d'utilisation équivalent du processeur U_p^*, et un ensemble de tâches apériodiques servies par un serveur TBS ayant un facteur d'utilisation du processeur U_s, une condition nécessaire à l'ordonnançabilité du système entier est*

$$U_s \leq U_{s_{max}} \tag{IV.6}$$

$$où \qquad U_{s_{max}} = 1 - U_p + \sum_{i=1}^{n} \frac{c_i}{p_i s_i}.$$

Notons que $U_{s_{max}}$ représente le cas limite dans lequel l'ensemble de la largeur de bande induite par les pertes peut être utilisé pour servir les tâches apériodiques. Une grande différence entre $U_{s_{min}}$ et $U_{s_{max}}$ indique que les temps creux induits par les pertes ne sont pas uniformément distribués, et que seulement une largeur de bande égale à $U_{s_{min}}$

peut être garantie. Dans ce cas, la largeur de bande résiduelle égale à $U_{s_{max}} - U_{s_{min}}$ ne peut pas être assignée au serveur TBS mais peut être exploitée par l'algorithme BWP pour exécuter des instances de tâches bleues lorsque cela est possible.

Sur la base de ces résultats, nous illustrons à présent sur un exemple, la gestion sous TBS-RTO des tâches apériodiques non critiques.

1.2.1.3 Illustration. Considérons l'ensemble précédent $\mathcal{T}$ constitué de deux tâches périodiques $T_1(6, 10, 2)$ et $T_2(3, 6, 2)$. Le facteur d'utilisation des tâches en tenant compte des pertes est égal à $U_p^* = 90\%$ (cf. Chap II. Définition 9). La figure IV.4 fournit un exemple d'ordonnancement produit par le serveur TBS avec $U_s = 1 - U_p^* = 10\%$ (seule la largeur de bande garantie est allouée au serveur). La condition $U_p^* + U_s \leq 1$ est vérifiée, le système considéré est donc ordonnançable. Une requête apériodique de durée d'exécution au pire cas de 10 unités de temps, survient à l'instant $t = 20$. Elle reçoit alors une échéance fictive $d = \max(5, 0) + \frac{10}{0.10} = 20 + 100 = 120$. Les requêtes dont les échéances sont plus proches que d sont ordonnancées en priorité. Par conséquent, nous pouvons observer un temps de réponse à la requête apériodique de 28 unités de temps.

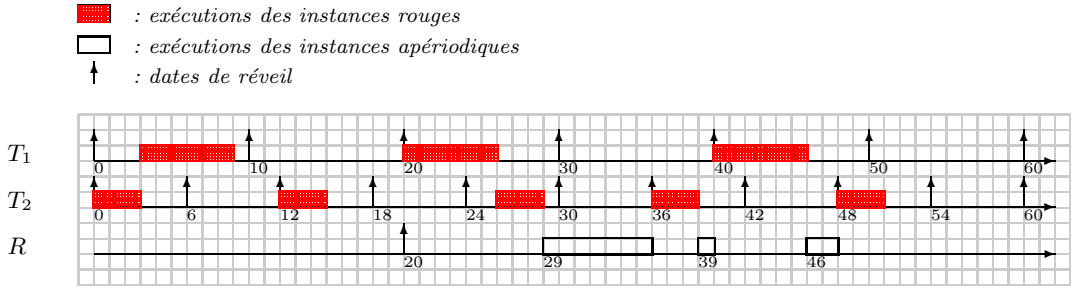

FIG. IV.4 – Gestion des apériodiques non critiques sous TBS-RTO

1.2.2 Gestion des tâches apériodiques critiques

Dans le cas d'une tâche apériodique critique possédant une échéance d_k, celle-ci peut intégrer le système uniquement sous la condition où elle pourra être ordonnancée dans le respect de son échéance, tout en n'entraînant pas de violations d'échéances parmi les instances de tâches rouges. A l'origine, le serveur TBS n'a pas été conçu pour gérer des tâches apériodiques critiques. Cependant, dans le cas où les tâches apériodiques critiques sont servies selon la technique FIFO (First In First Out), le test d'acceptation de ce type de tâches sous TBS est énoncé dans le théorème suivant [BS99] :

Théorème 39 *L'ordonnançabilité d'une requête apériodique critique R_k est garantie si et seulement si l'échéance fictive notée d, assignée par le serveur TBS est inférieure ou égale à l'échéance d_k de la requête.*

Le cas où les requêtes apériodiques sont ordonnancées selon une politique préemptive (selon leur échéance par exemple) peut être traité selon la méthode décrite dans [SBS95]. En particulier, à chaque fois qu'une requête apériodique est préemptée par une autre requête, la requête courante est séparée en deux parties : la partie qui s'est exécutée et celle qu'il reste à exécuter. La première partie est traitée comme une tâche apériodique se terminant plus tôt que prévu (durée effective inférieure au WCET). La seconde partie est laissée dans la file des tâches apériodiques et traitée comme une nouvelle requête apériodique avec une durée d'exécution réduite.

Dans le cadre des travaux de thèse, nous avons supposé que les tâches apériodiques critiques sous TBS étaient ordonnancées selon la technique FIFO. Ainsi, l'acceptation des requêtes apériodiques critiques repose entièrement sur le test décrit dans le théorème 39. Les requêtes apériodiques critiques acceptées sont ensuite ordonnancées selon EDF de façon conjointe avec les tâches rouges.

1.3 Le modèle d'ordonnancement TB*-RTO

1.3.1 Gestion des tâches apériodiques non critiques

1.3.1.1 Description. Dans la section précédente, nous avons souligné le fait que les pertes autorisées au niveau des tâches périodiques peuvent créer des "trous" dans la séquence d'ordonnancement qui ne peuvent pas être exploités en tant que largeur de bande continue. Par conséquent, le serveur TBS ne peut pas tirer avantage de ces trous pour avancer les échéances assignées aux tâches apériodiques. Cependant, dans le cas du serveur TB*, Caccamo et Buttazzo ont montré dans [CB98] que de tels trous peuvent être exploités dans le but d'améliorer le temps de réponse des requêtes apériodiques. Dans le chapitre I (cf. Section 3.3.3), nous avons vu que l'échéance assignée à la requête apériodique R_k sous TB*, s'obtenait de manière récurrente via la formule donnée par :

$$d_k^{s+1} = t + C_k^a + I_p(t, d_k^s) \tag{IV.7}$$

où t est le temps courant (correspondant à la date de réveil r_k de la requête R_k ou bien à la date de terminaison de la requête précédente), C_k^a est la durée d'exécution au pire cas requise par R_k, et $I_p(t, d_k^s)$ est l'interférence due aux instances périodiques dans l'intervalle $[t, d_k)$.

Par ailleurs, rappelons que l'interférence périodique $I_p(t, d_k^s)$ s'exprime comme la somme de deux termes, $I_a(t, d_k^s)$ et $I_f(t, d_k^s)$, où $I_a(t, d_k^s)$ est l'interférence due aux instances périodiques actives au temps courant ayant des échéances strictement inférieures à d_k^s, et $I_f(t, d_k^s)$ est l'interférence future due aux instances périodiques activées après le temps courant t et ayant leur échéance avant d_k^s.

En présence de pertes, les deux termes $I_a(t, d_k^s)$ et $I_f(t, d_k^s)$ prennent le sens suivant : $I_a(t, d_k^s)$ devient l'interférence due aux instances périodiques rouges actives au temps courant ayant des échéances strictement inférieures à d_k^s, et $I_f(t, d_k^s)$ devient l'interférence future due aux instances périodiques rouges activées après le temps courant t et ayant

leur échéance avant d_k^s. Par conséquent, nous obtenons de nouvelles égalités adaptées au modèle de tâches Skip-Over qui s'expriment de la manière suivante [CB98] :

$$I_a(t, d_k^s) = \sum_{\tau_i \, active,\, red,\, d_i < d_k^s} c_i(t) \tag{IV.8}$$

$$I_f(t, d_k^s) = \sum_{i=1}^{n} \max\left(0, \lceil \frac{d_k^s - next\_ri(t)}{T_i} \rceil - blue_i(t, d_k^s) - 1\right) C_i \tag{IV.9}$$

où $next\_ri(t)$ représente l'instant supérieur ou égal à t auquel la prochaine instance (rouge ou bleue) de la tâche périodique τ_i sera activée, et $blue_i(t, d_k^s)$ représente le nombre d'instances bleues de τ_i dans l'intervalle $[t, d_k^s)$ (c'est-à-dire, les instances bleues activées après ou au temps t avec une échéance avant d_k^s).

Sur la base de ces nouvelles équations, nous illustrons à présent sur un exemple, la gestion sous TB*-RTO des tâches apériodiques non critiques.

1.3.1.2 Illustration. Considérons toujours l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$ utilisé précédemment pour illustrer les modèles d'ordonnancement BG-RTO et TBS-RTO. Le facteur équivalent d'utilisation du processeur est égal à $U_p^* = 90\%$. La largeur de bande allouée au serveur TB* est telle que $U_s = 1 - U_p^* = 10\%$, de manière à garantir l'ordonnançabilité de l'ensemble hybride de tâches. Une requête apériodique de durée d'exécution de 10 unités de temps, survient à l'instant $t = 20$. Elle reçoit alors une échéance $d_k^0 = \max(20, 0) + \frac{10}{0.1} = 20 + 100 = 120$ selon l'algorithme TBS. En appliquant les nouvelles équations pour le calcul de $I_a(t, d_k^s)$ et $I_f(t, d_k^s)$, nous obtenons les échéances raccourcies rassemblées dans la Table IV.2. L'ordonnancement résultant de l'assignation de la plus petite échéance possible $d_k^* = d_k^6 = 39$ à la requête apériodique est illustré sur la figure IV.5.

Etape	0	1	2	3	4	5	6
d_k^s	120	84	63	51	48	42	39

TAB. IV.2 – Dates d'échéances calculées par l'algorithme TB* sous RTO

Le temps de réponse optimal observé pour la requête R est alors de 19 unités de temps.

1.3.2 Gestion des tâches apériodiques critiques

A l'image du serveur TBS, nous supposons que les tâches apériodiques critiques sous TB* sont ordonnancées selon la technique FIFO (First In First Out). Le test d'acceptation de ce type de tâches sous TB* repose alors sur la condition *nécessaire et suffisante* énoncée dans le théorème suivant [BS99] :

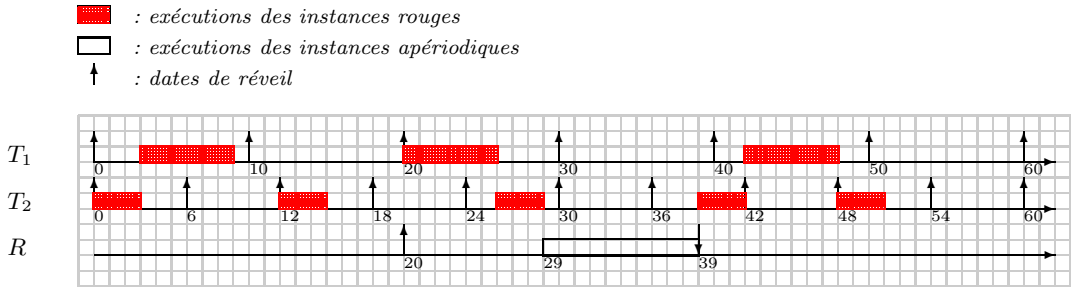

FIG. IV.5 – Gestion des apériodiques non critiques sous TB*-RTO

Théorème 40 *L'ordonnançabilité d'une requête apériodique critique R_k est garantie si et seulement si la plus petite échéance fictive notée d*, assignée par le serveur TB* est inférieure ou égale à l'échéance d_k de la requête.*

1.4 Le modèle d'ordonnancement EDL-RTO

1.4.1 Gestion des tâches apériodiques non critiques

1.4.1.1 Description. Dans le cas d'un modèle sans pertes, les temps creux d'un ordonnanceur EDL sont utilisés pour ordonnancer les tâches apériodiques au plus tôt, en reportant l'exécution des tâches périodiques au plus tard (cf. Chapitre I Section 3.3.1). Sous un modèle à contraintes de QoS, les pertes autorisées au niveau des tâches périodiques créent des temps creux additionnels dans la séquence d'ordonnancement. L'idée est alors d'exploiter au maximum cette quantité de temps disponible pour avancer l'exécution des tâches apériodiques. Le fonctionnement du serveur EDL sous un modèle de tâches Skip-Over consiste donc à exécuter les tâches apériodiques au plus tôt dans les temps creux calculés en exécutant les tâches *rouges* au plus tard. L'application de l'algorithme EDL à des tâches RTO a été détaillée dans le chapitre précédent (cf. Chapitre III Section 2.1). La détermination des temps creux repose sur le calcul exact de la plus grande quantité possible de temps creux sur un intervalle de temps donné. Dans le cas présent, l'intervalle utile est compris entre l'instant d'arrivée de la requête apériodique et sa fin d'exécution non connue *a priori*.

L'algorithme du serveur EDL utilisé pour le calcul des temps creux pour la gestion des tâches apériodiques non critiques sous un modèle de tâches périodiques RTO est décrit ci-dessous :

Algorithme 5
EDL-RTO Algorithm ;
début
calculer $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$ *et* $\mathcal{D}^* = (\Delta_0^*, \Delta_1^*, ..., \Delta_i^*, \Delta_{i+1}^*, ..., \Delta_q^*)$
soit $\mathcal{R}(\tau)$ *= ensemble des tâches apériodiques actives à l'instant* τ
tantque *((une tâche apériodique survient à l'instant* τ*) et (*$\mathcal{R}(\tau) = \varnothing$*))* **faire**
déterminer la plus grande échéance M parmi les tâches périodiques actives

soit $k_i(\tau) = k_q$
soit $j = 0$
tantque *(*$k_i(\tau) \geq \tau$*)* **faire**
si *(*$k_i(\tau) > M$*)*
$\Delta_i^*(\tau) = \Delta_i^*$
sinon
$\Delta_i^*(\tau) = \sup(0, F_i(\tau))$
finsi
ajouter $k_i(\tau)$ *dans* $\mathcal{K}(\tau)$ *et* $\Delta_i^*(\tau)$ *dans* $\mathcal{D}^*(\tau)$
$j = j + 1$
$k_i(\tau) = k_{q-j}$
fintantque
return $(\mathcal{K}(\tau), \mathcal{D}^*(\tau))$
fintantque
fin

Remarquons que le calcul des temps-creux en-ligne, correspondant à l'occurrence d'une tâche apériodique, est uniquement nécessaire dans le cas où il n'y avait pas de tâches apériodiques auparavant dans le système, comme l'a montré Silly dans [Sil99]. Illustrons à présent le fonctionnement du modèle d'ordonnancement EDL-RTO sur un exemple.

1.4.1.2 Illustration. Considérons de nouveau l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$. Cet ensemble de tâches périodiques avec contraintes de QoS est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 20$. On suppose qu'une requête apériodique de durée 10 unités de temps survient au temps $\tau = 20$, impliquant un calcul en-ligne des temps creux. A partir des formules de calcul adaptées à un modèle de tâches RTO, nous pouvons extraire le vecteur dynamique des échéances $\mathcal{K}(\tau) = (20, 30, 42, 50, 54)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (1, 9, 2, 1, 6)$, comme l'illustre la figure IV.6

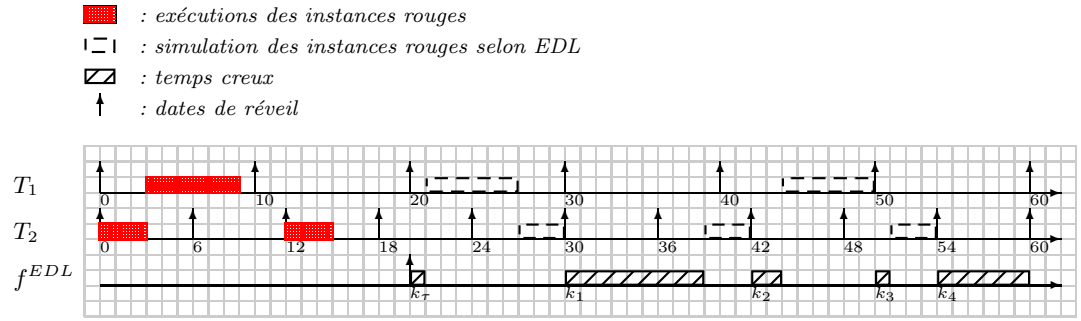

FIG. IV.6 – Calcul des temps creux en dynamique à l'instant $\tau = 20$

L'exécution au plus tôt de la requête apériodique occurrente dans les temps creux de la séquence EDL établie au temps $\tau = 20$ est représentée sur la figure IV.7. Nous observons que la requête reçoit bien 10 unités de temps (1 unité de temps dans le premier intervalle débutant à $\tau = 20$ et 9 unités de temps dans le second débutant à $t = 30$). Son temps de réponse (optimal) est ici de 19 unités de temps. Ce temps est identique à celui observé avec le serveur TB* car les deux serveurs EDL et TB* sont tous les deux optimaux du point de vue de la minimisation du temps de réponse des requêtes apériodiques.

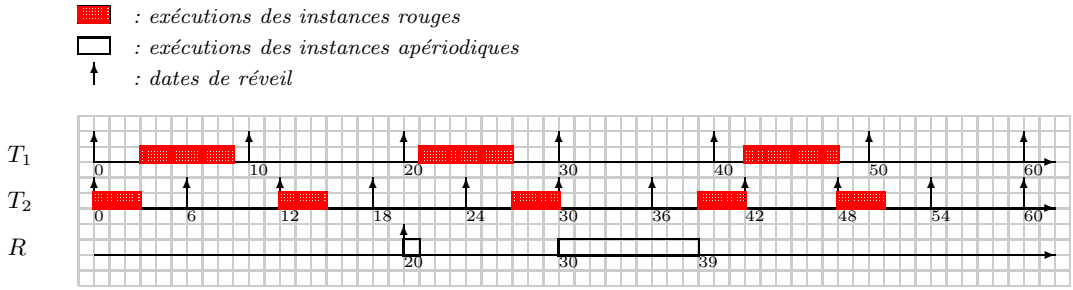

FIG. IV.7 – Gestion des apériodiques non critiques sous EDL-RTO

1.4.2 Gestion des tâches apériodiques critiques

Considérons à présent le problème de l'acceptation de tâches apériodiques à contraintes strictes. On suppose que le système peut présenter plusieurs tâches apériodiques critiques à l'état prêt au temps τ correspondant aux différentes requêtes précédemment acceptées. Par conséquent, celles-ci n'ont pas terminé leur exécution à τ. On note $\mathcal{R}(\tau) = \{R_i(c_i(\tau), d_i)$, i=1 à $q(\tau)\}$ l'ensemble des q tâches apériodiques critiques présentes dans le système à τ. $c_i(\tau)$ désigne la durée d'exécution dynamique de R_i (c'est-à-dire sa durée d'exécution restante) et d_i son échéance. L'ensemble $\mathcal{R}(\tau)$ est ordonné tel que $i < j$ implique $d_i \leq d_j$.

Le test d'acceptation dans le cas de tâches avec pertes sous RTO s'appuie sur celui énoncé précédemment (cf. Chapitre I Section 3.3.1 Théorème 14) dans le cas de tâches sans pertes.

Théorème 41 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDL-RTO}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \qquad \text{(IV.10)}$$

Preuve :

(partie seulement si) : la preuve de cette partie est menée par contradiction en supposant qu'il existe une tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, qui ne vérifie pas l'équation (IV.10) c'est-à-dire que $\Omega^{EDL-RTO}_{\mathcal{T}(\tau)}(\tau, d_i) < \sum_{j=1}^{i} c_j(\tau)$. A partir du Théorème 15 (cf. Chapitre I Section 3.3.1), il paraît évident que le fait d'appliquer EDL-RTO à $\mathcal{T}(\tau)$ engendre une maximisation des temps creux sur n'importe quel intervalle $[\tau, t]$, $t \geq \tau$, et en particulier sur $[\tau, d_i]$. La durée totale d'exécution requise par les tâches apériodiques critiques sur $[\tau, d_i]$ est donnée par $\sum_{j=1}^{i} c_j(\tau)$. Puisque les tâches sont ordonnancées par échéances décroissantes, nous pouvons en conclure que R_i ne s'exécutera pas dans le respect de son échéance et par conséquent que R ne peut être acceptée.

(partie si) : Supposons que pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, la condition (IV.10) est vérifiée. Le fait d'appliquer EDL-RTO à $\mathcal{T}(\tau)$ depuis l'instant τ permet de garantir l'ordonnançabilité de l'ensemble des requêtes périodiques tout en maximisant la quantité totale de temps creux sur tout intervalle $[\tau, t]$, $\tau \leq t$, selon le Théorème 15. Il paraît évident que seules les tâches apériodiques critiques moins urgentes que R peuvent être affectées par l'arrivée de R. Toute tâche apériodique critique R_i est ordonnançable si la somme des exécutions requises par toutes les tâches ayant une échéance inférieure ou égale à d_i, est inférieure ou égale à la somme des temps creux dans $[\tau, d_i]$. Etant donné que cette condition correspond à (IV.10) et qu'elle est vérifiée par n'importe quelle tâche R_i telle que $d_i \geq d$, il en découle que R sera acceptée. □

Dans ce cas, seule la sommation des temps creux $\Omega^{EDL-RTO}_{\mathcal{T}(\tau)}(\tau, d_i)$ calculée entre τ et d_i, diffère vis-à-vis du test d'acceptation sous un modèle de tâches classiques sans pertes.

La description algorithmique du test d'acceptation est présentée ci-après :

Algorithme 6
EDL-RTO schedulability algorithm ;
début
soit $\mathcal{R}(\tau)$ = *ensemble des tâches apériodiques critiques actives au temps* τ
tantque *(une tâche apériodique critique* R *survient au temps* τ*)* **faire**
soit ordonnançable $= 1$
choisir la tâche R_j *dans* $\mathcal{R}(\tau) \cup \{R\}$ *de plus grande échéance*
calculer $(\mathcal{K}(\tau), \mathcal{D}^*(\tau))$ *entre* τ *et* d_j
pour *toutes les tâches* R_i *de* $\mathcal{R}(\tau) \cup \{R\}$ *telles que* $d_i \geq d$
calculer la laxité $\delta_i(\tau)$
si *(*$\delta_i(\tau) < 0$*)*
ordonnançable $= 0$ *et stop*
finsi
finpour
return *ordonnançable*
fintantque
fin

Le test d'acceptation s'appuie sur l'information relative à la laxité de chaque tâche apériodique du système. Lorsqu'une tâche apériodique critique survient, la localisation

et la durée des temps creux situés dans l'intervalle compris entre le temps courant et la plus grande date d'échéance parmi les tâches apériodiques présentes dans le système, sont établies. Ensuite le calcul de la laxité de toutes les tâches apériodiques possédant une échéance supérieure ou égale à celle de la requête occurrente, est effectué. S'il n'y a pas assez de laxité (c'est-à-dire que la laxité est négative), alors la tâche apériodique occurrente est rejetée puisqu'elle ne pourra pas être exécutée sans entraîner de violations d'échéances. Le test d'acceptation proposé présente une complexité en $O(\lfloor \frac{R}{p} \rfloor n + aper(\tau))$ dans le pire-cas, où $aper(\tau)$ désigne le nombre de tâches apériodiques critiques actives au temps τ dont l'échéance est supérieure ou égale à l'échéance de la requête occurrente. En effet, étant donné que le vecteur statique des temps creux est mémorisé, la reconstruction de la séquence n'est nécessaire qu'entre τ et la plus grande échéance de tâche rouge parmi les tâches prêtes à τ, d'où le facteur $\lfloor \frac{R}{p} \rfloor n$.

2 Ordonnancement des tâches apériodiques sous BWP

Supposons à présent que les tâches périodiques avec contraintes de QoS sont ordonnancées selon l'algorithme BWP. La figure IV.8 décrit la schématique d'ordonnancement relative à l'utilisation d'un serveur de tâches apériodiques avec l'ordonnanceur BWP.

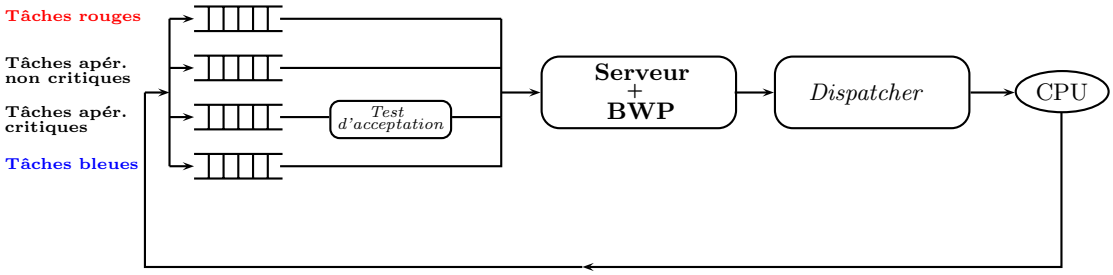

FIG. IV.8 – Schéma d'ordonnancement des tâches apériodiques sous BWP

– **Cas n°1 : pas de tâches apériodiques critiques ou non critiques**

Cette situation correspond exactement à la définition du comportement de l'ordonnanceur BWP. Les tâches rouges et les tâches bleues occurrentes entrent directement dans le système. Les tâches rouges sont ordonnancées au plus tôt selon l'algorithme EDF. Dès lors qu'il n'y a plus de tâches rouges prêtes, les tâches bleues sont à leur tour ordonnancées au plus tôt selon EDF.

– **Cas n°2 : présence de tâches apériodiques non critiques uniquement**

Dans le cas où des tâches apériodiques non critiques surviennent, celles-ci sont acceptées sans condition aucune au sein du système et sont exécutées conjointement

avec les tâches rouges selon la politique du serveur de tâches apériodiques utilisé. L'exécution des tâches bleues est suspendue jusqu'à ce qu'il n'y ait plus de tâches apériodiques à exécuter.

– **Cas n°3 : présence de tâches apériodiques critiques uniquement**

Les tâches apériodiques critiques entrent dans le système sur test d'acceptation ; leur acceptation ne devant pas remettre en cause le respect des échéances de toutes les tâches rouges présentes dans le système. Le test d'acceptation mis en œuvre dépend du serveur de tâches apériodiques utilisé. Les tâches bleues sont les tâches les moins prioritaires du système.

– **Cas n°4 : présence de tâches apériodiques critiques et non critiques**

Les tâches apériodiques non critiques entrent directement dans le système tandis que les tâches apériodiques critiques subissent un test d'acceptation pour intégrer le système. Ensuite, l'ordonnancement conjoint des tâches rouges, des tâches apériodiques critiques et non critiques est lié au serveur de tâches apériodiques mis en œuvre. Les tâches bleues ne sont pas élues par le modèle d'ordonnancement tant que des tâches apériodiques sont présentes dans le système.

2.1 Le modèle d'ordonnancement BG-BWP

2.1.1 Gestion des tâches apériodiques non critiques

2.1.1.1 Description. Nous avons vu précédemment que dans le cas de tâches périodiques basées sur le modèle Skip-Over, le serveur BG ordonnance les tâches apériodiques lorsqu'il n'y a aucune tâche *rouge* active dans le système. Quant aux instances de tâches bleues, celles-ci sont exécutées lorsqu'il n'y a plus ni de tâches rouges ni de tâches apériodiques à l'état prêt au sein du système.

2.1.1.2 Illustration. Considérons l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$ introduit dans la section précédente. Rappelons que le facteur d'utilisation équivalent de cet ensemble de tâches en prenant en compte les pertes est de $U_p^* = 90\%$, ce qui garantit l'ordonnancement des instances rouges obligatoires. On suppose à présent qu'une requête apériodique non critique de durée 10 unités de temps survient au temps $t = 120$. La figure IV.9 visualise les exécutions des différentes tâches.

Les tâches rouges sont toujours ordonnancées au plus tôt selon l'algorithme EDF, et leur comportement n'est jamais perturbé par la présence des autres types de tâches, ce qui permet de garantir leurs exécutions. Le temps de réponse observé pour la tâche apériodique dans ce cas est de 22 unités de temps. Remarquons que l'exécution de

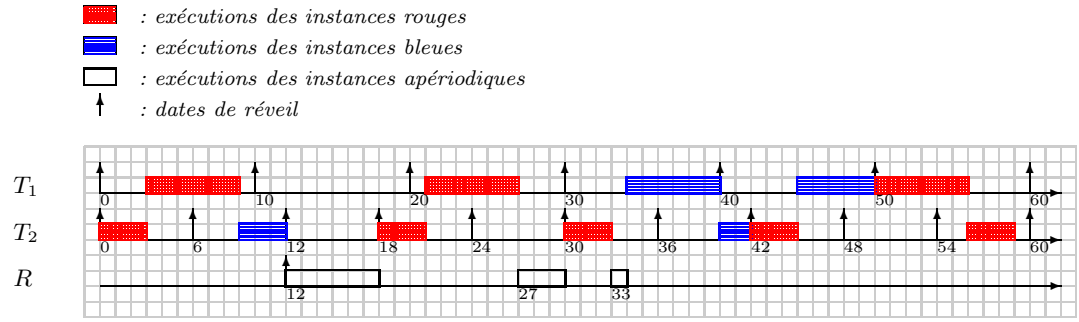

FIG. IV.9 – Gestion des apériodiques non critiques sous BG-BWP

l'instance de tâche bleue T_2 réveillée au temps $t = 6$ est tentée ici (contrairement au modèle BG-RTO). Cette instance réussit à s'exécuter dans le respect de son échéance si bien que l'instance suivante réveillée au temps $t = 12$ est encore bleue. Ainsi, lorsque la requête apériodique survient à $t = 12$, elle bénéficie de cette instance bleue moins prioritaire pour s'exécuter.

2.1.2 Gestion des tâches apériodiques critiques

L'acceptation des tâches apériodiques sous BG-BWP nécessite la construction préliminaire de la fonction de disponibilité f^{EDS} appliquée à un modèle de tâches BWP. Celle-ci représente l'ensemble des temps creux disponibles lorsque les tâches périodiques rouges sont exécutées au plus tôt. La séquence EDS-BWP doit en effet être déterminée pour pouvoir calculer la somme des temps creux entre l'instant d'occurrence d'une tâche apériodique critique et son échéance, de manière à valider ou non l'acceptation de cette nouvelle tâche au sein du système. Les équations de calcul de la séquence EDS (Earliest Deadline as Soon as possible) présentées ci-après pour le modèle de tâches BWP, sont dérivées de celles établies par Chetto et Chetto pour un modèle de tâches sans pertes [CC90].

2.1.2.1 Calcul des temps creux selon EDS sous le modèle BWP

Calcul du vecteur statique des dates de réveil. Le vecteur $\mathcal{E}(\tau) = (\tau, e_{h+1}, ..., e_i, ..., e_q)$ représente les instants supérieurs ou égaux à τ (le temps courant) précédant un temps creux. Soit h l'index tel que $e_h = \sup\{e; e \in \mathcal{E} \text{ et } e > \tau\}$. Comme dans le cas statique, tous les instants e_i correspondent aux dates de réveil distinctes des tâches périodiques.

Calcul du vecteur dynamique des temps creux. Le vecteur dynamique des temps creux $\mathcal{D}(\tau) = (\Delta_h(\tau), \Delta_{h+1}(\tau), ..., \Delta_i(\tau), ..., \Delta_q(\tau))$ traduit les longueurs des temps creux correspondant aux différents instants du vecteur des dates de réveil. $\Delta_i(\tau)$ représente la longueur du temps creux se terminant au temps e_i. Le théorème suivant fournit

la formule de récurrence pour le calcul du vecteur $\mathcal{D}(\tau)$ sous BWP, tenant compte des pertes s_j autorisées par chacune des tâches T_j :

Théorème 42 *Les durées des temps creux du vecteur* $\mathcal{D}(\tau) = (\Delta_h(\tau), \Delta_{h+1}(\tau), ..., \Delta_i(\tau), ..., \Delta_q(\tau))$ *calculées sur* $[0, P'[$ *pour des tâches BWP sont définies comme suit :*

$$\Delta_i(\tau) = \sup(0, F_i(\tau)), \quad \textit{pour } i = \tau \textit{ à } q \tag{IV.11}$$

$$\textit{avec } F_i(\tau) = e_i - \sum_{j=1}^{n} (\lceil \frac{e_i}{p_j} \rceil - \sup(0, \lceil \frac{e_i + (1 - m_j(\tau))p_j}{p_j s_j} \rceil) + \sup(0, \lceil \frac{(1 - m_j(\tau))p_j}{p_j s_j} \rceil))c_j - \sum_{k=i+1}^{q} \Delta_k(\tau)$$

Preuve :

La séquence BWP observée sur $[0, e_i]$ est identique à la séquence RTO observée sur $[(s_j - m_j(\tau))\, p_j, e_i + (s_j - m_j(\tau))\, p_j]$, où l'on tient compte du décalage $m_j(\tau)$, $0 \leq m_j(\tau) \leq (s_i - 1)$, associée à la tâche T_j. L'évaluation des pertes sur cet intervalle correspond au nombre d'instances bleues présentes dans la séquence RTO considérée sur $[(s_j - m_j(\tau))p_j, e_i + (s_j - m_j(\tau))p_j]$. Cette quantité est égale au nombre de pertes observées sur l'intervalle $e_i + (s_j - m_j(\tau))p_j$ auquel on soustrait le nombre de pertes observées sur l'intervalle $(s_j - m_j(\tau))p_j$. Les pertes totales par tâche sur $[0, e_i]$ s'élèvent alors à $\sup(0, \lceil \frac{e_i + (s_j - m_j(\tau))p_j - (s_j - 1)p_j}{p_j . s_j} \rceil)c_j - \sup(0, \lceil \frac{(s_j - m_j(\tau))p_j - (s_j - 1)p_j}{p_j . s_j} \rceil)c_j$, soit après simplifications $\sup(0, \lceil \frac{e_i + (1 - m_j(\tau))p_j}{p_j s_j} \rceil) - \sup(0, \lceil \frac{(1 - m_j(\tau))p_j}{p_j s_j} \rceil)c_j$. □

Illustration du calcul de $f^{EDS-BWP}$. Considérons l'ensemble de tâches $\mathcal{T} = \{T_1, T_2\}$ constitué de deux tâches périodiques BWP, $T_1(6, 10, 2)$ et $T_2(3, 6, 2)$ avec $s_1 = s_2 = 2$. Le calcul des intervalles de temps creux fournit la séquence EDS-BWP représentée sur la figure IV.10.

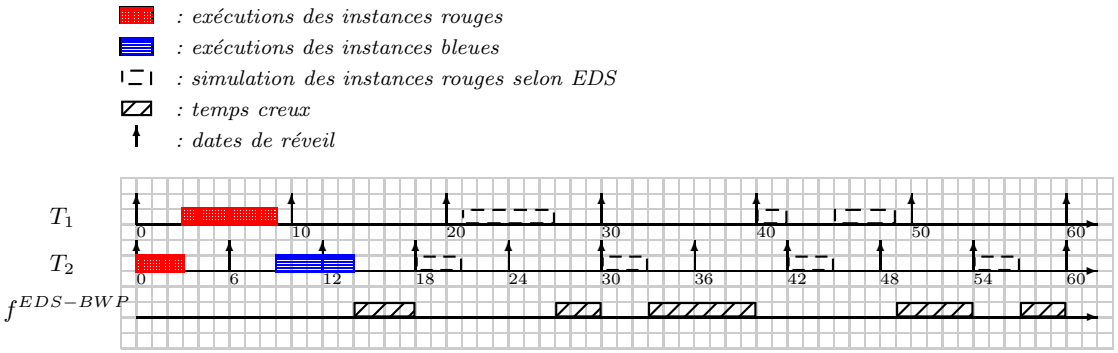

FIG. IV.10 – Calcul des temps creux selon EDS sous BWP

2.1.2.2 Test d'acceptation des apériodiques critiques sous BG-BWP. Le test d'acceptation dans le cas de tâches avec pertes sous BWP s'appuie sur celui énoncé précédemment (cf. Chapitre I Théorème10) dans le cas de tâches sans pertes.

Théorème 43 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega_{\mathcal{T}(\tau)}^{EDS-BWP}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{IV.12}$$

Dans ce cas, seule la somme des temps creux $\Omega_{\mathcal{T}(\tau)}^{EDS-BWP}(\tau, d_i)$ calculée entre τ et d_i sur la base d'une séquence EDS-BWP, diffère vis-à-vis du test d'acceptation sous un modèle de tâches classique sans pertes.

Notons qu'une fois acceptées, les tâches apériodiques critiques sont ordonnancées au plus tôt selon l'algorithme EDF, conjointement avec les tâches périodiques rouges.

2.2 Le modèle d'ordonnancement TBS-BWP

2.2.1 Gestion des tâches apériodiques non critiques

2.2.1.1 Description. Les conditions d'ordonnançabilité relatives à l'utilisation du serveur TBS avec un ensemble de tâches périodiques RTO (cf. Section 1.2 Théorèmes 37 et 38) s'appliquent également dans le cas du modèle BWP. Il en découle que la formule utilisée par le serveur TBS pour assigner une échéance fictive aux tâches apériodiques occurrentes sous le modèle de tâches BWP est identique à celle utilisée sous le modèle de tâches RTO (cf. Section 1.2 Equation IV.4). Quant à la gestion des tâches bleues sous le modèle d'ordonnancement TBS-BWP, celles-ci sont exécutées lorsqu'il n'y a plus de tâches rouges ni de tâches apériodiques à l'état prêt dans le système.

Illustrons à présent sur un exemple, la gestion sous TBS-BWP des tâches apériodiques non critiques.

2.2.1.2 Illustration. Considérons l'ensemble précédent $\mathcal{T}$ constitué de deux tâches périodiques $T_1(6, 10, 2)$ et $T_2(3, 6, 2)$. La figure IV.11 fournit un exemple d'ordonnancement produit par le serveur TBS avec $U_s = 1 - U_p^* = 10\%$ (seule la largeur de bande garantie est allouée au serveur). La condition $U_p^* + U_s \leq 1$ est vérifiée, le système considéré est donc ordonnançable. Une requête apériodique de durée d'exécution au pire cas de 10 unités de temps, survient à l'instant $t = 12$. Elle reçoit alors une échéance fictive $d = \max(12, 0) + \frac{10}{0.10} = 12 + 100 = 112$. Les requêtes dont les échéances sont plus proches que d sont ordonnancées en priorité. Par conséquent, nous pouvons observer un temps de réponse à la requête apériodique de 22 unités de temps.

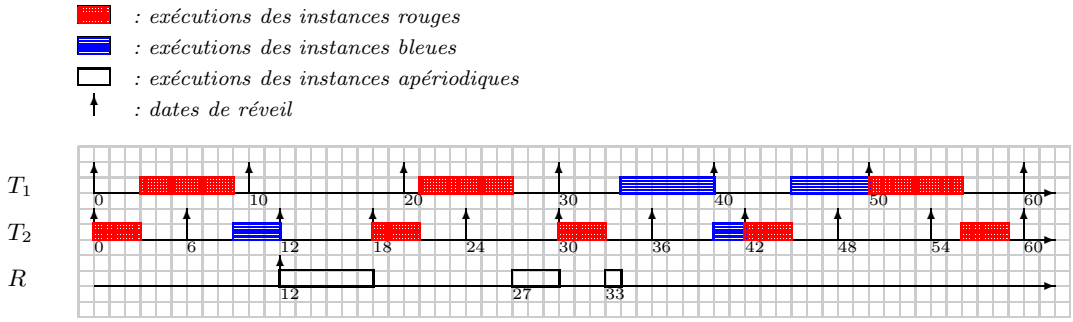

FIG. IV.11 – Gestion des apériodiques non critiques sous TBS-BWP

2.2.2 Gestion des tâches apériodiques critiques

Sous l'hypothèse selon laquelle les tâches apériodiques critiques sont servies selon la technique FIFO (First In First Out), la gestion de ce type de tâches sous le modèle d'ordonnancement TBS-BWP est identique à celle exposée dans le cas de TBS-RTO. Ainsi, l'acceptation des requêtes apériodiques critiques sous TBS-BWP repose entièrement sur le test décrit précédemment dans le théorème 39 (cf. Section 1.2). Les requêtes apériodiques critiques acceptées sont ensuite ordonnancées selon EDF de façon conjointe avec des les tâches rouges. Quant aux tâches bleues, celles-ci sont toujours considérées comme les tâches les moins prioritaires du système.

2.3 Le modèle d'ordonnancement TB*-BWP

2.3.1 Gestion des tâches apériodiques non critiques

2.3.1.1 Description. Comme dans le cas du serveur TBS, les équations introduites pour le modèle d'ordonnancement TB*-RTO sont valables pour le modèle TB*-BWP. Ainsi, la formule récurrente pour le calcul de l'échéance assignée à la requête apériodique au moment de son occurrence (cf. Section 1.3), est utilisée ici de la même manière dans le cas de tâches périodiques ordonnancées par BWP.

2.3.1.2 Illustration. Nous illustrons à présent sur un exemple le comportement du modèle d'ordonnancement TB*-BWP au niveau de la gestion des tâches apériodiques non critiques. Considérons toujours l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$ utilisé précédemment pour illustrer les modèles d'ordonnancement BG-BWP et TBS-BWP. Le facteur équivalent d'utilisation du processeur est égal à $U_p^* = 90\%$. La largeur de bande allouée au serveur TB* est telle que $U_s = 1 - U_p^* = 10\%$, de manière à garantir l'ordonnançabilité de l'ensemble hybride de tâches. Une requête apériodique survient à l'instant $t = 12$. Elle reçoit alors une échéance $d_k^0 = \max(12, 0) + \frac{10}{0.1} = 12 + 100 = 112$ selon l'algorithme TBS. En appliquant les équations du calcul de $I_a(t, d_k^s)$ et $I_f(t, d_k^s)$ adaptées au cas Skip-Over, nous obtenons les échéances raccourcies rassemblées dans la Table IV.3. L'ordonnancement résultant de l'assignation de la plus petite échéance

possible $d_k^* = d_k^5 = 31$ à la requête apériodique est illustré sur la figure IV.12.

Etape	0	1	2	3	4	5
d_k^s	112	76	55	43	34	31

TAB. IV.3 – Dates d'échéances calculées par l'algorithme TB* sous BWP

Le temps de réponse optimal observé pour la requête R est alors de 19 unités de temps, ce qui représente un meilleur temps de réponse que celui obtenu dans le cas des serveurs BG et TBS.

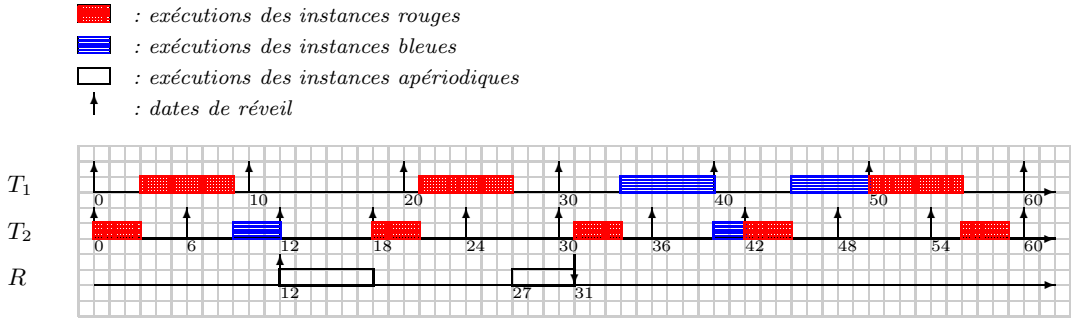

FIG. IV.12 – Gestion des apériodiques non critiques sous TB*-BWP

2.3.2 Gestion des tâches apériodiques critiques

Sous l'hypothèse selon laquelle les tâches apériodiques critiques sont servies selon la technique FIFO (First In First Out), la gestion de ce type de tâches sous le modèle d'ordonnancement TB*-BWP est identique à celle exposée dans le cas de TB*-RTO. Ainsi, l'acceptation des requêtes apériodiques critiques sous TB*-BWP repose entièrement sur le test décrit précédemment dans le théorème 40 (cf. Section 1.3). Les requêtes apériodiques critiques acceptées sont ensuite ordonnancées selon EDF de façon conjointe avec des les tâches rouges.

2.4 Le modèle d'ordonnancement EDL-BWP

2.4.1 Gestion des tâches apériodiques non critiques

2.4.1.1 Description. Dans le but de minimiser le temps de réponse des tâches apériodiques non critiques, le modèle d'ordonnancement EDL-BWP consiste à exécuter les tâches apériodiques non critiques au plus tôt dans les temps creux calculés en exécutant les tâches rouges au plus tard. Ceci nécessite donc une adaptation du calcul de temps creux sous EDL à un modèle de tâches Skip-Over ordonnancées par BWP.

L'application de l'algorithme EDL à des tâches BWP a été traitée dans le chapitre précédent (cf. Chapitre III Section 2.2). Notons que l'intervalle des temps creux utile pour la gestion d'une requête apériodique est compris entre l'instant d'arrivée de la requête apériodique et sa fin d'exécution non connue *a priori*. L'algorithme du serveur EDL utilisé pour le calcul des temps creux pour la gestion des tâches apériodiques non critiques sous un modèle de tâches périodiques BWP est décrit ci-dessous :

Algorithme 7
EDL-BWP Algorithm ;
début
soit $\mathcal{R}(\tau)$ = *ensemble des tâches apériodiques actives à l'instant* τ
tantque *((une tâche apériodique survient à l'instant* τ*) et (*$\mathcal{R}(\tau) = \varnothing$*))* **faire**
calculer $\mathcal{K} = (k_0, k_1, ..., k_i, k_{i+1}, ..., k_q)$
soit $k_i(\tau) = k_q$
soit $j = 0$
tantque *(*$k_i(\tau) \geq \tau$*)* **faire**
$\Delta_i^*(\tau) = \sup(0, F_i(\tau))$
ajouter $k_i(\tau)$ *dans* $\mathcal{K}(\tau)$ *et* $\Delta_i^*(\tau)$ *dans* $\mathcal{D}^*(\tau)$
$j = j + 1$
$k_i(\tau) = k_{q-j}$
fintantque
return $(\mathcal{K}(\tau), \mathcal{D}^*(\tau))$
fintantque
fin

L'algorithme proposé est exécuté à chaque fois qu'une nouvelle tâche apériodique survient, alors qu'aucune tâche apériodique n'étant présente auparavant dans le système. Comparé à l'algorithme EDL-RTO pour lequel le vecteur $\mathcal{K}$ est calculé hors-ligne, l'algorithme EDL-BWP doit calculer le vecteur statique des échéances $\mathcal{K}$ en-ligne. Ceci est dû au fait que des décalages imprévisibles vis-à-vis de la séquence RTO statique de départ apparaissent suite aux exécutions dynamiques réussies des tâches bleues. Ensuite, le vecteur dynamique des temps creux $\mathcal{D}^*(\tau)$ doit être calculé entièrement car pour la même raison, nous n'avons pas accès à la connaissance du vecteur statique $\mathcal{D}^*$ dans le cas EDL-BWP.

Illustrons à présent le fonctionnement du modèle d'ordonnancement EDL-BWP sur un exemple.

2.4.1.2 Illustration. Considérons de nouveau l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$. Cet ensemble de tâches périodiques avec contraintes de QoS est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 12$. On suppose qu'une requête apériodique de durée 10 unités de temps survient au temps $\tau = 12$, impliquant un calcul en-ligne des temps creux. A partir des formules adaptées à un modèle de tâches BWP, nous calculons le vecteur dynamique des échéances $\mathcal{K}(\tau) = (12, 24, 30, 36, 48, 50)$ ainsi

que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (9, 0, 3, 6, 0, 7)$, comme l'illustre la figure IV.13

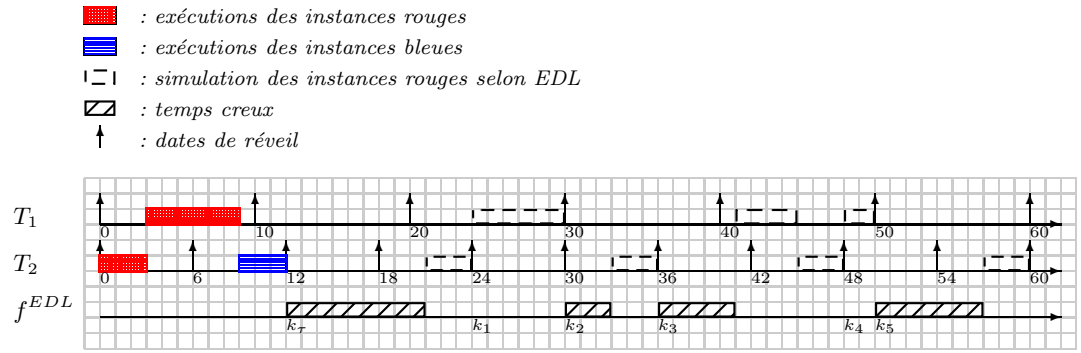

FIG. IV.13 – Calcul des temps creux en dynamique à l'instant $\tau = 12$

L'ordonnancement au plus tôt de la requête apériodique occurrente dans les temps creux de la séquence EDL établie au temps $\tau = 12$ est représentée sur la figure IV.14. Nous observons que la requête reçoit bien 10 unités de temps (9 unités de temps dans le premier intervalle débutant à $\tau = 12$ et 1 unité de temps dans le second débutant à $t = 30$). Son temps de réponse (optimal) est ici de 19 unités de temps. Ce temps est identique à celui observé avec le serveur TB* car les deux serveurs EDL et TB* sont tous les deux optimaux du point de vue de la minimisation du temps de réponse des requêtes apériodiques.

Notons cependant que dans le cas où la requête apériodique s'exécuterait avec une durée effective (ACET) inférieure à sa durée d'exécution au pire-cas (WCET), le temps de réponse observé sous EDL serait meilleur que celui observé sous TB*. Supposons en effet que la requête R ait une durée d'exécution effective telle que $ACET = 0.90 * WCET$, soit 9 unités de temps. Les temps de réponse offerts sous TB*-BWP et EDL-BWP seraient alors respectivement égaux à 18 et 9 unités de temps (cf. figures IV.12 et IV.14), soit un temps de réponse double avec TB* par rapport à EDL.

2.4.2 Gestion des tâches apériodiques critiques

Considérons à présent le problème de l'acceptation de tâches apériodiques à contraintes strictes. On suppose que le système peut présenter plusieurs tâches apériodiques critiques à l'état prêt au temps τ correspondant aux différentes requêtes précédemment acceptées. Par conséquent, celles-ci n'ont pas terminé leur exécution à τ. On note $\mathcal{R}(\tau) = \{R_i(c_i(\tau), d_i),\ i{=}1 \text{ à } q(\tau)\}$ l'ensemble des q tâches apériodiques critiques présentes dans le système à τ. $c_i(\tau)$ désigne la durée d'exécution dynamique de R_i (c'est-à-dire sa durée

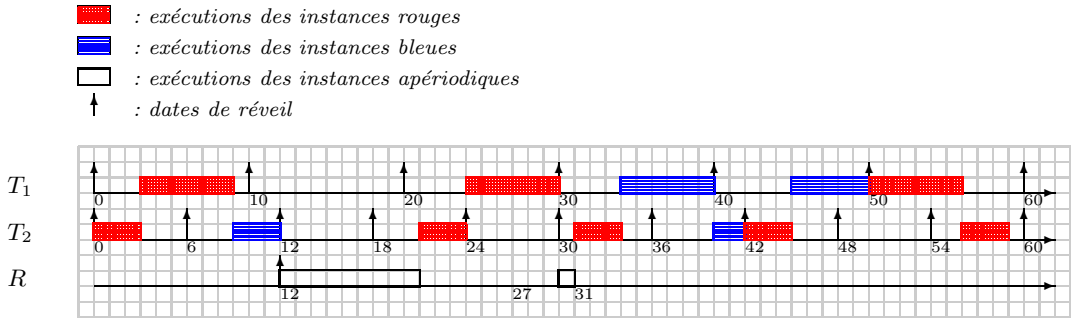

FIG. IV.14 – Gestion des apériodiques non critiques sous EDL-BWP

d'exécution restante) et d_i son échéance. L'ensemble $\mathcal{R}(\tau)$ est ordonné tel que $i < j$ implique $d_i \leq d_j$.

Le test d'acceptation dans le cas de tâches avec pertes sous BWP s'appuie sur celui énoncé précédemment (cf. Chapitre I Section 3.3.1 Théorème 14) dans le cas de tâches sans pertes.

Théorème 44 *$R(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega_{\mathcal{T}(\tau)}^{EDL-BWP}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{IV.13}$$

Preuve :

(partie seulement si) : la preuve de cette partie est menée par contradiction en supposant qu'il existe une tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, qui ne vérifie pas l'équation (IV.10) c'est-à-dire que $\Omega_{\mathcal{T}(\tau)}^{EDL-RTO}(\tau, d_i) < \sum_{j=1}^{i} c_j(\tau)$. A partir du Théorème 15 (cf. Chapitre I Section 3.3.1), il paraît évident que le fait d'appliquer EDL-RTO à $\mathcal{T}(\tau)$ engendre une maximisation des temps creux sur n'importe quel intervalle $[\tau, t]$, $t \geq \tau$, et en particulier sur $[\tau, d_i]$. La durée totale d'exécution requise par les tâches apériodiques critiques sur $[\tau, d_i]$ est donnée par $\sum_{j=1}^{i} c_j(\tau)$. Puisque les tâches sont ordonnancées par échéances décroissantes, nous pouvons en conclure que R_i ne s'exécutera pas dans le respect de son échéance et par conséquent que R ne peut être acceptée.

(partie si) : Supposons que pour toute tâche $R_i \in \mathcal{R}(\tau) \cup \{R\}$ telle que $d_i \geq d$, la condition (IV.10) est vérifiée. Le fait d'appliquer EDL-RTO à $\mathcal{T}(\tau)$ depuis l'instant τ permet de garantir l'ordonnançabilité de l'ensemble des requêtes périodiques tout en maximisant la quantité totale de temps creux sur tout intervalle $[\tau, t]$, $\tau \leq t$, selon le Théorème 15. Il paraît évident que seules les tâches apériodiques critiques moins urgentes que R peuvent être affectées par l'arrivée de R. Toute tâche apériodique critique R_i est ordonnançable si la somme des exécutions requises par toutes les tâches ayant une échéance inférieure ou égale à d_i, est inférieure ou égale à la somme des temps creux dans $[\tau, d_i]$. Etant donné que cette condition correspond à (IV.10) et qu'elle est vérifiée par n'importe quelle tâche R_i telle que $d_i \geq d$, il en découle que R sera acceptée. □

Dans ce cas, seule la sommation des temps creux $\Omega_{\mathcal{I}(\tau)}^{EDL-BWP}(\tau, d_i)$ calculée entre τ et d_i, diffère vis-à-vis du test d'acceptation sous un modèle de tâches classiques sans pertes.

La description algorithmique du test d'acceptation est présentée ci-après :

Algorithme 8
EDL-BWP schedulability algorithm ;
début
soit $\mathcal{R}(\tau)$ = *ensemble des tâches apériodiques critiques actives au temps* τ
tantque *(une tâche apériodique critique* R *survient au temps* τ*)* **faire**
soit ordonnançable $= 1$
choisir la tâche R_j *dans* $\mathcal{R}(\tau) \cup \{R\}$ *de plus grande échéance*
calculer $(\mathcal{K}(\tau), \mathcal{D}^*(\tau))$ *entre* τ *et* d_j
pour *toutes les tâches* R_i *de* $\mathcal{R}(\tau) \cup \{R\}$ *telles que* $d_i \geq d$
calculer la laxité $\delta_i(\tau)$
si *(*$\delta_i(\tau) < 0$*)*
ordonnançable $= 0$ *et stop*
finsi
finpour
return *ordonnançable*
fintantque
fin

Le test d'acceptation fait appel à l'algorithme EDL adapté au modèle de tâches BWP pour la détermination des temps creux qui sont utilisés pour calculer la laxité des tâches apériodiques critiques. Comme dans le cas de EDL-RTO, cette laxité est comparée à zéro, ce qui conditionne le refus éventuel (cas où la laxité est négative) de la tâche apériodique occurrente. La complexité du test d'acceptation proposé est en $O(N + aper(\tau))$ dans le pire-cas, où N représente le nombre de requêtes périodiques rouges sur l'intervalle $[\tau, kP[$ et $aper(\tau)$ désigne le nombre de tâches apériodiques critiques actives au temps τ (y compris la tâche occurrente) dont l'échéance est supérieure ou égale à l'échéance de la requête occurrente.

Dans la suite, nous nous focalisons sur le serveur EDL, ce dernier étant optimal du point de vue de la minimisation du temps de réponse aux requêtes apériodiques. Nous présentons les modèles d'ordonnancement basés sur les 2 nouveaux algorithmes d'ordonnancement de tâches périodiques sous contraintes de QoS introduits dans le Chapitre III, à savoir RLP et RLP/T.

3 Le modèle d'ordonnancement EDL-RLP

Considérons le cas où l'algorithme RLP est retenu pour ordonnancer les tâches périodiques avec contraintes de QoS. Le schéma d'ordonnancement intégrant la gestion des tâches apériodiques sous RLP est représenté sur la figure IV.15.

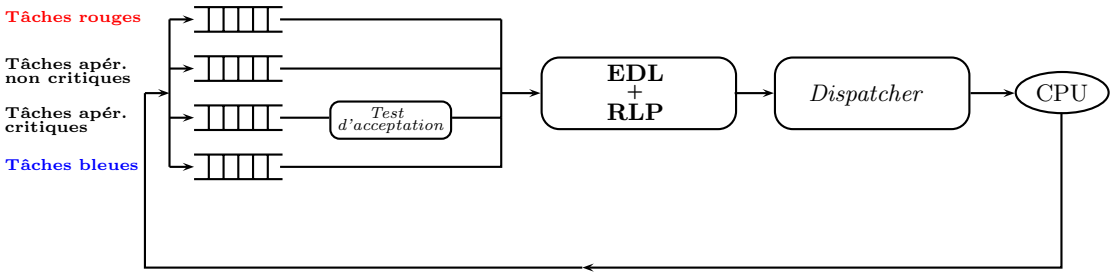

FIG. IV.15 – Schéma d'ordonnancement des tâches apériodiques sous RLP

– **Cas n°1 : pas de tâches apériodiques critiques ou non critiques**

Cette situation correspond exactement à la définition du comportement de l'ordonnanceur RLP. Les tâches rouges et les tâches bleues occurrentes entrent directement dans le système. Les tâches rouges sont ordonnancées au plus tôt selon l'algorithme EDF lorsqu'il n'y a pas de tâches bleues à l'état prêt. Dans le cas contraire, les tâches bleues sont ordonnancées au plus tôt dans les temps creux calculés en exécutant les tâches rouges au plus tard.

– **Cas n°2 : présence de tâches apériodiques non critiques uniquement**

Dans le cas où des tâches apériodiques non critiques surviennent, celles-ci sont acceptées sans condition aucune au sein du système et sont exécutées au plus tôt dans les temps creux calculés en exécutant les tâches rouges au plus tard. L'exécution des tâches bleues est suspendue jusqu'à ce qu'il n'y ait plus de tâches apériodiques à exécuter.

– **Cas n°3 : présence de tâches apériodiques critiques uniquement**

Les tâches apériodiques critiques entrent dans le système sur test d'acceptation. Leur acceptation ne doit pas remettre en cause le respect des échéances de toutes les tâches rouges présentes dans le système. Une fois acceptées, s'il n'y a pas de tâches bleues à l'état prêt, les tâches apériodiques critiques sont ordonnancées conjointement avec les tâches rouges selon l'algorithme EDF. Dans le cas contraire, les tâches bleues sont exécutées au plus tôt en exécutant les tâches apériodiques critiques et les tâches rouges au plus tard.

– **Cas n°4 : présence de tâches apériodiques critiques et non critiques**

Les tâches apériodiques non critiques entrent directement dans le système tandis que les tâches apériodiques critiques subissent un test d'acceptation pour intégrer le système. Ensuite, les tâches apériodiques non critiques sont ordonnancées au

plus tôt dans les temps creux calculés en exécutant à la fois les tâches apériodiques critiques et les tâches rouges au plus tard. Les tâches bleues ne sont pas élues par le modèle d'ordonnancement tant que des tâches apériodiques non critiques sont présentes dans le système.

3.1 Gestion des tâches apériodiques non critiques

3.1.1 Description

La gestion des tâches apériodiques non critiques sous le modèle d'ordonnancement EDL-RLP est identique à celui décrit pour le modèle d'ordonnancement EDL-BWP. Les tâches apériodiques non critiques sont ordonnancées au plus tôt dans les temps creux calculées en exécutant les tâches rouges au plus tard. Dans ce cas, l'algorithme EDL est de nouveau appliqué à des tâches BWP (cf. Chapitre III Section 2.2). L'algorithme du serveur EDL utilisé pour le calcul des temps creux pour la gestion des tâches apériodiques non critiques sous un modèle de tâches périodiques RLP est identique à celui décrit précédemment (cf. Section 2.4 Algorithme 7).

3.1.2 Illustration

Considérons le même ensemble de tâches que celui utilisé pour illustrer les différents modèles d'ordonnancement basés sur BWP, à savoir l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$. On suppose qu'une requête apériodique de durée 10 unités de temps survient au temps $\tau = 12$, impliquant un calcul en-ligne des temps creux. Les vecteurs dynamiques $\mathcal{K}(\tau) = (12, 24, 30, 36, 48, 50)$ et $\mathcal{D}^*(\tau) = (9, 0, 3, 6, 0, 7)$ calculés sont représentés sur la figure IV.16.

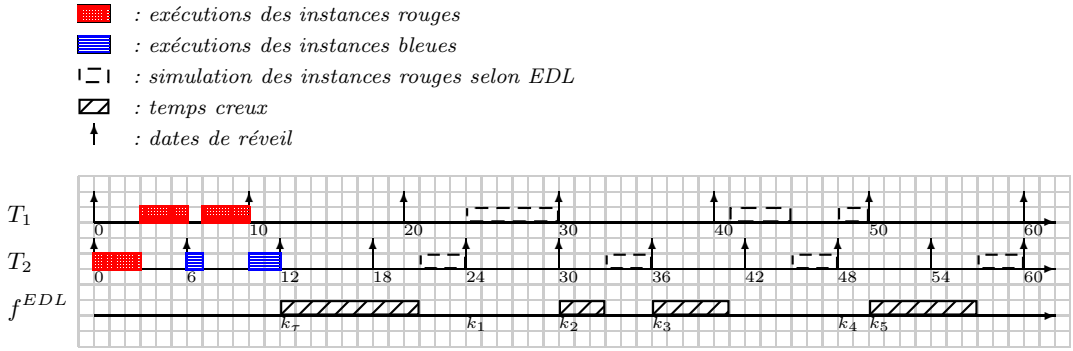

FIG. IV.16 – Calcul des temps creux en dynamique à l'instant $\tau = 12$

L'ordonnancement au plus tôt de la requête apériodique occurrente dans les temps creux de la séquence EDL établie au temps $\tau = 12$ est représentée sur la figure IV.17. Nous observons que la requête reçoit bien 10 unités de temps (9 unités de temps dans

le premier intervalle débutant à $\tau = 12$ et 1 unité de temps dans le second débutant à $t = 30$). Son temps de réponse (optimal) est ici de 19 unités de temps.

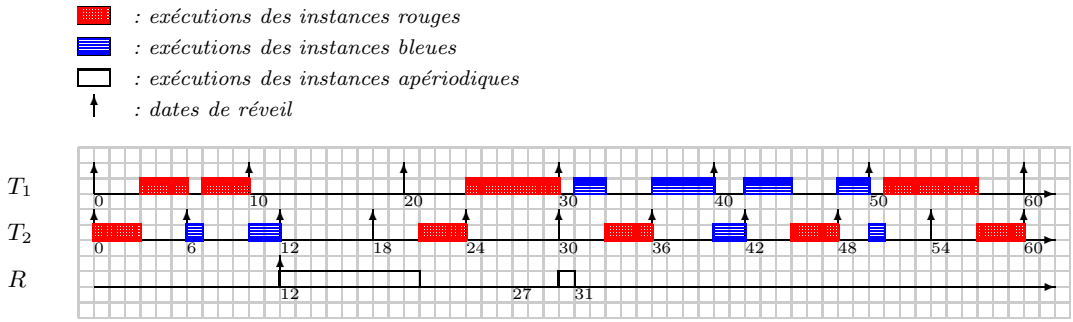

FIG. IV.17 – Gestion des apériodiques non critiques sous EDL-RLP

3.2 Gestion des tâches apériodiques critiques

Etant donné que les équations de construction de la séquence EDL pour la détermination des temps creux sont identiques sous BWP et RLP, le test d'acceptation des tâches apériodiques critiques sous EDL-RLP correspond exactement à celui énoncé pour EDL-BWP (cf. Section 2.4 Théorème 44).

4 Le modèle d'ordonnancement EDL-RLP/T

Considérons enfin le schéma d'ordonnancement propre à l'algorithme RLP/T. Celui-ci est décrit sur la figure IV.18.

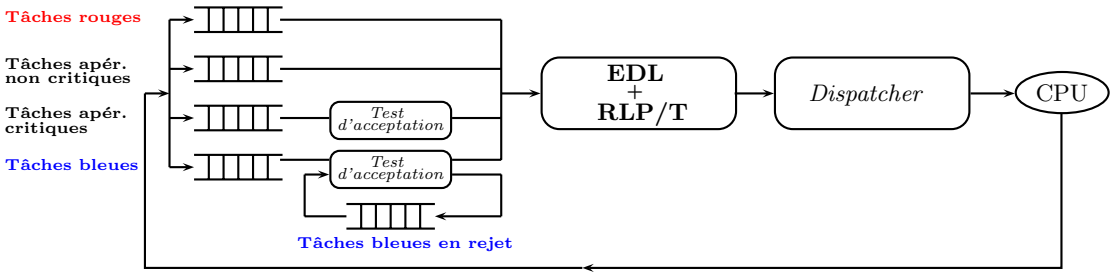

FIG. IV.18 – Schéma d'ordonnancement des tâches apériodiques sous RLP/T

– **Cas n°1 : pas de tâches apériodiques critiques ou non critiques**

Cette situation correspond exactement à la définition du comportement de l'ordonnanceur RLP/T. Les tâches rouges occurrentes entrent directement dans le système

tandis que les tâches bleues entrent sur test d'acceptation. Si le test échoue, la tâche bleue occurrente est définitivement rejetée. Les tâches rouges et les tâches bleues acceptées sont ensuite ordonnancées de façon conjointe selon l'algorithme EDF.

– **Cas n°2 : présence de tâches apériodiques non critiques uniquement**

Dans le cas où des tâches apériodiques non critiques surviennent, celles-ci sont acceptées sans condition aucune au sein du système et sont exécutées au plus tôt selon EDF dans les temps creux en exécutant les tâches rouges au plus tard. Les tâches bleues occurrentes ainsi que celles préalablement acceptées sont toutes placées dans une file de rejet. L'acceptation de l'ensemble de ces tâches est réévaluée dès lors qu'il n'y a plus de tâches apériodiques non critiques à l'état prêt (retour au cas n°1).

– **Cas n°3 : présence de tâches apériodiques critiques uniquement**

Les tâches apériodiques critiques entrent dans le système sur test d'acceptation. Leur acceptation ne doit pas remettre en cause le respect des échéances des tâches rouges ou des tâches apériodiques critiques préalablement acceptées dans le système. Lorsqu'une tâche apériodique critique intègre le système alors qu'il n'y en avait pas auparavant, les tâches bleues préalablement acceptées subissent un nouveau test d'acceptation intégrant l'exécution de la nouvelle tâche apériodique critique. Tant que des tâches apériodiques critiques sont présentes dans le système, les tâches bleues occurrentes entrent dans le système sur la base d'un nouveau test d'acceptation tenant compte des tâches apériodiques critiques déjà acceptées.

– **Cas n°4 : présence de tâches apériodiques critiques et non critiques**

Les tâches apériodiques non critiques entrent directement dans le système tandis que les tâches apériodiques critiques subissent un test d'acceptation pour intégrer le système. Ensuite, les tâches apériodiques non critiques sont ordonnancées au plus tôt en exécutant les tâches apériodiques critiques au plus tard dans les temps creux calculés en ordonnançant les tâches rouges au plus tard également. Les tâches bleues occurrentes ainsi que celles préalablement acceptées sont toutes placées dans une file de rejet. L'acceptation de l'ensemble de ces tâches est réévaluée en tenant compte de la présence des tâches apériodiques critiques dès lors qu'il n'y a plus de tâches apériodiques non critiques à l'état prêt (retour au cas n°3).

4.1 Gestion des tâches apériodiques non critiques

4.1.1 Description

L'objectif est toujours de minimiser le temps de réponse des apériodiques non critiques en garantissant l'exécution des tâches rouges. Les tâches apériodiques non critiques sont ainsi exécutées au plus tôt selon l'algorithme EDF dans les temps creux calculés en

exécutant les tâches rouges au plus tard selon l'algorithme EDL. Lors de l'occurrence d'une tâche apériodique (alors qu'il n'y en avait pas auparavant dans le système), les temps creux d'un ordonnanceur EDL sont calculés. Sous le modèle d'ordonnancement EDL-RLP/T, le système peut présenter à un instant donné, des tâches de type rouge et/ou bleue. Les tâches rouges sont garanties par définition. Les tâches bleues ont été garanties au moment de leur occurrence par le test d'acceptation des bleues mis en œuvre sous RLP/T, mais contrairement aux tâches rouges, elles peuvent être remises en cause par l'exécution des apériodiques non critiques considérées comme plus prioritaires. Ainsi, en présence de tâches apériodiques non critiques, le respect des échéances des tâches bleues n'est plus garanti. Dans ce cas, la gestion des tâches bleues sous RLP/T se ramène à celle observée sous BWP ou RLP. Par conséquent, en présence de tâches apériodiques, le serveur EDL utilisé avec l'ordonnanceur RLP/T possède un comportement identique à celui précédemment décrit pour les modèles EDL-BWP et EDL-RLP (cf. Section 2.4 Algorithme 7).

4.1.2 Illustration

Considérons de nouveau l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$. Cet ensemble de tâches périodiques avec contraintes de QoS est ordonnancé au plus tôt selon l'algorithme EDF jusqu'au temps $\tau = 12$. On suppose qu'une requête apériodique de durée 10 unités de temps survient au temps $\tau = 12$, impliquant un calcul en-ligne des temps creux. A partir des formules de calcul adaptées à un modèle de tâches BWP, nous pouvons extraire le vecteur dynamique des échéances $\mathcal{K}(\tau) = (12, 24, 30, 36, 48, 50)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (9, 0, 3, 6, 0, 7)$, comme l'illustre la figure IV.19

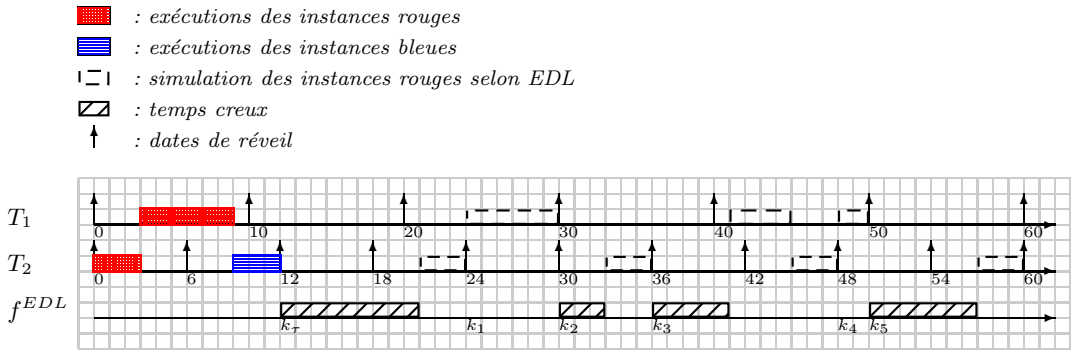

FIG. IV.19 – Calcul des temps creux en dynamique à l'instant $\tau = 12$

L'ordonnancement au plus tôt de la requête apériodique occurrente dans les temps creux de la séquence EDL établie au temps $\tau = 12$ est représentée sur la figure IV.20. La requête apériodique s'exécute en priorité vis-à-vis des instances de tâches bleues. La tâche apériodique reçoit bien 10 unités de temps (9 unités de temps dans le premier intervalle débutant à $\tau = 12$ et 1 unité de temps dans le second débutant à $t = 30$).

Son temps de réponse (optimal) est ici de 19 unités de temps. Ce temps est identique à celui observé dans le cas de EDL-BWP et EDL-RLP. Notons cependant que la QoS observée au niveau des tâches périodiques est meilleure que dans les cas des modèles d'ordonnancement EDL-BWP et EDL-RLP.

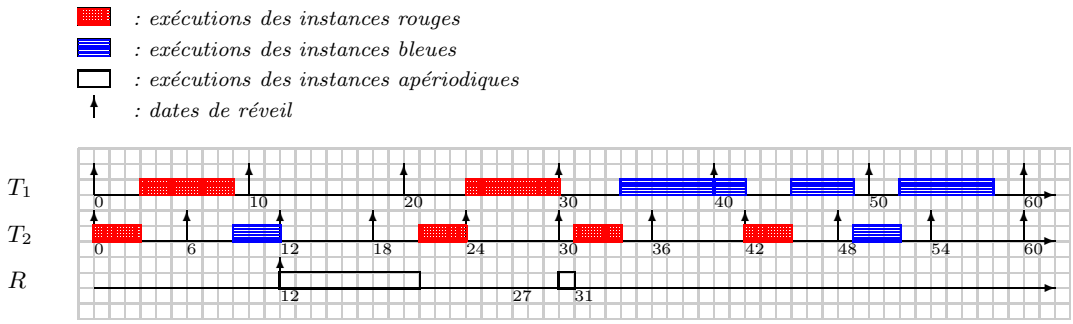

FIG. IV.20 – Gestion des apériodiques non critiques sous EDL-RLP/T

4.2 Gestion des tâches apériodiques critiques

4.2.1 Description

Comme dans le cas des modèles d'ordonnancement EDL-BWP et EDL-RLP, le test d'acceptation des tâches apériodiques critiques se base sur la construction de la séquence EDL appliquée au modèle de tâches Skip-Over, dans laquelle l'exécution des tâches rouges est repoussée au plus tard. Par conséquent, le test d'acceptation des tâches apériodiques critiques sous EDL-RLP/T correspond exactement à celui énoncé pour EDL-BWP (cf. Section 2.4 Théorème 44).

4.2.2 Gestion des tâches bleues

Dans le cas où le système ne présente pas d'activité apériodique, nous avons vu que la gestion des tâches bleues était implémentée sous RLP/T par le biais d'un test d'acceptation (cf. Chapitre IV Section 3.2 Théorème 33) vérifiant l'ordonnançabilité de l'ensemble des tâches périodiques avec pertes au moment de l'occurrence d'une nouvelle instance de tâche bleue.

Dans le cas où des tâches apériodiques non critiques concourent avec les tâches périodiques pour obtenir la ressource processeur, celles-là s'exécutent toujours en priorité vis-à-vis des tâches bleues qui, par définition, peuvent être abandonnées à tout moment. Ainsi, tant qu'il y a des tâches apériodiques non critiques à l'état prêt, aucune tâche bleue ne peut s'exécuter.

En revanche, dans le cas où des tâches apériodiques critiques sont présentes dans le système, un nouveau test d'acceptation des instances de tâches bleues a été défini de

manière à continuer à prendre en compte ce type de tâches. Le nouveau test intègre à la fois les durées d'exécution des tâches apériodiques critiques et celles des tâches bleues préalablement acceptées pour déterminer l'acceptation de la tâche bleue occurrente. Soit τ le temps courant coïncidant avec l'arrivée d'une instance B de tâche bleue. A son arrivée, l'instance de tâche $B(r, c, d)$ est caractérisée par sa date de réveil r, sa durée d'exécution au pire-cas c, et son échéance d, avec $r + c \leq d$. Le système peut présenter en son sein plusieurs tâches bleues à l'instant τ ; chacune d'elles ayant été acceptée avant τ et n'ayant pas terminé son exécution au temps τ. Soit $\mathcal{B}(\tau) = \{B_i(c_i(\tau), d_i), i = 1 \text{ à } bleue(\tau)\}$ l'ensemble des tâches bleues supportées par le système au temps τ. Le paramètre $c_i(\tau)$ est appelé *durée d'exécution dynamique* et représente la durée d'exécution restante de B_i à τ. Nous supposons de plus que $\mathcal{B}(\tau)$ est ordonné de telle manière que $i < j$ implique $d_i \leq d_j$. Par ailleurs, plusieurs tâches apériodiques critiques peuvent être présentes à l'instant τ ; chacune d'elles ayant été acceptée avant τ et n'ayant pas terminé son exécution au temps τ. Soit $\mathcal{R}(\tau) = \{R_i(c_i(\tau), d_i), i = 1 \text{ à } aper(\tau)\}$ l'ensemble des tâches apériodiques critiques supportées par le système au temps τ. De même, le paramètre $c_i(\tau)$ représente la durée d'exécution restante de R_i à τ. Nous supposons également que $\mathcal{R}(\tau)$ est ordonné de telle manière que $i < j$ implique $d_i \leq d_j$.

Le test d'acceptation des tâches bleues au sein d'un système ordonnancé selon EDL-RLP/T présenté ci-dessous dans le théorème 45, est basé sur celui établi par Silly et al. [SCE90] pour l'acceptation de requêtes apériodiques critiques dans un système gérant un ensemble de tâches périodiques basiques, c'est-à-dire sans pertes (cf. Chapitre I. Section 3.3.1.4).

Théorème 45 *$B(\tau, c, d)$ est acceptée si et seulement si pour toute tâche $B_i \in \mathcal{B}(\tau) \cup \mathcal{R}(\tau) \cup \{B\}$ telle que $d_i \geq d$, on vérifie $\delta_i(\tau) \geq 0$, avec $\delta_i(\tau)$ défini comme suit :*

$$\delta_i(\tau) = \Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1,\, bleues}^{i} c_j(\tau) - \sum_{j=1,\, aper}^{i} c_j(\tau) \qquad \text{(IV.14)}$$

$\delta_i(\tau)$, appelée laxité de la tâche R_i au temps τ, représente la longueur de l'intervalle de temps qui sépare sa fin d'exécution au plus tôt de son échéance. $\Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i)$ représente la somme totale des temps creux calculée sur l'intervalle $[\tau, d_i]$. Les sommations $\sum_{j=1,\, bleues}^{i} c_j(\tau)$ et $\sum_{j=1,\, aper}^{i} c_j(\tau)$ traduisent la somme des exécutions restantes à l'instant τ sur respectivement, l'ensemble des tâches bleues et l'ensemble des tâches apériodiques critiques dont l'échéance est inférieure ou égale à d_i.

La procédure implémentant ce nouveau test d'acceptation fait appel à l'algorithme EDL adapté à des tâches Skip-Over, pour le calcul dynamique de la somme totale des temps creux. Celle-ci est utilisée pour établir la laxité des tâches bleues et celle des tâches apériodiques critiques qui sont ensuite comparées à zéro. Par conséquent, le test d'acceptation proposé a une complexité en $O(\lfloor \frac{R}{p} \rfloor n + bleue(\tau) + aper(\tau))$ dans le pire-cas, où n représente le nombre de tâches périodiques, R la plus grande échéance, p la plus petite période, $bleue(\tau)$ le nombre de tâches bleues actives à l'instant τ dont l'échéance

est supérieure ou égale à l'échéance de la tâche bleue occurrente, et $aper(\tau)$ le nombre de tâches apériodiques critiques actives à l'instant τ dont l'échéance est supérieure ou égale à l'échéance de la tâche bleue occurrente.

Le pseudo-code du test d'acceptation mis en œuvre sous le modèle d'ordonnancement EDL-RLP/T est fourni ci-dessous dans l'algorithme 9.

Algorithme 9
EDL-RLP/T_blue_acceptance_test_with_aperiodic_tasks*(t : temps courant)*
début
soit $\mathcal{B}(t)$ *l'ensemble des tâches bleues actives au temps t*
soit $\mathcal{R}(t)$ *l'ensemble des tâches apériodiques critiques actives au temps t*
tantque *(une tâche bleue B survient au temps t)* **faire**
ordonnançable =1
Choisir la tâche T_j *de* $\mathcal{B}(t) \cup \mathcal{R}(t) \cup \{B\}$ *possédant la plus grande échéance*
Calculer $f^{EDL}(t, d_j)$
pour *toutes les tâches* T_i *de* $\mathcal{B}(t) \cup \mathcal{R}(t) \cup \{B\}$ *telles que* $d_i \geq d$ **faire**
Calculer la laxité $\delta_i(t)$
si $(\delta_i(t) < 0)$
ordonnançable = 0 et STOP
finsi
finpour
return ordonnançable
fintantque
fin

4.2.3 Illustration

Considérons toujours l'ensemble $\mathcal{T} = \{T_1(6, 10, 2), T_2(3, 6, 2)\}$ pour illustrer notre propos. On suppose qu'une requête apériodique critique pourvue d'une date absolue d'échéance $d = 28$ et d'une durée d'exécution de 5 unités de temps, survient au temps $\tau = 13$, impliquant un calcul en-ligne des temps creux. A partir des formules adaptées à un modèle de tâches BWP, nous calculons le vecteur dynamique des échéances $\mathcal{K}(\tau) = (13, 24, 30, 36, 48, 50)$ ainsi que le vecteur dynamique des temps creux $\mathcal{D}^*(\tau) = (8, 0, 3, 6, 0, 7)$, comme l'illustre la figure IV.21

Testons à présent l'acceptation de la requête apériodique critique en appliquant le test décrit dans le Théorème 44 (cf. Section 2.4). La somme des temps creux dans l'intervalle $[13, 28]$ est donné par $\Omega^{EDL}_{\mathcal{T}(13)}(13, 28) = 8$. La durée d'exécution requise par les différentes tâches apériodiques critiques sur l'intervalle $[13, 28]$ est égale à $\sum_{j=1}^{i} c_j(12) = 5$. Par conséquent, la laxité de la tâche apériodique critique au temps $t = 13$ est égale à $\delta_i(13) = \Omega^{EDL}_{\mathcal{T}(13)}(13, 28) - \sum_{j=1}^{i} c_j(13) = 8 - 5 = 3$, la tâche peut donc être acceptée au sein du système.

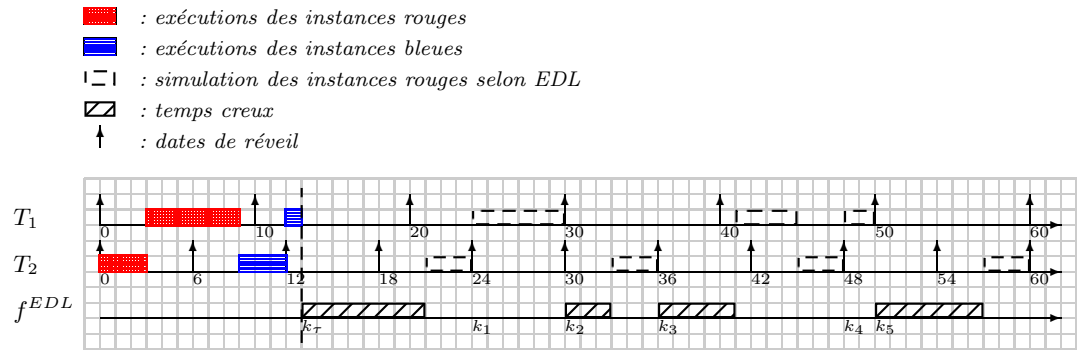

FIG. IV.21 – Calcul des temps creux en dynamique à l'instant $\tau = 12$

A l'instant $t = 13$ correspondant à l'occurrence de la tâche apériodique critique, seule une instance de tâche bleue était présente dans le système. Testons à nouveau son acceptation sur la base du nouveau test introduit précédemment (cf Théorème 45). La somme des temps creux dans les intervalles $[13, 20]$ et $[13, 28]$ est donnée par $\Omega^{EDL}_{\mathcal{T}(13)}(13, 20) = 7$ et $\Omega^{EDL}_{\mathcal{T}(13)}(13, 28) = 8$ respectivement. La durée d'exécution requise par la tâche bleue sur l'intervalle $[13, 20]$ est égale à $\sum_{j=1}^{i} c_j(13) = 6 - 1 = 5$ (l'instance bleue de T_1 réveillée à $t = 10$ a besoin de 5 unités de temps pour finir son exécution). Au niveau de la tâche apériodique critique cette quantité est égale à $\sum_{j=1}^{i} c_j(13) = 5$ (la tâche n'a pas commencé son exécution). Par conséquent, la laxité de l'instance de tâche bleue de T_1 au temps $t = 13$ est égale à $\delta_i(13) = \Omega^{EDL}_{\mathcal{T}(13)}(13, 20) - \sum_{j=1}^{i} c_j(13) = 7 - 5 = 2$, ce qui est acceptable. La laxité de la tâche apériodique critique au temps $t = 13$ est égale à $\delta_i(13) = \Omega^{EDL}_{\mathcal{T}(13)}(13, 28) - \sum_{j=1}^{i} c_j(13) = 8 - 6 = 2$. Cette valeur positive indique que la tâche pourra également s'accomplir dans le respect de son échéance. Ainsi, l'instance de tâche bleue de la tâche T_1 réveillée à $t = 10$ est conservée au sein du système à l'instant $t = 13$ et s'exécute au plus tôt selon l'algorithme EDF conjointement avec la tâche apériodique et les tâches rouges, comme l'illustre la figure IV.22.

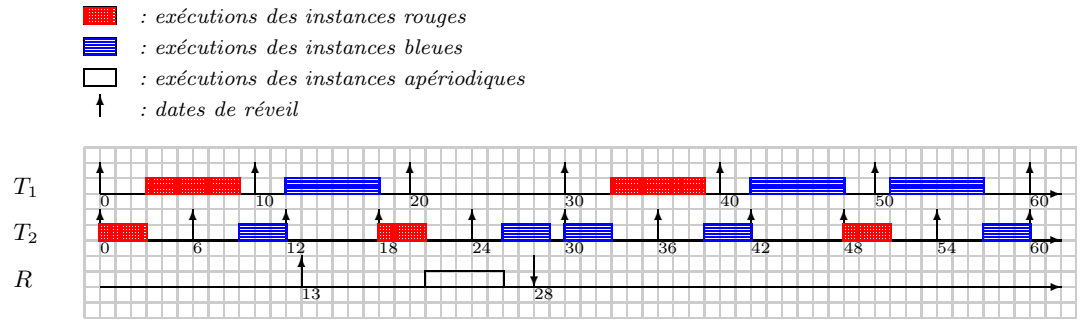

FIG. IV.22 – Gestion des apériodiques critiques sous EDL-RLP/T

5 Résultats de simulation

L'objectif de ce paragraphe est de présenter et commenter les résultats de simulations effectuées dans le but d'évaluer comparativement la performance des modèles d'ordonnancement "*Serveur+OrdonnanceurQoS*" précédemment décrits. Les résultats de simulations menées en fonction de la charge périodique et/ou apériodique appliquée au système, visent à montrer les variations observées au niveau des critères suivants :

- le temps de réponse des tâches apériodiques non critiques,
- le taux de respect des tâches périodiques,
- le taux d'acceptation des tâches apériodiques critiques.

Rappelons quel 'objectif est de minimiser le temps de réponse des tâches apériodiques non critiques et de maximiser le nombre de requêtes apériodiques critiques acceptées, tout en respectant le niveau minimum de QoS des tâches périodiques prédéfini par l'utilisateur.

Dans cette perspective, nous évaluons dans un premier temps l'influence du serveur de tâches apériodiques utilisé, pour un ordonnanceur principal donné tel que RTO ou RLP. La performance du serveur EDL est comparée à celle du serveur Background (BG), du serveur TBS, de l'algorithme TB(2), et enfin à celle du serveur optimal TB*, sous l'ordonnanceur BWP. Puis, nous étudions l'influence de l'ordonnanceur principal, pour un serveur donné. La performance de l'ordonnanceur RLP/T est comparée à celle observée sous RTO, BWP et RLP, lorsque le serveur de tâches apériodiques EDL est utilisé. Les expérimentations en simulation tendent également à évaluer l'impact du paramètre de pertes sur les performances relatives des différentes stratégies.

5.1 Environnement de simulation

L'environnement de simulation présenté dans le chapitre III a été enrichi pour générer et ordonnancer des configurations de tâches apériodiques critiques et/ou non critiques. L'architecture fonctionnelle étendue du simulateur est représentée sur la figure IV.23.

Un *générateur de configuration de tâches apériodiques* a été conçu. Il accepte en entrée 4 paramètres : m, p, U_{ap} et U_{apc}. Les paramètres m et U_{ap} correspondent respectivement au nombre de tâches souhaité pour la configuration de tâches apériodiques non critiques et à la charge processeur associée. De façon similaire, les paramètres p et U_{apc} correspondent respectivement au nombre de tâches souhaité pour la configuration de tâches apériodiques critiques et à la charge processeur associée. En sortie, on obtient une configuration de tâches apériodiques non critiques $\mathcal{R} = \{R_i(r_i, C_i), i = 1 \text{ à } m\}$, et une configuration de tâches apériodiques critiques $\mathcal{S} = \{S_i(r_i, C_i, d_i), i = 1 \text{ à } p\}$. La largeur de bande alloué aux serveurs TB est établie à la valeur $U_s = 1 - U_p^*$ où U_p^* représente le facteur équivalent d'utilisation du processeur correspondant à la configuration de tâches périodiques, ce qui garantie l'ordonnançabilité de l'ensemble hybride de tâches gérées par le simulateur.

Après la génération des ensembles de tâches apériodiques, le simulateur ordonnance ceux-ci en-ligne pour chaque serveur à comparer, ici BG, TBS, TB(2), TB*, et EDL.

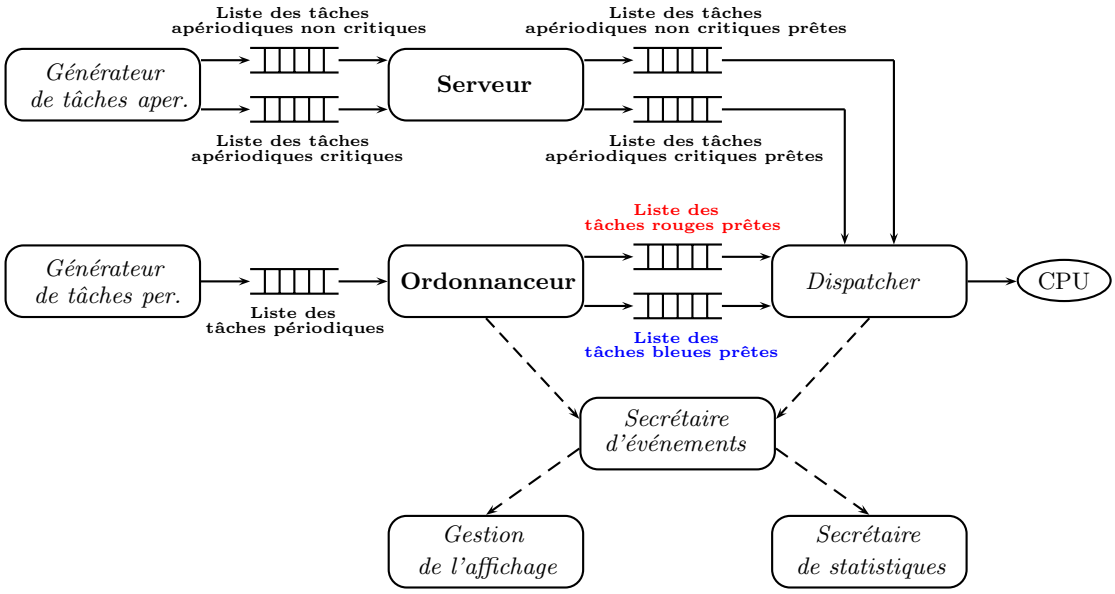

FIG. IV.23 – Architecture fonctionnelle étendue du simulateur

Le *dispatcher* lance l'exécution des instances de tâches prêtes placées en tête de liste par le module d'ordonnancement d'une part et par le serveur d'autre part.

Pour chaque stratégie d'ordonnancement, 50 configurations de tâches périodiques ont été générées, chacune d'elles comprenant 10 tâches avec un *ppcm* des périodes égal à 3360. Pour chaque nouvelle configuration de tâches périodiques, de nouveaux ensembles de tâches apériodiques ont été générées. Les simulations ont été effectuées sur 10 hyperpériodes. Les performances de chacun des modèles d'ordonnancement ont été évaluées en faisant varier la charge périodique du système ainsi que le facteur de pertes appliqué aux tâches.

5.2 Critères mesurés

Trois paramètres ont servi à comparer les performances des modèles d'ordonnancement étudiés :

5.2.1 Le taux de respect

Il s'agit du nombre de requêtes périodiques dont l'exécution s'est achevée avant son échéance. Ce paramètre représente une métrique de la qualité de service observée au niveau des tâches périodiques.

$$T_{QoS} = \frac{\text{nombre d'échéances satisfaites}}{\text{nombre total de requêtes}}$$

5.2.2 Le temps de réponse

C'est la moyenne, sur l'ensemble des tâches apériodiques non critiques exécutées, du temps nécessaire à une tâche pour s'exécuter, normalisé par rapport à sa durée d'exécution.

$$T_{reponse} = \frac{f_i - r_i}{C_i}$$

5.2.3 Le taux d'acceptation

C'est la moyenne du rapport entre le nombres de tâches apériodiques critiques acceptées (garanties) par le test d'acceptation et le nombre de tâches apériodiques générées dans la configuration.

$$T_{accept} = \frac{\text{nombre de tâches garanties}}{\text{nombre de tâches générées}}$$

5.3 Evaluation de l'influence du serveur de tâches apériodiques utilisé

5.3.1 Evaluation du temps de réponse des tâches apériodiques non critiques

Nous rapportons tout d'abord les résultats de simulation obtenus au niveau du temps de réponse moyen des tâches apériodiques non critiques, en fonction du serveur de tâches apériodiques utilisé. Ces résultats tendent à vérifier que l'algorithme EDL offre d'une part de meilleures performances que les algorithmes d'ordonnancement conjoint les plus basiques comme le serveur BG, et que d'autre part, il présente des performances similaires (voire légèrement supérieures sous certaines conditions), vis-à-vis de celles observées avec l'algorithme TB*. Pour chacun des serveurs expérimentés, à savoir BG, TBS, TB(2), TB* et EDL, les performances ont été évaluées avec l'ordonnanceur RTO et l'ordonnanceur BWP. Etant donné que les résultats de performance obtenus avec RTO sont très proches de ceux obtenus avec BWP, nous ne présenterons ici que ceux relatifs aux modèles d'ordonnancement basés sur BWP.

L'évaluation a été effectuée en deux phases. Nous avons tout d'abord comparé les performances des différents modèles d'ordonnancement en supposant que toutes les tâches apériodiques s'exécutent exactement selon leur durée d'exécution au pire-cas (WCET). Puis, nous avons considéré un cadre plus proche de la réalité dans lequel les tâches apériodiques s'exécutent selon une durée d'exécution effective (ACET) inférieure à leur durée d'exécution au pire-cas.

5.3.1.1 Exécutions sur la base du WCET Les mesures reposent sur l'évaluation du temps séparant l'occurrence d'une requête apériodique non critique de sa fin d'exécution. L'évaluation est menée en fonction de la charge périodique U_p appliquée au système, pour une charge apériodique non critique $U_{ap} = 40\%$ constante.

Les résultats obtenus pour $s_i = 3$ (une instance sur trois peut être abandonnée) et $s_i = 10$ (une instance sur dix peut être abandonnée) sont décrits sur les figures IV.24 et IV.25 respectivement.

Les courbes nous montrent que le serveur EDL et le serveur TB* offrent de meilleures performance que le serveur BG. De plus, notons que cet avantage est d'autant plus significatif que la charge périodique U_p appliquée au système est importante. Comme nous pouvons le voir, la plus grande différence de performance entre les serveurs BG et EDL apparaît pour des charges périodiques élevées. A titre illustratif, lorsque $s_i = 3$ (cf. Figure IV.24), les serveurs EDL et TB* servent les tâches apériodiques en moyenne une fois et demi plus vite que le serveur BG lorsque $U_p \geq 70\%$.

Les deux serveurs optimaux EDL et TB* fournissent les mêmes performances. Notons également que la performance obtenue par TB(2) se situe entre celle du serveur BG et celles des serveurs EDL et TB* tandis que le serveur TBS offre des temps de réponse se rapprochant de ceux obtenus sous BG.

Par ailleurs, pour une charge périodique élevée, les variations du paramètre s_i nous montre que le temps de réponse est d'autant plus faible que le paramètre de QoS appliqué au niveau des tâches périodiques est grand. Par exemple, si l'on considère la performance de EDL et TB* pour $U_p = 85\%$, nous observons que le temps de réponse moyen des tâches apériodiques pour des tâches ayant un paramètre de pertes égal à 3 (cf. Figure IV.24) subit une augmentation d'un facteur 5 lorque les tâches périodiques possèdent un paramètre de pertes égal à 10 (cf. Figure IV.25).

Enfin, lorsque le facteur total d'utilisation du processeur $U_p^* + U_{ap}$ est proche de 100% (cf. Figure IV.25 lorsque $U_p = 105\%$), l'ensemble des algorithmes offrent des performances médiocres.

Notons cependant que pour des charges périodiques élevées, la performance optimale du serveur TB* est obtenue au prix d'un grand nombre d'étapes de raccourcissement de l'échéance. En effet, la largeur de bande $U_s = 1 - U_p^*$ est d'autant plus petite que la charge périodique U_p^* est élevée, ce qui entraîne que l'échéance à l'étape 1 calculée selon TBS et donnée par $d_k = d_k = \max(r_k, d_{k-1}) + \frac{C_k^a}{U_s}$ peut être très grande et très éloignée de l'échéance optimale.

La complexité du serveur EDL quant à elle, ne dépend pas de la charge périodique appliquée au système, ce qui constitue un avantage non négligeable de cette dernière approche.

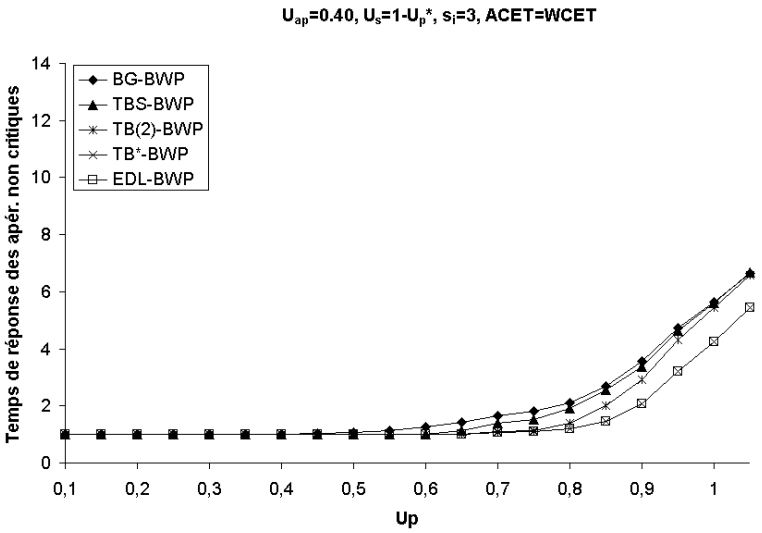

FIG. IV.24 – Temps de réponse moyen en fonction de U_p ($s_i = 2$)

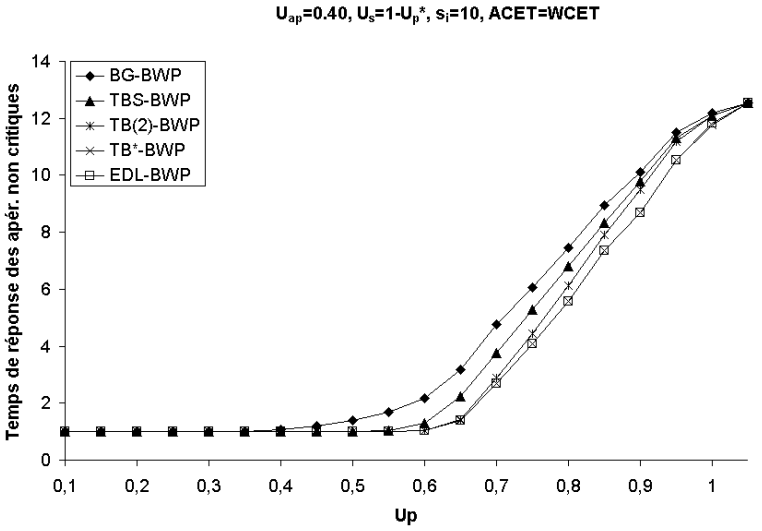

FIG. IV.25 – Temps de réponse moyen en fonction de U_p ($s_i = 6$)

5.3.1.2 Exécutions sur la base du ACET Nous nous plaçons à présent dans le cas où les tâches apériodiques s'exécutent non plus sur une durée égale à leur WCET mais sur une durée effective, appelée ACET, inférieure à leur durée d'exécution au pire-cas. Cette durée est définie lors de la phase d'initialisation et est telle que $ACET = k.WCET$ avec $0 < k < 1$. Nous l'avons volontairement considérée identique pour toutes les tâches apériodiques occurrentes, de manière à souligner l'impact de ce paramètre sur les performances des modèles d'ordonnancement.

Nous rapportons ci-dessous les résultats de l'évaluation du temps de réponse des tâches apériodiques non critiques en fonction de la charge périodique U_p appliquée au système. Comme précédemment, les mesures ont été effectuées pour des paramètres de pertes distincts ($s_i = 3$ et $s_i = 10$). Pour chaque configuration de pertes, nous présentons les résultats obtenus (cf. figures IV.26, IV.27, IV.28, IV.29) avec des durées d'exécution de 90% et 75% vis-à-vis du WCET.

Nous observons qu'à charge périodique équivalente, le temps de réponse observé ici est meilleur que dans le cas où les tâches apériodiques s'exécutent selon leur WCET. Ceci s'explique par le fait que la quantité de temps WCET-ACET non utilisé par une requête apériodique à un instant donné, représente du temps processeur additionnel utilisé pour avancer l'exécution des autres requêtes présentes à ce même instant.

Par ailleurs, il est intéressant de constater que le serveur EDL offre de meilleures performances que le serveur TB* qui n'est optimal du point de vue de la minimisation du temps de réponse des requêtes apériodiques, *uniquement* sous l'hypothèse d'une exécution des requêtes selon leur WCET, comme nous l'avions souligné dans le Chapitre I (cf. Section 3.3.3). C'est pourquoi nous remarquons ici que le temps de réponse est meilleur avec le serveur EDL et ce, quel que soit le paramètre de pertes autorisées. A titre illustratif, nous observons que pour une durée d'exécution effective ACET=0.90*WCET, le gain de performance de EDL vis-à-vis de TB* est constant et en moyenne de 7% lorsque la charge périodique appliquée est telle que $U_p \geq 85\%$ pour $s_i = 3$ (cf. Figure IV.26). Dès lors que l'on considère les courbes pour lesquelles la durée d'exécution effective des tâches apériodiques est telle que ACET=0.75*WCET (cf. Figures IV.28 et IV.29), le gain de performance atteint est de l'ordre de 15%, soit plus du double de celui observé dans le cas où ACET=0.90*WCET.

En résumé, nous pouvons dire que l'influence de la durée d'exécution réelle de la requête apériodique sur les performances des serveurs étudiés est significative. La prévalence de EDL sur TB* est d'autant plus importante que les tâches s'accomplissent avec des durées d'exécution faibles par rapport à leur WCET. Ainsi, nous pouvons affirmer que l'un des avantages du serveur EDL sur le serveur TB* réside dans le fait que les temps de réponse sont optimaux, même si le temps d'exécution effectif des tâches n'est pas égal au temps d'exécution au pire-cas.

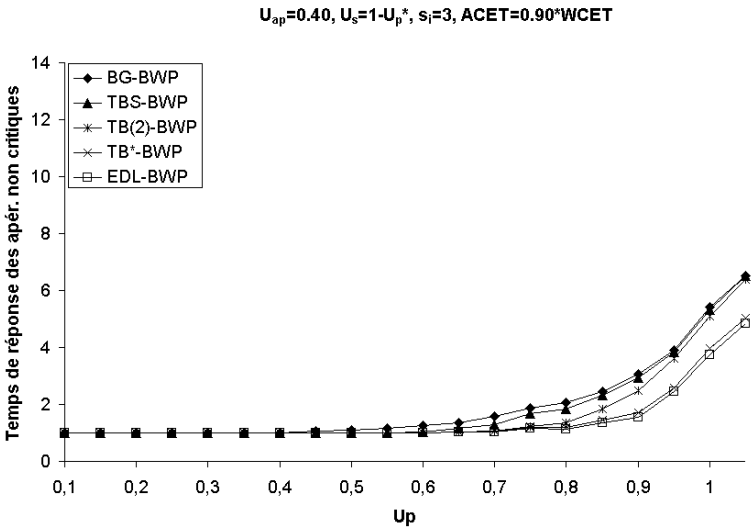

FIG. IV.26 – Temps de réponse moyen en fonction de U_p ($s_i = 3$, ACET=0.90 WCET)

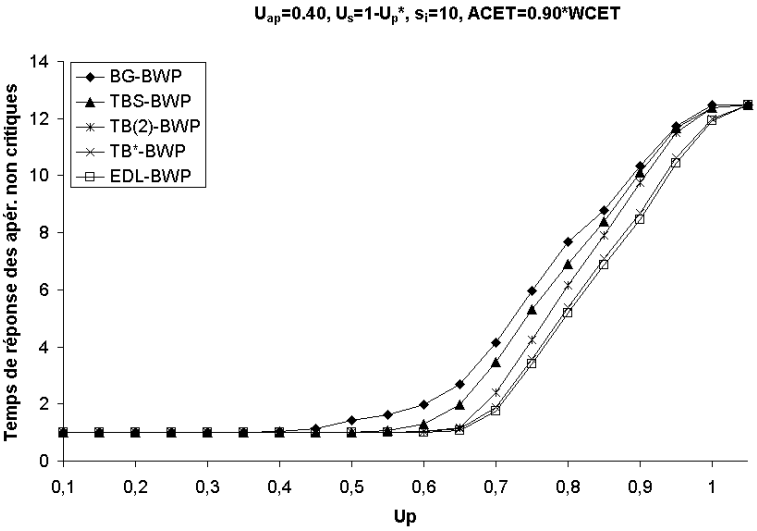

FIG. IV.27 – Temps de réponse moyen en fonction de U_p ($s_i = 10$, ACET=0.90 WCET)

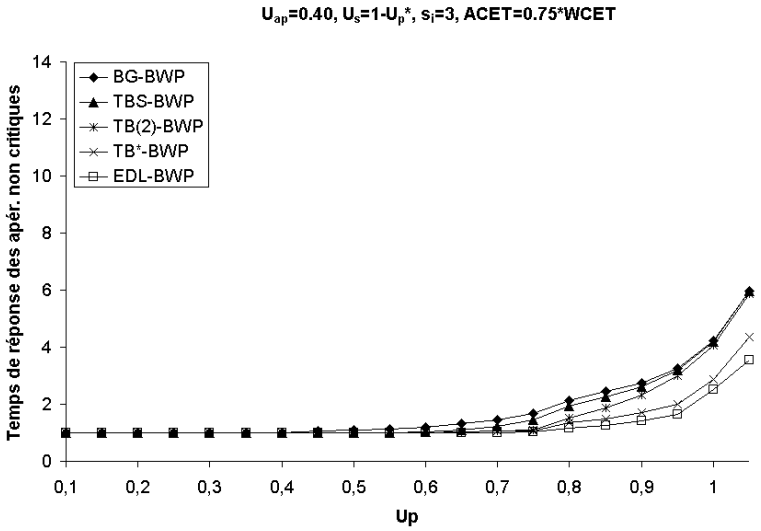

FIG. IV.28 – Temps de réponse moyen en fonction de U_p ($s_i = 3$, ACET=0.75 WCET)

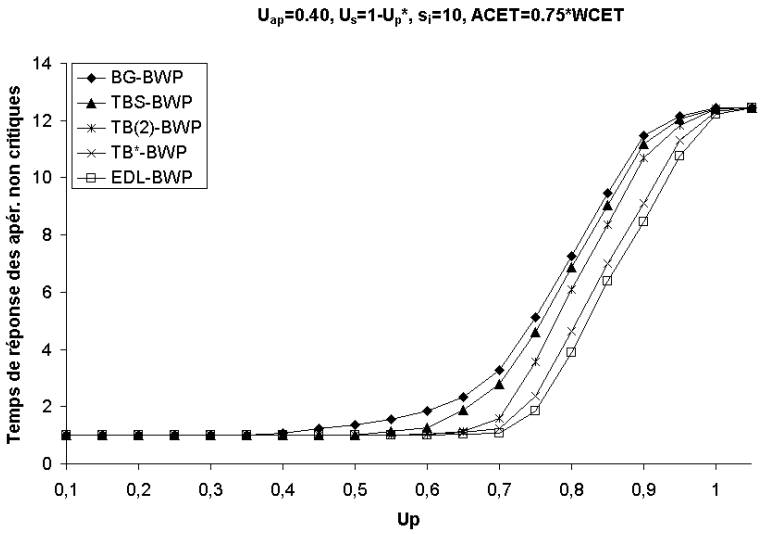

FIG. IV.29 – Temps de réponse moyen en fonction de U_p ($s_i = 10$, ACET=0.75 WCET)

5.3.2 Evaluation du taux d'acceptation des tâches apériodiques critiques

Nous examinons à présent la performance des modèles d'ordonnancement BG-BWP, TBS-BWP, TB(2)-BWP, TB*-BWP et EDL-BWP au niveau du taux d'acceptation moyen des requêtes apériodiques critiques. L'évaluation est effectuée pour une charge périodique appliquée au système variant de 10 à 105% tandis que la charge apériodique critique reste constante à un taux égal à $U_{apc} = 40\%$. Les résultats obtenus pour $s_i = 3$ et $s_i = 10$ sont décrits sur les figures IV.30 et IV.31 respectivement.

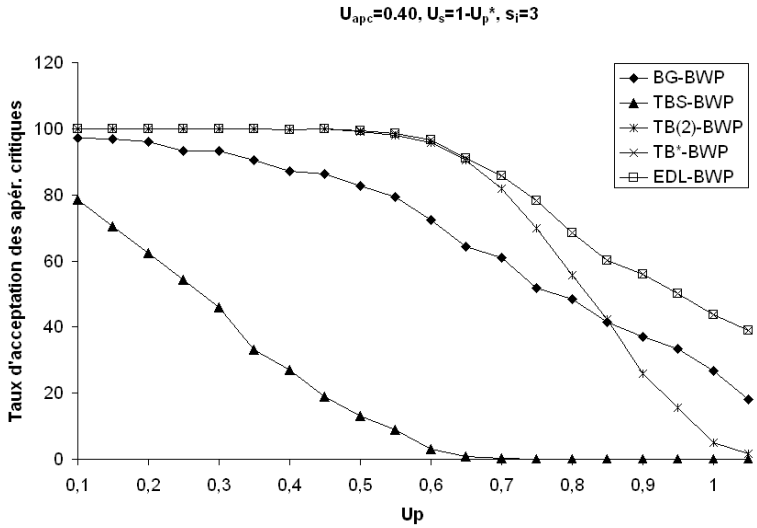

FIG. IV.30 – Taux d'acceptation moyen en fonction de U_p ($s_i = 3$)

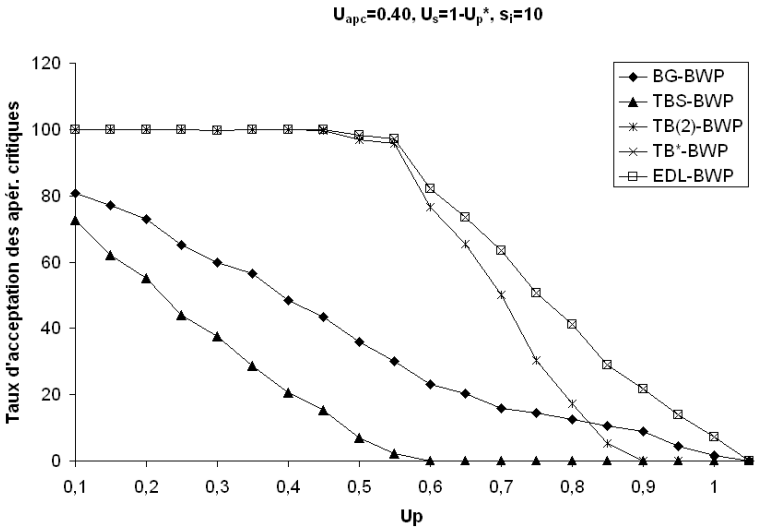

FIG. IV.31 – Taux d'acceptation moyen en fonction de U_p ($s_i = 10$)

Dans l'ensemble, nous observons que les performances des serveurs EDL, TBS, TB(2), TB* et BG sont identiques pour de faibles charges périodiques ($U_p \leq 60\%$). Pour $U_p > 60\%$, nous pouvons voir que les serveurs EDL et TB* fournissent les performances les meilleures avec plus de 20% de requêtes supplémentaires acceptées vis-à-vis du serveur BG lorsque $s_i = 3$, et jusqu'à plus de 60% de plus lorsque $s_i = 10$. Toutes charges confondues, TBS accepte peu de requêtes comparé aux autres modèles. Ceci est dû au fait que le serveur TBS n'a pas été conçu initialement pour gérer des tâches apériodiques critiques. L'avantage de performance de EDL et TB* sur TB(2) est d'autant plus significatif que la charge périodique U_p est élevée. De plus, nous pouvons noter que les serveurs EDL et TB* offrent des résultats de performance identiques, au sens où ils acceptent autant de requêtes apériodiques critiques l'un comme l'autre. Le serveur TB(2) fournit de meilleurs résultats que le serveur BG uniquement jusqu'à la valeur $U_p = 85\%$. Au-delà de cette charge, le serveur BG semble plus intéressant à utiliser. Enfin, nous remarquons que le nombre de requêtes apériodiques critiques acceptées est plus important lorsque les violations d'échéances tolérées sont importantes (petite valeur de s_i). Par exemple, pour $U_p = 90\%$, les serveurs EDL et TB* appliqués à des tâches périodiques telles que $s_i = 3$, accepteront deux fois plus de requêtes apériodiques critiques qu'avec des tâches périodiques possédant un paramètre $s_i = 10$.

Puisque nous venons de souligner l'optimalité et la supériorité des performances du serveur EDL vis-à-vis des autres serveurs de tâches apériodiques, nous nous focalisons dans la suite sur l'évaluation des modèles d'ordonnancement basés sur le serveur EDL, à savoir EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/T.

5.4 Evaluation de l'influence de l'ordonnanceur avec QoS utilisé

Le but poursuivi dans le cadre de ces simulations est d'étudier l'influence du choix de l'ordonnanceur avec contraintes de QoS, sur les performances du système. Les résultats de mesures rapportés ici portent sur l'évaluation du temps de réponse des tâches apériodiques non critiques, du taux d'acceptation des tâches apériodiques critiques, mais également sur celle du taux de respect des tâches périodiques.

5.4.1 Evaluation du temps de réponse des tâches apériodiques non critiques

Nous présentons en premier lieu les résultats de simulation obtenus au niveau du temps de réponse moyen des tâches apériodiques non critiques, en fonction de l'ordonnanceur de tâches périodiques utilisé. Les mesures ont été effectuées en supposant que toutes les tâches apériodiques s'exécutent exactement selon leur durée d'exécution au pire-cas (WCET). Rappelons également que la détermination du temps de réponse des requêtes apériodiques repose sur l'évaluation du temps séparant l'occurrence de la requête apériodique de sa fin d'exécution. Les simulations ont été menées en fonction de la charge périodique U_p appliquée au système, pour une charge apériodique non critique $U_{ap} = 40\%$ constante. Les résultats obtenus pour $s_i = 3$ et $s_i = 10$ sont décrits sur les figures IV.32 et IV.33 respectivement.

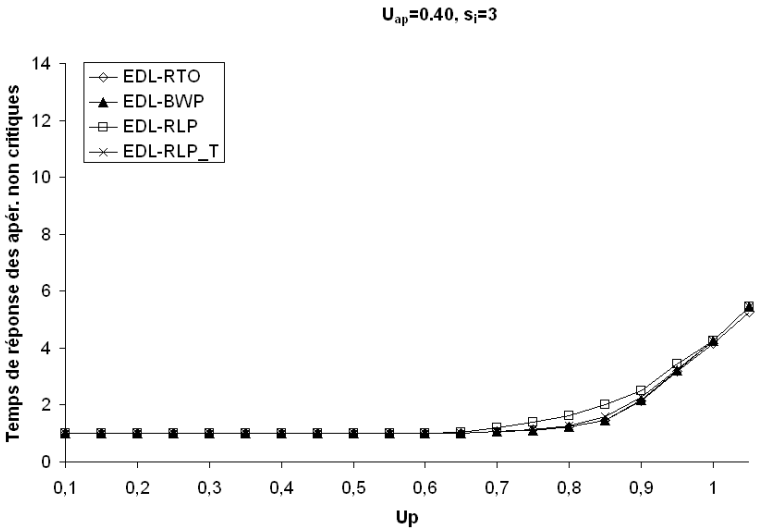

FIG. IV.32 – Temps de réponse moyen en fonction de U_p ($s_i = 3$)

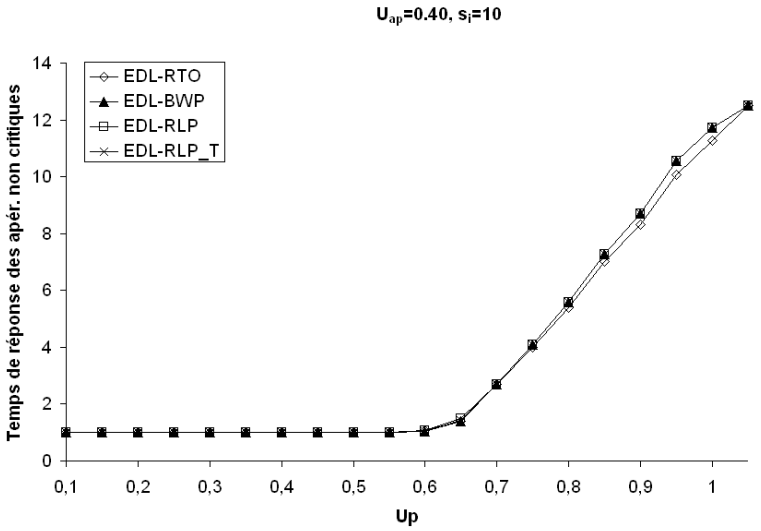

FIG. IV.33 – Temps de réponse moyen en fonction de U_p ($s_i = 10$)

Les courbes nous montrent que les 4 modèles d'ordonnancement offrent des performances très proches les unes des autres. Notons que EDL-RLP apparaît comme le modèle le moins performant dès que $U_p \geq 70\%$, lorsque les pertes autorisées au niveau des tâches périodiques sont importantes. A titre illustratif, remarquons que le temps de réponse observé sous EDL-RLP est majoré de plus de 30% pour $U_p = 85\%$, par rapport aux autres modèles (cf. figure IV.32). En revanche, lorsque les pertes autorisées sont faibles, c'est le modèle EDL-RTO qui se démarque des autres en offrant le temps de réponse le plus faible dès que $U_p \geq 80\%$. Cependant la plus grande différence de performance entre EDL-RTO et les autres modèles est seulement de 5% et est atteinte pour $U_p = 95\%$ (cf. figure IV.33).

En résumé, nous pouvons dire que globalement, les performances des différents modèles d'ordonnancement sous contraintes de QoS basés sur le serveur EDL, offrent des performances similaires en termes de temps de réponse aux requêtes apériodiques non critiques et ce, quels que soient la charge périodique du système et les pertes autorisées au niveau des tâches périodiques.

5.4.2 Evaluation du taux d'acceptation des tâches apériodiques critiques

Nous examinons à présent la performance des modèles d'ordonnancement EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/T au niveau du taux d'acceptation moyen des requêtes apériodiques critiques. L'évaluation est effectuée pour une charge périodique appliquée au système variant de 10 à 105% tandis que la charge apériodique critique reste constante à un taux égal à $U_{apc} = 40\%$. Les résultats obtenus pour $s_i = 3$ et $s_i = 10$ sont présentés sur les figures IV.34 et IV.35 respectivement.

Dans l'ensemble, nous observons que les performances des modèles sont quasiment les mêmes pour de faibles charges périodiques ($U_p < 50\%$). Au-delà, il apparaît nettement que le modèle EDL-RLP est moins bon que le reste des modèles étudiés. En effet, dans le cas où $s_i = 3$, pour $U_p \geq 60\%$, EDL-RLP accepte en moyenne 15% de requêtes apériodiques en moins (cf. figure IV.34). Pour des pertes plus faibles, cet écart de performance tend à diminuer pour des charges périodiques élevées (cf. figure IV.35).

En ce qui concerne les 3 autres modèles, ceux-ci présentent des performances identiques lorsque les pertes autorisées au niveau des tâches périodiques sont peu importantes. En revanche, lorsque le paramètre de pertes s_i est plus petit, des différences de performances apparaissent. Dans le cas représenté ici où $s_i = 3$, EDL-RLP/T est le modèle le plus performant jusqu'à la charge $U_p = 70\%$. Pour $U_p > 70\%$, c'est le modèle EDL-BWP qui accepte le plus grand nombre de requêtes apériodiques, avec un écart maximum de seulement 4% par rapport à la performance obtenue sous EDL-RLP/T.

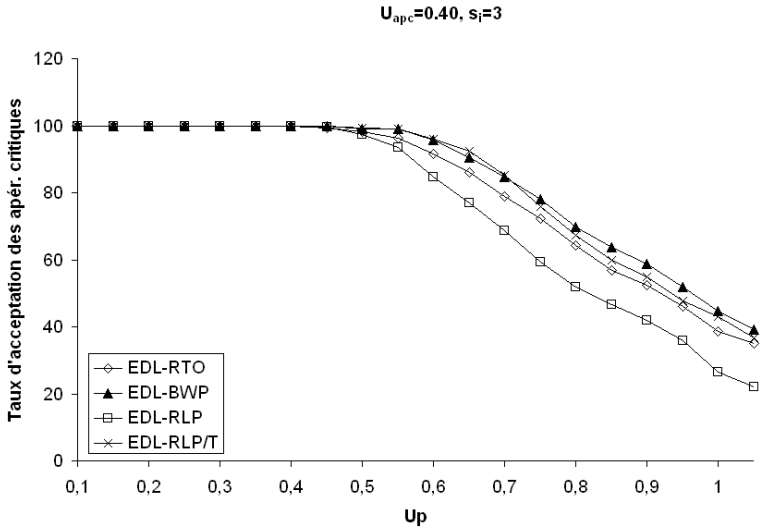

FIG. IV.34 – Taux d'acceptation moyen en fonction de U_p ($s_i = 3$)

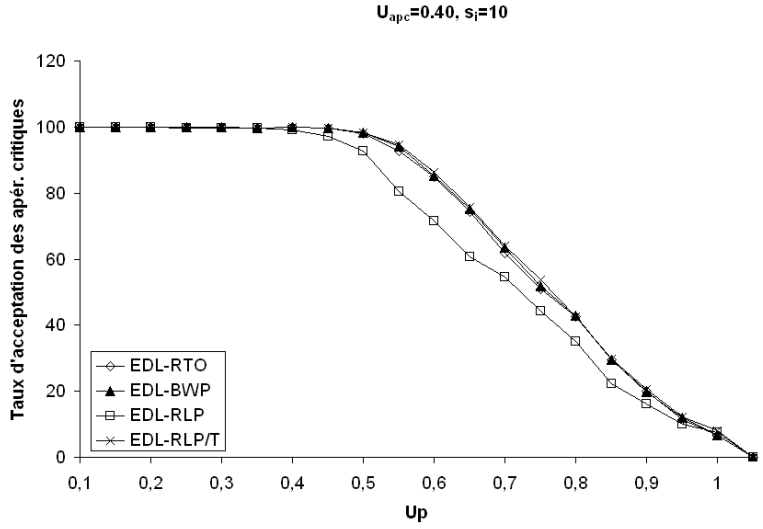

FIG. IV.35 – Taux d'acceptation moyen en fonction de U_p ($s_i = 10$)

5.4.3 Evaluation du taux de respect des tâches périodiques

Etant donné que les modèles EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/T offrent quasiment les mêmes performances en termes de temps de réponse aux requêtes apériodiques non critiques et en termes de taux d'acceptation des requêtes apériodiques critiques, nous nous sommes intéressés à l'évaluation du taux de respect des tâches périodiques pour ces quatre modèles.

L'évaluation a été menée sous deux aspects. Nous avons tout d'abord comparé les performances des différents modèles d'ordonnancement entre eux, en présence d'activité apériodique en mesurant la QoS des tâches périodiques. Puis, nous avons considéré les modèles d'ordonnancement de façon individuelle, de manière à évaluer l'influence de la charge apériodique sur leurs performances relatives.

5.4.3.1 Variation de la charge périodique. Les mesures reposent sur la fraction de requêtes périodiques qui s'accomplissent dans le respect de leur échéance. L'évaluation est menée en fonction de la charge périodique U_p appliquée au système, pour une charge apériodique non critique constante, égale à 40%. Les figures IV.36 et IV.37 illustrent les résultats obtenus pour $s_i = 3$ et $s_i = 10$ respectivement.

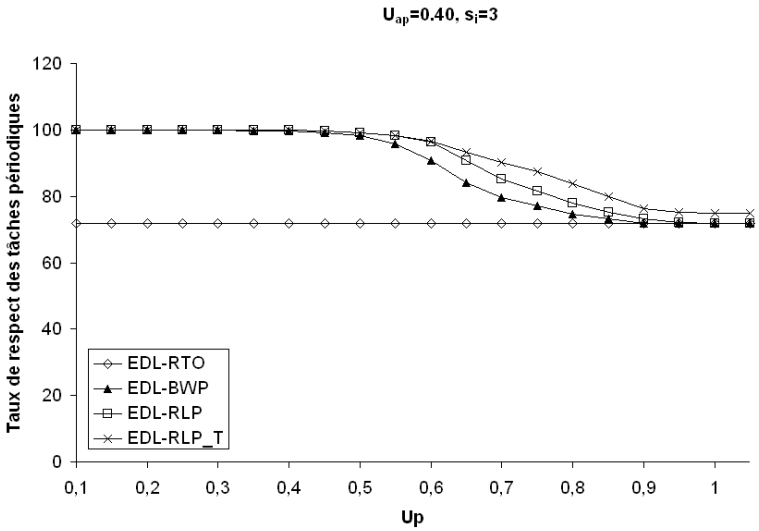

FIG. IV.36 – Taux de respect en fonction de U_p ($s_i = 3$)

Les courbes nous montrent que les modèles EDL-BWP, EDL-RLP et EDL-RLP/T offrent de meilleures performances que le modèle EDL-RTO pour lequel le niveau de QoS offert est toujours le même quelle que soit la charge périodique appliquée. Pour $s_i = 3$, la QoS reste ainsi constante à un taux de $3/4 = 75\%$.

En ce qui concerne les modèles EDL-BWP, EDL-RLP et EDL-RLP/T, ceux-ci offrent des performances quasiment identiques pour de faibles pertes autorisées. Dans le cas contraire cependant, lorsque les pertes tolérées par les tâches sont plus importantes,

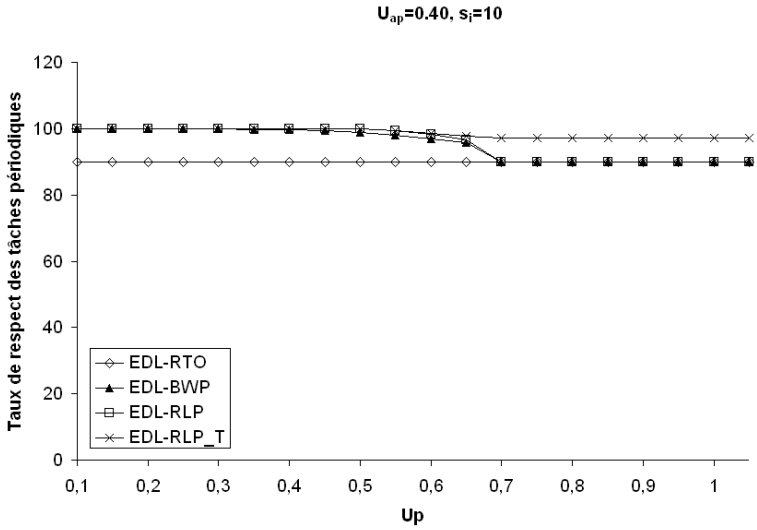

FIG. IV.37 – Taux de respect en fonction de U_p ($s_i = 10$)

leurs performances se distinguent les unes des autres. La meilleure performance est obtenue avec le modèle EDL-RLP/T tandis que EDL-BWP offre les moins bons résultats parmi les 3 modèles ; EDL-RLP se situant dans une performance intermédiaire. Illustrons numériquement notre propos : pour $U_p = 85\%$ par exemple, les taux de respect des tâches périodiques observés sous EDL-BWP, EDL-RLP et EDL-RLP/T, sont respectivement égaux à 77%, 81%, et 87%.

5.4.3.2 Variation de la charge périodique et apériodique. Les mesures effectuées ici visent à évaluer l'impact de la charge apériodique sur le taux de respect des requêtes périodiques. L'évaluation a été faite pour une charge périodique appliquée au système, qui varie de 10% à 105%, tandis que la charge apériodique non critique reste constante. Quatre séries ont été simulées pour lesquelles $U_{ap} = 20\%$, $U_{ap} = 40\%$, $U_{ap} = 60\%$ et $U_{ap} = 80\%$. Les résultats obtenus pour EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/T, pour $s_i = 2$, sont décrits sur les figures IV.38, IV.39, IV.40 et IV.41 respectivement.

Les résultats de performance montrent que la performance du modèle EDL-RTO est nettement moins bonne que celle obtenue sous les autres modèles. Ce résultat était attendu dans la mesure où l'algorithme d'ordonnancement RTO ne tente jamais l'exécution des instances de tâches bleues. Néanmoins, rappelons que le modèle EDL-RTO offre un solution efficace en termes de complexité calculatoire au problème de la détermination des temps creux. Ceci s'assortit en contrepartie d'une QoS constante au niveau des tâches périodiques, dépendant directement du paramètre s_i (ici dans le cas où $s_i = 2$, nous observons que seulement 50% des instances périodiques respectent leurs échéances).

Les modèles EDL-BWP, EDL-RLP et EDL-RLP/T en revanche, affichent un overhead d'ordonnancement pour la gestion des tâches apériodiques plus important, mais

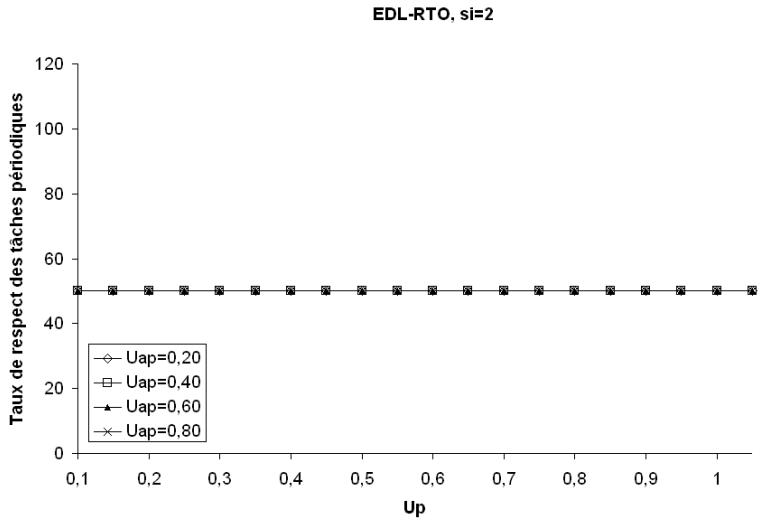

FIG. IV.38 – Taux de respect sous EDL-RTO en fonction de U_p ($s_i = 2$)

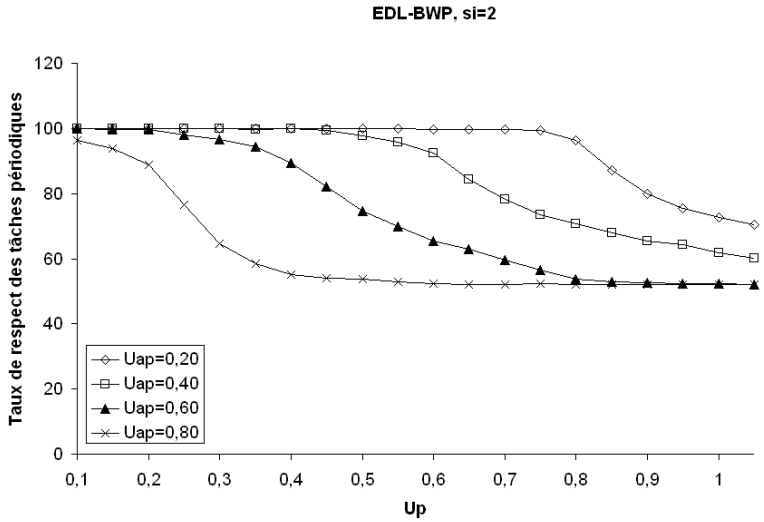

FIG. IV.39 – Taux de respect sous EDL-BWP en fonction de U_p ($s_i = 2$)

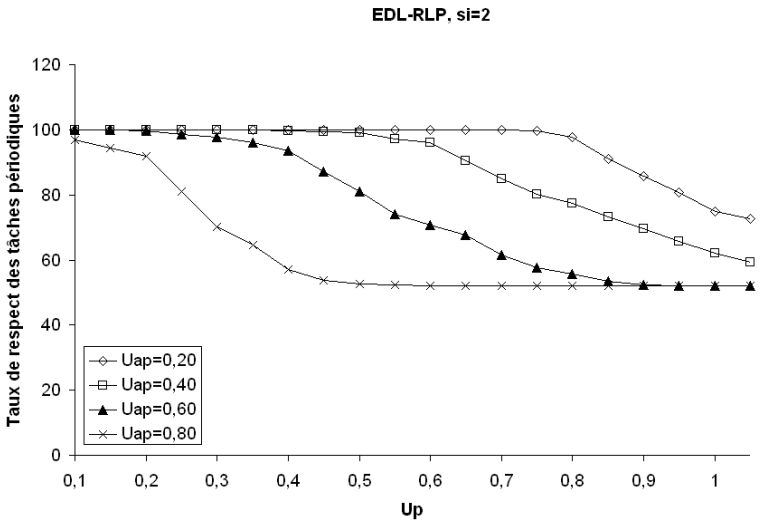

FIG. IV.40 – Taux de respect sous EDL-RLP en fonction de U_p ($s_i = 2$)

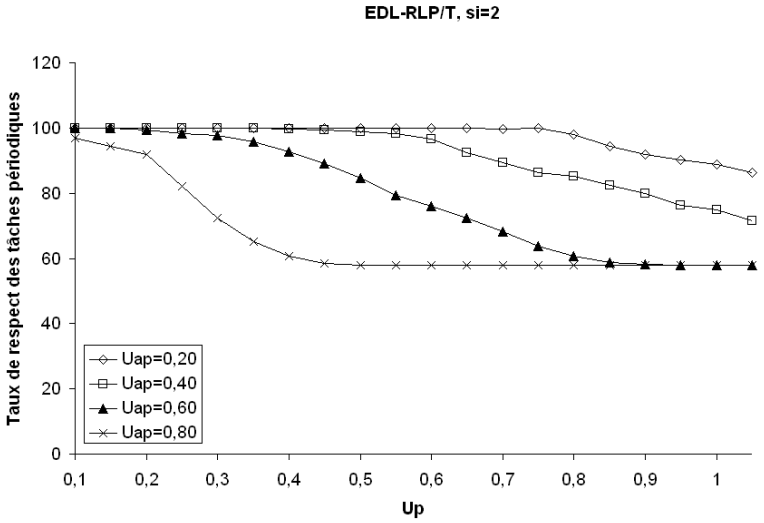

FIG. IV.41 – Taux de respect sous EDL-RLP/T en fonction de U_p ($s_i = 2$)

offrent de meilleures performances. Par exemple, pour $U_{ap} = 20\%$, le taux de respect des tâches périodiques sous EDL-RLP/T est double de celui obtenu sous EDL-RTO, tant que $U_p \leq 75\%$.

Par ailleurs, le taux de respect des tâches périodiques est d'autant plus élevé que la charge apériodique du système est faible. Ceci s'explique par le fait que les requêtes apériodiques viennent s'exécuter en priorité vis-à-vis des instances de tâches bleues, réduisant ainsi la QoS au niveau des tâches périodiques. Pour une charge totale du système $U_p + U_{ap} > 140\%$, tous les modèles tendent à avoir le même comportement. Notons enfin que l'avantage de EDL-RLP/T vis-à-vis des autres modèles est d'autant plus important que le taux de violations d'échéances tolérées au niveau des tâches périodiques est important (petite valeur de s_i).

6 Synthèse

Nous présentons brièvement dans cette partie une synthèse d'une part de la performance des différents serveurs de tâches apériodiques, et d'autre part de celle des modèles d'ordonnancement basés EDL.

6.1 Synthèse des performances des serveurs de tâches apériodiques

Les simulations menées précédemment au niveau des serveurs BG, TBS, TB* et EDL, ont permis d'évaluer leurs performances respectives en termes du service assuré pour les tâches apériodiques. Le but poursuivi était la minimisation du temps de réponse des tâches apériodiques non critiques et la maximisation du taux d'acceptation des tâches apériodiques critiques.

L'interprétation des courbes obtenues et l'étude du fonctionnement de ces différents serveurs de tâches apériodiques nous a conduit aux résultats synthétisés dans le tableau suivant (cf. IV.4). Les '★' traduisent les bonnes performances réalisées par les serveurs vis-à-vis de chacun des critères présentés.

Serveurs	Performance service apér.	Complexité algorithmique	Besoins mémoire	Complexité d'implémentation
BG	★	★★★★	★★★★	★★★★
TBS	★★	★★★	★★★	★★★
*TB**	★★★	★★	★★	★★
EDL	★★★★	★	★	★

TAB. IV.4 – Synthèse des performances des serveurs de tâches apériodiques

Les algorithmes diffèrent de par leur performance et leurs complexités. Remarquons que le serveur BG qui présente des performances médiocres, affiche des temps de calcul relativement faibles. En revanche, le gain de performance observé au niveau des serveurs TB* et EDL s'assortit d'une augmentation de la complexité calculatoire et de la taille des structures de données liées à l'implémentation.

En résumé, les résultats présentés soulignent le fait que les deux critères que sont la performance et la complexité sont orthogonaux. Par conséquent, nous pouvons dire que le choix d'un serveur donné, relève des contraintes imposées par à la fois par l'application et par son support d'exécution.

6.2 Synthèse des performances des modèles basés sur EDL

Nous nous proposons à présent de synthétiser les résultats obtenus pour les modèles d'ordonnancement d'ensembles hybrides de tâches basés sur le serveur optimal EDL, à savoir les modèles EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/P. Le bilan des performances offertes pour chaque type de tâches, apparaît ci-dessous dans la Table IV.5.

Rappelons que l'optimisation des performances porte sur :

- la minimisation du temps de réponse des tâches apériodiques non critiques,
- la maximisation du taux d'acceptation des tâches apériodiques critiques,
- la maximisation de la QoS des tâches périodiques.

Modèle d'ordonnancement	Performance apér. non critiques	Performance apér. critiques	Performance périodiques
EDL-RTO	★★★★	★★★	★
EDL-BWP	★★★	★★★★	★★
EDL-RLP	★★	★★	★★★
EDL-RLP/T	★★★	★★★★	★★★★

Tab. IV.5 – Synthèse des performances des serveurs de tâches apériodiques

Ce tableau récapitulatif montre que les différents modèles d'ordonnancement diffèrent légèrement quant à leurs performances vis-à-vis de la gestion des requêtes apériodiques. En revanche, nous pouvons remarquer l'écart significatif de performances relatif à la QoS offerte au niveau des tâches périodiques.

Il apparaît donc que le choix du modèle d'ordonnancement doit s'effectuer selon les besoins de l'application. Celui-ci dépend à la fois de la QoS souhaitée au niveau des tâches périodiques et de l'importance accordée à la gestion des requêtes apériodiques, celles-ci étant dictées par les exigences de l'application.

7 Conclusion

Dans ce chapitre, nous avons décrit l'utilisation des serveurs de tâches apériodiques BG, TBS, TB* et EDL avec les ordonnanceurs Skip-Over classiques, à savoir RTO et BWP. Ensuite, nous nous sommes focalisés sur les modèles d'ordonnancement basés sur EDL et reposant sur les nouvelles stratégies d'ordonnancement RLP et RLP/T introduites dans le chapitre précédent. Puis, nous avons mené plusieurs jeux de simulation dans le but de comparer d'une part, la performance des serveurs de tâches apériodiques entre eux, et d'autre part celle des ordonnanceurs définis sous contraintes de QoS en présence d'activité apériodique. Les résultats ont permis de vérifier l'optimalité du serveur EDL au niveau de la minimisation du temps de réponse aux requêtes apériodiques non critiques et ce, quelle que soit la durée d'exécution effective des requêtes. Par ailleurs, nous avons vu que les modèles d'ordonnancement basés sur EDL offrent des performances similaires au niveau du service des requêtes apériodiques. Cependant, ils se distinguent fortement au niveau des résultats affichés en termes de QoS des tâches périodiques, avec une prédominance des performances pour le modèle EDL-RLP/T.

L'objectif du chapitre suivant va être d'évaluer la robustesse et la stabilité des algorithmes d'ordonnancement de tâches périodiques sous contraintes de QoS, et de proposer des solutions visant à apporter des améliorations sur la base des constats effectués.

Chapitre V

Gestion de la surcharge avec robustesse et stabilité

Le but de ce chapitre est d'étudier la "robustesse" et la "stabilité" des différents algorithmes d'ordonnancement sous contraintes de QoS qui ont été décrits dans le chapitre III, à savoir RTO, BWP, RLP et RLP/T. En premier lieu, nous introduisons les définitions et terminologies relatives à ces deux nouveaux critères de performance. Ensuite, une analyse des algorithmes RTO, BWP, RLP et RLP/T est conduite dans le but d'identifier les performances de chacun des ordonnanceurs vis-à-vis du critère de robustesse d'une part et de stabilité d'autre part. Puis, nous présentons deux nouvelles variantes des algorithmes, dénommées MS et LF, visant à améliorer la stabilité du système. L'objectif est alors de souligner l'impact de la politique d'ordonnancement des instances bleues sur la stabilité du système. Enfin, nous évaluons dans quelle mesure ces nouveaux algorithmes constituent une amélioration des algorithmes de base.

---

1 Introduction

Dans le chapitre II, nous avons vu que le Mécanisme à Echéance [CAM79], consiste à implanter une tâche en deux versions : une version primaire qui produit un résultat de bonne qualité mais dont on ne connait pas *a priori* la durée d'exécution, et une version secondaire qui fournit un résultat dont la QoS est dégradée mais dont le temps d'exécution maximum est connu et borné.

Une analogie entre ce modèle et le modèle Skip-Over à contraintes de QoS peut être faite. Dans les deux cas, l'objectif de ces approches est de gérer les situations de surcharge. Pour le Mécanisme à Echéance, il s'agit de commuter vers un mode de fonctionnement dégradé en produisant des résultats de moindre qualité, mais en faisant en sorte que toute échéance soit respectée. Pour le modèle Skip-Over, il s'agit également de commuter en dégradé, en produisant des résultats de qualité constante, mais en admettant que certains

résultats ne soient pas du tout produits. Les instances de tâches bleues du modèle Skip-Over peuvent être comparées aux tâches primaires du Mécanisme à Echéance, tandis que les instances de tâches rouges peuvent être associées aux tâches secondaires. Les stratégies d'ordonnancement mises en œuvre sous le modèle du Mécanisme à Echéance sont également proches de celles mises en œuvre sous le modèle Skip-Over. La stratégie d'ordonnancement de *Première Chance* qui donne une priorité supérieure aux secondaires vis-à-vis des primaires et qui ordonnance les primaires dans les temps creux du processeur après leur secondaire associé, peut être assimilée à la stratégie BWP dans laquelle les instances rouges sont toujours prioritaires vis-à-vis des instances bleues, celles-ci étant ordonnancées dans les temps creux du processeur après ordonnancement des instances rouges. La stratégie d'ordonnancement de *Seconde Chance* qui repose sur l'exécution des primaires dans les temps creux du processeur lorsque les secondaires sont exécutés au plus tard, s'apparente quant à elle à la stratégie RLP qui consiste à exécuter les instances bleues au plus tôt dans les temps creux calculés en exécutant les instances rouges au plus tard.

A partir de ces observations, nous avons mené la présente étude en nous inspirant des travaux effectués par Elyounsi et Silly dans [Ely91, Sil96] qui ont abordé le problème de la stabilité et de la robustesse des algorithmes d'ordonnancement propres au Mécanisme à Echéance.

2 Définitions et terminologie

2.1 Le critère de robustesse

La robustesse est une qualité fondamentale d'une stratégie d'ordonnancement temps réel puisqu'elle se rapporte à l'évaluation du taux de succès global de l'ensemble de tâches ordonnancé. En d'autres termes, la robustesse se réfère à la performance globale absolue du système. Considérons la définition suivante relative à la robustesse d'un système temps réel, proposée dans [Ely91, Sil96] :

Définition 10 *Un algorithme d'ordonnancement X est* plus robuste *qu'un algorithme d'ordonnancement Y si le taux de respect global des tâches avec X est supérieur au taux de respect global des tâches avec Y.*

Dans le cas de notre étude, ce n'est pas la quantification de ce critère qui s'avère le point le plus intéressant, mais bien son utilisation de manière relative dans le but de comparer plusieurs stratégies d'ordonnancement entre elles. La définition précédente peut être étendue aux systèmes définis sous contraintes de QoS pour lesquels le critère de robustesse peut être vu comme reposant sur la capacité du système à ordonnancer les instances de tâches bleues. Afin d'assurer la robustesse du système avec pertes, il faut donc que l'algorithme d'ordonnancement maximise le nombre d'instances de tâches bleues totalement exécutées avant échéance.

2.2 Le critère de stabilité

Alors que la stabilité est un concept très précis dans le cadre de la théorie du contrôle et de la commande, ce critère peut prendre des définitions diverses dans le contexte temps réel. Par exemple, Stankovic [Sta85] a défini la stabilité d'un algorithme d'ordonnancement distribué en termes d'*équilibre de charge* (c'est-à-dire qu'un système est dit stable si la charge entre deux noeuds choisis au hasard ne diffère pas de plus de $x\%$). Dans le cas présent, la stabilité est définie en termes d'équilibre du nombre de réussites des tâches périodiques, les tâches étant toutes perçues comme de même importance pour le bon comportement de l'application. Nous donnons ici la définition du critère de stabilité proposé dans ce sens dans [Ely91, Sil96] :

Définition 11 *Un algorithme d'ordonnancement X est* plus stable *qu'un algorithme d'ordonnancement Y si la plus grande différence entre les taux de respect individuels des tâches avec X est inférieure à la plus grande différence entre ces taux avec Y.*

Notons que la stabilité ne se réfère pas à l'habileté du système à maintenir un certain niveau de performance mais à la performance individuelle relative à chacune des tâches du système. Cette caractéristique ne peut constituer à elle seule une mesure des performances du système. Considérée individuellement, elle pourrait même conduire à des comportements aberrants tels que la dégradation du taux de respect global des tâches, tout en conservant toujours des taux de respect individuels équilibrés. La stabilité doit donc être accompagnée de la notion de robustesse définie précédemment.

La stabilité d'un algorithme d'ordonnancement sous contraintes de QoS est ainsi définie comme étant sa capacité à assurer l'équilibre des taux de réussite des instances bleues des tâches (notion de *fairness* [Bar95]). Par ailleurs, par analogie avec les systèmes de contrôle-commande, les systèmes temps réel connaissent généralement une période de fonctionnement dite *transitoire* durant laquelle les performances du système sont différentes de celles attendues et observées par la suite au niveau du système. Dans cette perspective, la définition de stabilité peut être précisée comme suit :

Définition 12 *Un système temps réel sera dit* stable *si au bout d'un temps fini, la proportion d'instances réussies par rapport aux requêtes engendrées est la même pour toutes les tâches, dans une fourchette ϵ prédéfinie.*

Afin d'assurer la stabilité du système, il faut donc que l'algorithme d'ordonnancement des instances de tâches bleues tienne compte, en plus des caractéristiques temporelles des tâches, de l'évolution respective des réussites individuelles des instances bleues de ces dernières.

3 Robustesse et stabilité des ordonnanceurs Skip-Over

3.1 Analyse de la robustesse

D'après les définitions précédentes et l'analyse des simulations menées dans le chapitre III (cf. Section 4.3), l'algorithme RTO apparaît comme étant un algorithme peu ro-

buste par rapport à BWP, RLP et RLP/T, puisque la QoS offerte est toujours constante quelle que soit la charge périodique du système. Pour $s_i = 2$, le niveau de QoS offert est seulement de 50% (une instance sur deux s'accomplit dans le respect de son échéance).

Le fait de tenter l'exécution des instances bleues sous BWP accroît la robustesse du modèle Skip-Over. Cependant, cette robustesse est d'autant moins marquée que le système est surchargé. Ainsi, dès de $U_p > 100\%$, le taux de respect global des tâches périodiques sous BWP diminue rapidement.

Le premier algorithme proposé dans le cadre de la thèse, à savoir l'algorithme RLP, tend à réduire la pente de cette décroissance du taux de respect global des tâches périodiques en cas de surcharge, et par conséquent contribue à améliorer la robustesse du système. Cependant, ce gain est d'autant plus faible que les pertes autorisées au niveau des tâches périodiques sont importantes.

L'algorithme le plus robuste au sens des définitions précédentes, s'avère donc être l'algorithme RLP/T qui offre un taux de respect des tâches périodiques qui décroît très lentement en fonction de la charge périodique appliquée au système et ce, quel que soit le paramètre de pertes appliqué aux tâches.

A partir de ce constat, nous pouvons dire que les travaux menés jusqu'à présent sur le modèle Skip-Over ont permis d'accroître de façon significative la robustesse d'un système défini sous des contraintes de QoS. Examinons à présent, la stabilité de l'ensemble de ces stratégies.

3.2 Analyse de la stabilité

Nous avons restreint le cadre de l'étude à des tâches qui ont toutes la même importance dans le système. L'objectif de cette analyse est d'évaluer le degré d'équilibre des exécutions des instances bleues entre les tâches.

Par définition, l'algorithme d'ordonnancement RTO est très stable puisqu'il ne tente jamais d'exécuter des instances de tâches bleues. En conséquence, les tâches affichent toujours les mêmes taux de respect individuels correspondant aux exécutions réussies des instances de tâches rouges. Rappelons cependant que cette stabilité s'assortit d'une performance globale médiocre (lorsque s_i est petit, le taux de respect global des tâches est faible).

Nous allons à présent nous focaliser sur le comportement du système sous les stratégies d'ordonnancement BWP, RLP et RLP/T. L'objectif est de souligner le manque de stabilité de l'ensemble de ces stratégies basées sur un ordonnancement des instances bleues par Earliest Deadline (ED). Pour cette raison, nous dénoterons respectivement ces stratégies BWP-ED, RLP-ED et RLP/T-ED.

Considérons l'ensemble $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$ constitué de 4 tâches dont les paramètres sont décrits dans la table V.1. Les tâches possèdent un paramètre de pertes uniforme $s_i = 2$ et le facteur d'utilisation du processeur sans pertes $U_p = \sum_{i=1}^{n} \frac{C_i}{P_i}$ est égal à 1.30. Le facteur d'utilisation équivalent du processeur intégrant les pertes (cf. Chap II. Définition 9), est quant à lui égal à $U_p^* = 0.80$. L'ensemble $\mathcal{T}$ est donc ordonnançable au sens RTO (les exécutions des instances obligatoires, à savoir les instances rouges, sont garanties).

Task	T_1	T_2	T_3	T_4
c_i	4	3	1	2
p_i	15	10	6	5

TAB. V.1 – Un ensemble basique de tâches avec pertes

L'ordonnancement de $\mathcal{T}$ par BWP-ED, RLP-ED et BWP-ED est illustré sur les figures V.1, V.2 et V.3. Pour chaque algorithme représenté, nous avons fait figurer le pourcentage de réussite individuelle des tâches sur la partie droite de la séquence d'ordonnancement des tâches.

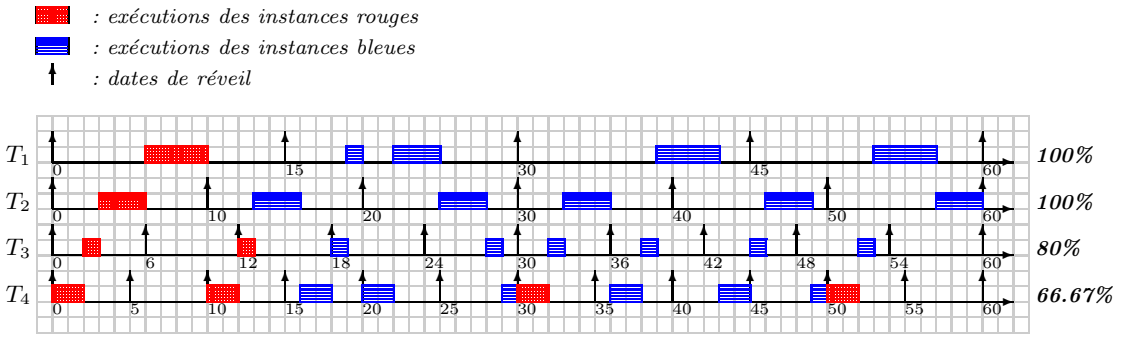

FIG. V.1 – Illustration du manque de stabilité de BWP-ED

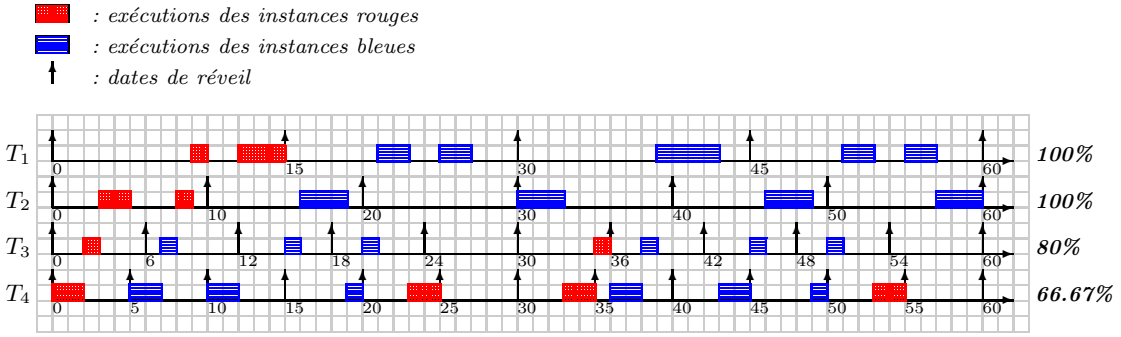

FIG. V.2 – Illustration du manque de stabilité de RLP-ED

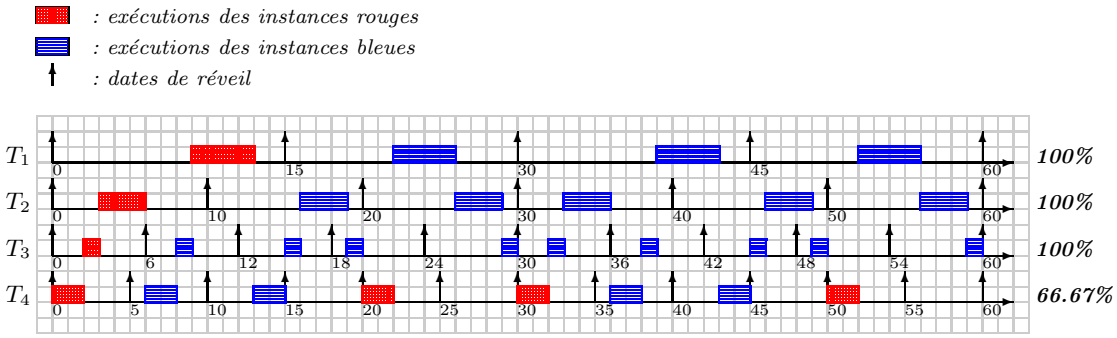

FIG. V.3 – Illustration du manque de stabilité de RLP/T-ED

L'examen des figures nous révèle que les tâches ne sont pas servies de manière équitable lorsque les tâches bleues sont ordonnancées par ED. En effet, les 3 ordonnanceurs BWP-ED, RLP-ED et RLP/T-ED affichent dans cet exemple, une différence maximale entre les pourcentages de réussites individuelles des tâches égale à 43,33%. Nous constatons que certaines tâches ne violent jamais leur échéance (ex : tâches T_1 et T_2) alors que d'autres subissent des violations d'échéances répétées (ex : tâche T_4). Ce constat nous a donc amenés à tester de nouvelles politiques d'ordonnancement des instances de tâches bleues, de manière à améliorer la stabilité.

Une quantification plus précise des différences observées entre les taux de respect des tâches périodiques sous BWP-ED, RLP-ED et RLP/T-ED, est fournie dans la dernière partie de ce chapitre.

4 Amélioration de la stabilité

Comme souligné précédemment, l'appréciation de la performance d'un ordonnanceur sous contraintes de QoS doit être faite par rapport à sa robustesse mais aussi sur le comportement de chaque tâche, qui est caractérisée par son taux de respect individuel. Comme l'analyse présentée ci-dessus nous a montré que BWP-ED, RLP-ED, et RLP/T-ED manquent de stabilité, deux nouvelles stratégies d'ordonnancement des instances de tâches bleues ont été définies dans le but d'améliorer ce critère de performance.

Les algorithmes d'ordonnancement proposés ici sont inspirés des travaux décrits dans [Ely91, Sil96], utilisant le Mécanisme à Echéance pour le recouvrement en-ligne des fautes temporelles.

4.1 Les variantes LF (Last Failure)

L'algorithme appelé *Last Failure first (LF)* revient à exécuter à tout instant l'instance de tâche bleue prête dont le nombre de succès successifs mesuré depuis le dernier échec, est le plus petit. L'échéance la plus proche est utilisée pour résoudre les conflits entre deux instances bleues de priorités identiques.

Dans ce sens, les nouveaux ordonnanceurs BWP-LF et RLP-LF octroient, à tout instant, la plus grande priorité à l'instance de tâche bleue ayant le plus petit nombre de réussites successives depuis le dernier échec. En revanche, le comportement de RLP/T-LF est légèrement différent de celui de BWP-LF et RLP-LF, dans le sens où la politique LF est appliquée à l'occurrence de l'instance de tâche bleue. Si le nouvel ensemble dans lequel la tâche bleue occurrente est incluse n'est pas ordonnançable, RLP/T-LF rejette systématiquement la ou les tâche(s) bleue(s) ayant le plus grand nombre de réussites successives depuis le dernier échec.

Considérons de nouveau la configuration de tâches $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$ (cf. Table V.1) utilisée pour illustrer le manque de stabilité de BWP-ED, RLP-ED et RLP/T-ED. L'ordonnancement de cet ensemble sous la variante LF des algorithmes BWP, RLP et RLP/T, est illustré sur les figures V.4, V.5 et V.6 respectivement. Les pourcentages de réussite individuels des tâches sont figurés à droite de chaque graphique.

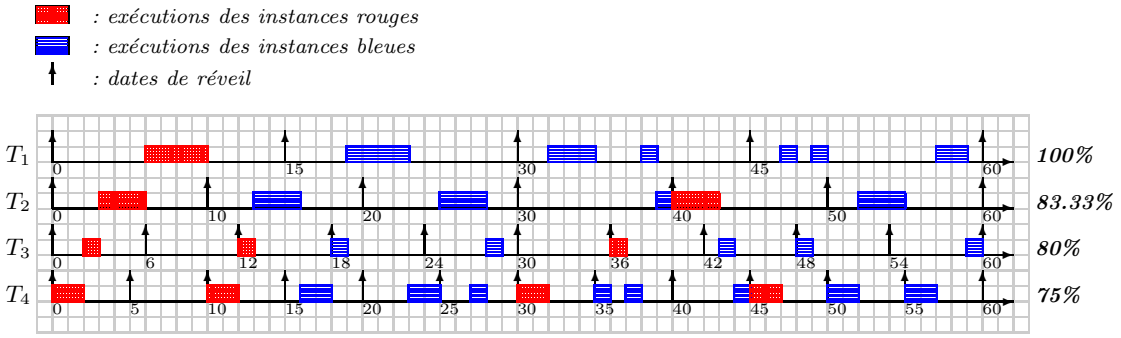

FIG. V.4 – Illustration du comportement de BWP-LF

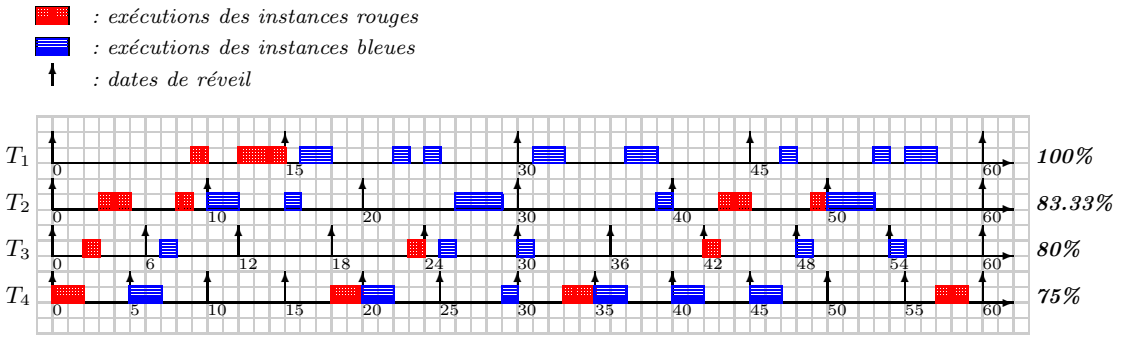

FIG. V.5 – Illustration du comportement de RLP-LF

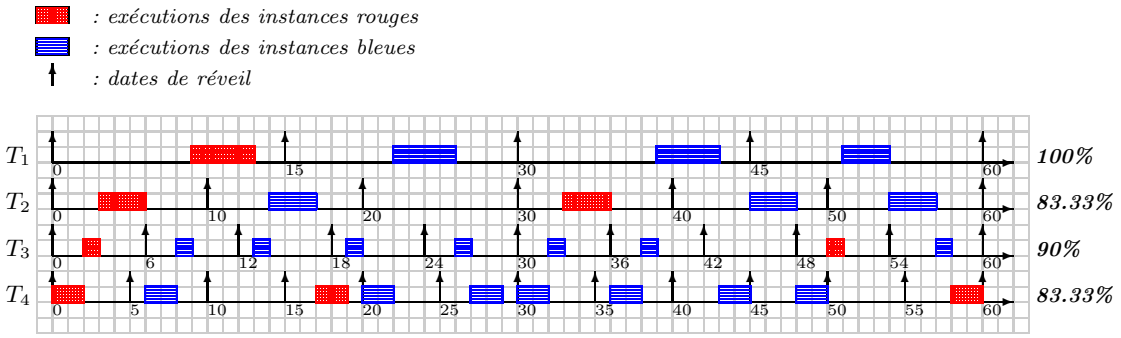

FIG. V.6 – Illustration du comportement de RLP/T-LF

L'exemple met en avant un meilleur équilibre des taux de respect individuels sous la variante LF. La différence maximale observée entre les pourcentage de réussite individuels a été ramenée à 25% pour BWP-LF et RLP-LF, et à 17.77% pour RLP/T-LF. Toutes les tâches subissent des violations d'échéances, à l'exception de la tâche T_1. Ce gain de stabilité sera quantifié de manière plus précise à la fin de ce chapitre (cf. Section 5) à travers un ensemble de simulations. Avant de les exposer plus en détails, nous présentons une seconde variante de BWP, RLP et RLP/T visant à accroître encore davantage, la stabilité des algorithmes.

4.2 Les variantes MS (Minimum Success)

L'algorithme appelé *Minimum Success first (MS)* consiste à exécuter à tout instant l'instance de tâche bleue dont le pourcentage de respect individuel calculé à partir de l'initialisation, est le plus faible. Comme dans le cas de la variante LF, les conflits de priorités sont résolus en faveur de l'instance bleue possédant l'échéance la plus proche.

Par conséquent, les nouveaux ordonnanceurs BWP-MS et RLP-MS octroient, à tout instant, la plus grande priorité à l'instance de tâche bleue ayant le plus petit pourcentage de réussite courant, calculé depuis l'initialisation. Par contre, comme dans le cas de la variante LF, le comportement de RLP/T-MS diffère de celui de BWP-MS et RLP-MS, et la politique MS est appliquée à l'occurrence de l'instance de tâche bleue. Si la tâche bleue occurrente conduit à la non-ordonnançabilité du système, RLP/T-MS va rejeter la ou les tâche(s) bleue(s) préalablement acceptées ayant le plus grand taux de respect individuel courant, calculé depuis l'initialisation et ce, pour maintenir le système en sous-charge.

Considérons toujours l'ensemble $\mathcal{T} = \{T_1, T_2, T_3, T_4\}$ précédemment décrit dans la Table V.1. L'illustration de la variante MS des algorithmes BWP, RLP et RLP/T, est présentée sur les figures V.7, V.8 et V.9 respectivement. Comme pour les versions ED et LF étudiées précédemment, les taux de respect individuels des tâches sont indiqués à droite de chaque séquence d'ordonnancement.

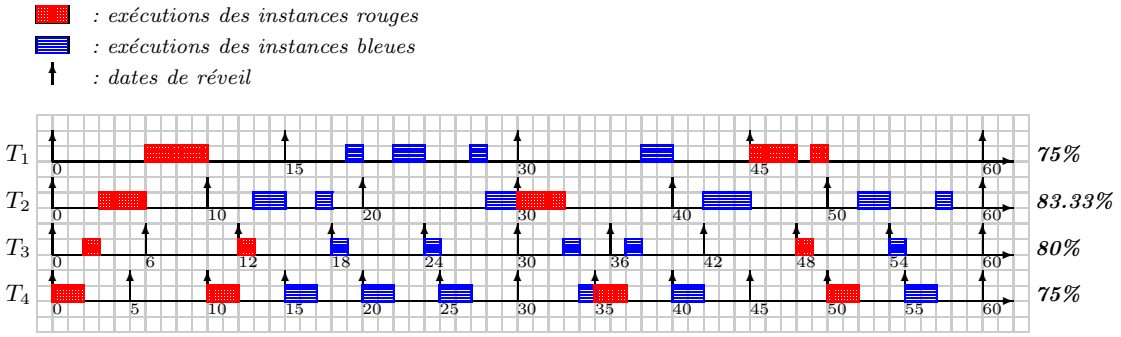

FIG. V.7 – Illustration du comportement de BWP-MS

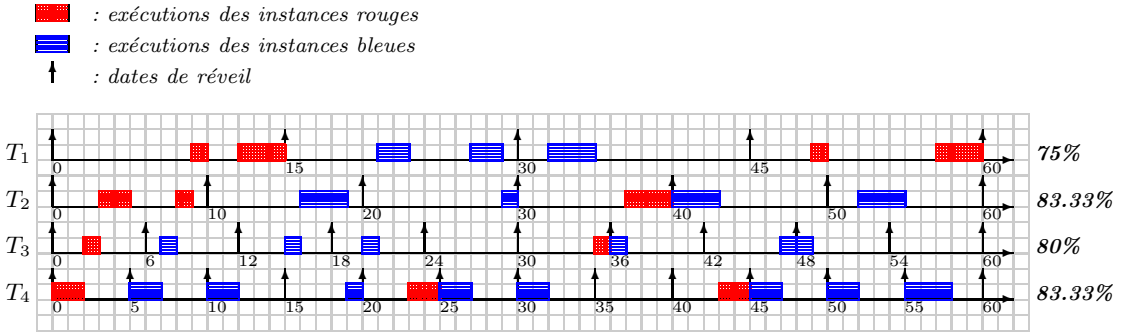

FIG. V.8 – Illustration du comportement de RLP-MS

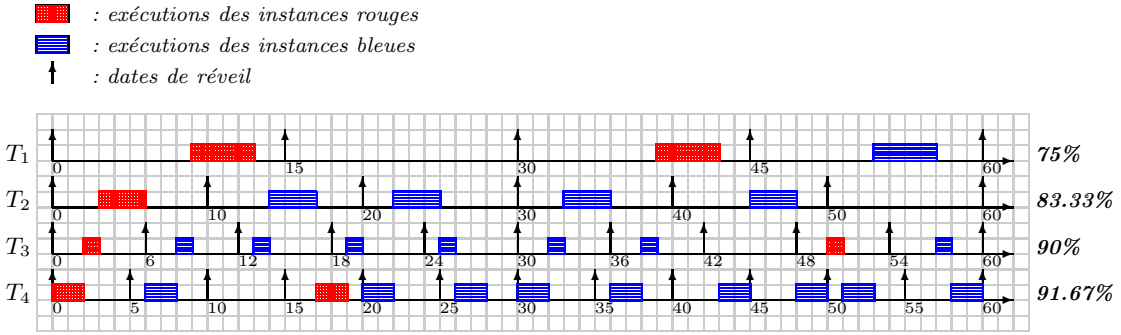

FIG. V.9 – Illustration du comportement de RLP/T-MS

L'exemple souligne un accroissement de la stabilité des ordonnanceurs. Les violations d'échéances sont réparties sur toutes les tâches, sans exception. L'étude des pourcentages de réussite individuels des tâches nous montre que leur différence maximale n'est plus que de 8.33% sous BWP-MS et RLP-MS, et de 16.67% sous RLP/T-MS.

L'objet du paragraphe suivant est de quantifier en simulation la stabilité des variantes ED, LF et MS pour l'ordonnancement des instances de tâches bleues.

5 Résultats de simulation

5.1 Environnement de simulation

Pour chaque stratégie d'ordonnancement, 50 configurations de tâches différentes ont été générées, chacune d'elles comprenant 10 tâches avec un *ppcm* des périodes égal à 3360. Les tâches possèdent un paramètre de pertes uniforme $s_i = 2$. Leur durée d'exécution au pire-cas dépend de la charge périodique U_p du système et est générée de manière aléatoire pour refléter le plus grand nombre d'applications possible. Les tâches considérées sont à échéances sur requêtes. Les simulations ont été effectuées sur 10 hyperpériodes.

5.2 Evaluation de la stabilité avec ED

Le comportement de BWP-ED, RLP-ED et RLP-ED a été étudié pour une charge périodique U_p variant de 90% à 180%. Les taux de succès individuels des tâches mesurés sous BWP-ED, RLP-ED et RLP/T-ED sont représentés sur les figures V.10, V.11 et V.12 respectivement. Notons que sur les graphiques, les tâches sont ordonnées de telle manière que $i < j$ implique $P_i < P_j$.

Les résultats de simulation nous montrent que les ordonnanceurs BWP-ED, RLP-ED et RLP/T-ED manquent de stabilité. Leur fonctionnement conduit à privilégier certaines tâches par rapport à d'autres, en particulier celles qui ont une faible période, de par l'ordonnancement des bleues par ED, et ce quelle que soit la stratégie globale d'ordonnancement, à savoir BWP, RLP ou RLP/T (cf. figures V.10, V.11 et V.12). Les taux de respect individuels observés sont "éparpillés" autour de la courbe représentant le taux de respect global des tâches. Cette constatation se vérifie aisément sur les graphiques obtenus : par exemple, dans le cas de BWP (cf. figure V.10) et pour $U_p = 140\%$, nous pouvons observer que plus de 98% des instances de la tâche T_1 réussissent à s'exécuter dans le respect de leur échéance, tandis que moins de 55% des instances de la tâche T_{10} s'accomplissent effectivement sur le même intervalle de temps.

Notons enfin que la dispersion des taux de succès individuels autour de la valeur du taux de respect global des tâches n'est pas tout à fait identique pour les 3 ordonnanceurs. En effet, sous BWP-ED et RLP/T-ED, cette dispersion est d'autant plus grande que la charge périodique appliquée au système est importante. En revanche, dans le cas de RLP-ED, cette dispersion des valeurs tend à décroître pour des charges périodiques élevées.

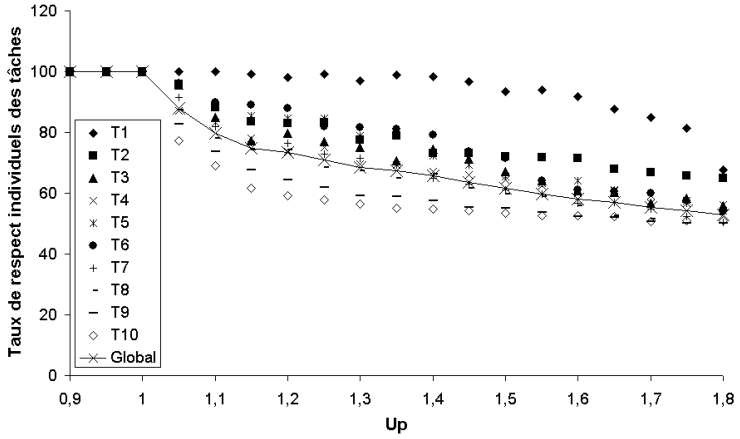

FIG. V.10 – Taux de respect individuels sous BWP-ED en fonction de U_p ($s_i = 2$)

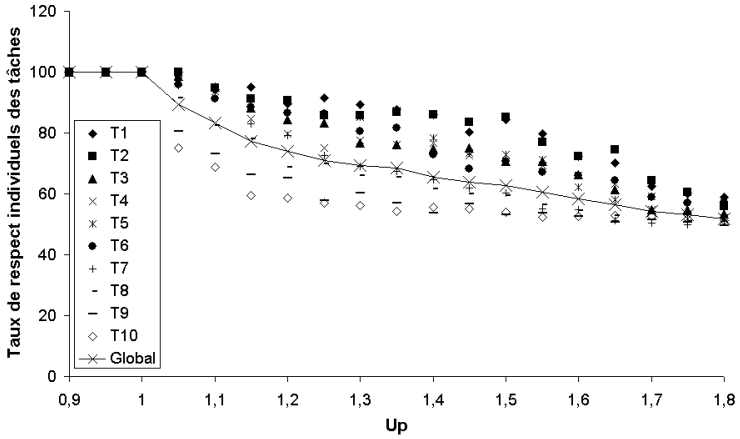

FIG. V.11 – Taux de respect individuels sous RLP-ED en fonction de U_p ($s_i = 2$)

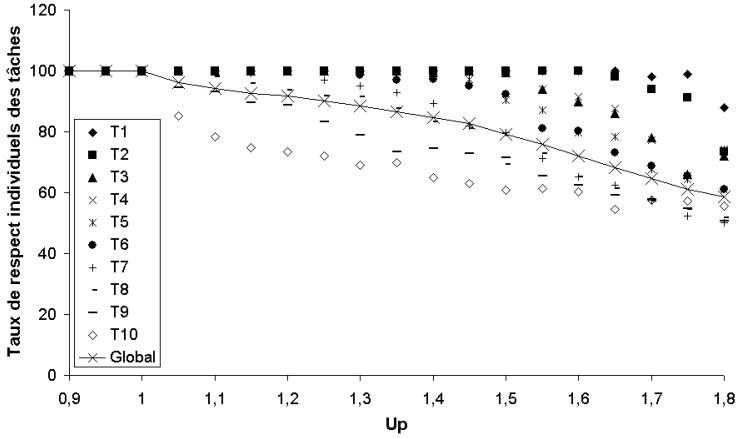

FIG. V.12 – Taux de respect individuels sous RLP/T-ED en fonction de U_p ($s_i = 2$)

Ainsi, pour $U_p = 170\%$, la plus grande différence entre les taux de respect individuels des tâches s'élève respectivement à 34%, 14% et 41% sous BWP-ED, RLP-ED et RLP/T-ED.

Ces études en simulation dénotent le manque de stabilité des ordonnanceurs BWP-ED, RLP-ED et RLP/T-ED dans lesquels les instances de tâches bleues sont ordonnancées selon Earliest Deadline (ED). Le concept de priorité sur lequel est basé l'algorithme ED et qui représente l'urgence relative d'une tâche, est d'évidence inadéquat à l'implémentation du modèle Skip-Over car conduisant à l'échec répété des mêmes instances de tâches bleues.

5.3 Evaluation de la stabilité avec LF

5.3.1 Evaluation des taux de respect individuels

Les variantes LF des ordonnanceurs BWP, RLP, et RLP/T, ont été évaluées avec le modèle de simulation décrit précédemment dans la section 5.1. Les figures V.13, V.14, et V.15 présentent les résultats des mesures effectuées en termes de taux de succès individuels des tâches sous BWP-LF, RLP-LF, et RLP/T-LF.

Les résultats expérimentaux affichent un accroissement de la stabilité du système avec la variante LF des stratégies BWP, RLP et RLP/T. Les taux de respect individuels des tâches ont été regroupés autour de la courbe correspondant au taux de respect global des tâches.

L'écart maximum observé entre deux taux de succès individuels de tâches sous BWP-LF, n'est plus que de 16%. Sous RLP/T-LF, le gain de stabilité obtenu par rapport à RLP/T-ED, est encore plus important avec une différence maximale entre les pourcentages de réussite individuels de moins de 11%.

Remarquons également que RLP/T-LF est d'autant plus stable que la charge périodique appliquée au système est faible. Ainsi, tant que $U_p < 115\%$, l'écart maximal entre les taux de réussite individuels reste inférieur à 4%. Ce constat n'est plus valable si l'on considère l'algorithme BWP-LF pour lesquel le degré de stabilité affiché semble indépendant de la charge périodique du système.

Quant à l'algorithme RLP-LF, comme dans le cas de RLP-ED, la meilleure stabilité est obtenue pour des charges périodiques très élevées ($U_p > 165\%$).

Au vu de ces résultats, nous pouvons dire que la variante LF pour l'ordonnancement des instances de tâches bleues pour le modèle Skip-Over, améliore de façon significative la stabilité du système, quelle que soit la stratégie globale d'ordonnancement.

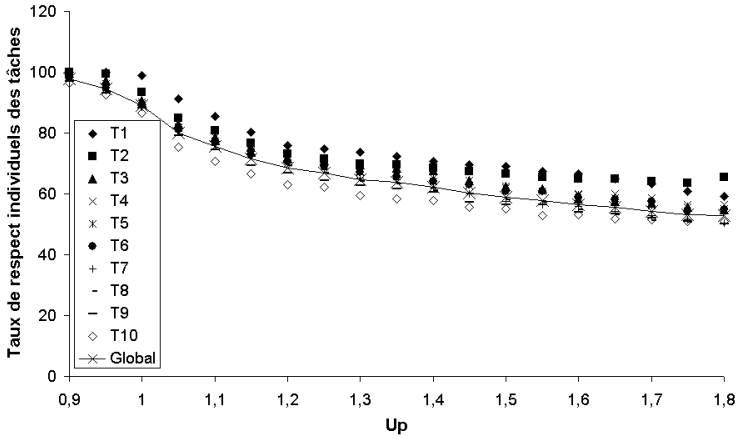

FIG. V.13 – Taux de respect individuels sous BWP-LF en fonction de U_p ($s_i = 2$)

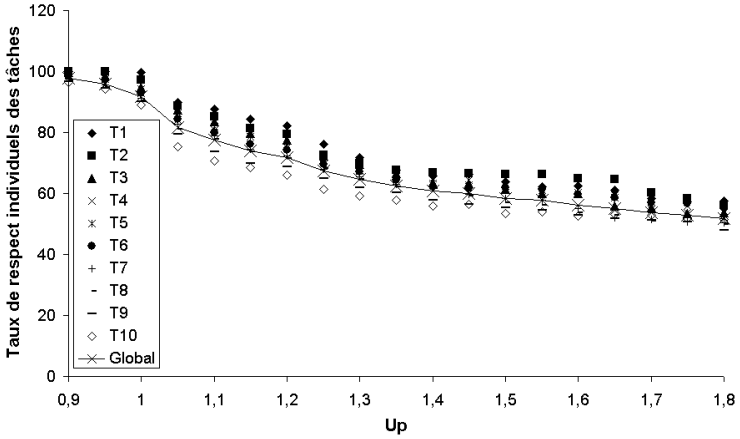

FIG. V.14 – Taux de respect individuels sous RLP-LF en fonction de U_p ($s_i = 2$)

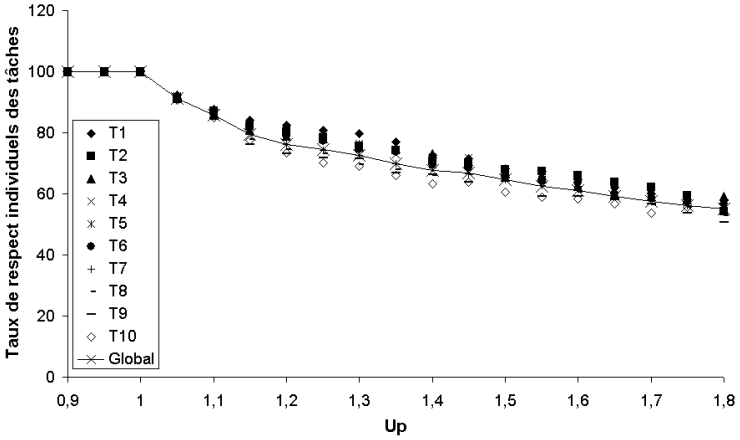

FIG. V.15 – Taux de respect individuels sous RLP/T-LF en fonction de U_p ($s_i = 2$)

5.3.2 Evaluation du temps moyen d'obtention de la stabilité

L'objectif de cette deuxième série de simulations est d'évaluer la performance dynamique de la variante LF des ordonnanceurs en termes de temps moyen au-delà duquel un équilibre entre les taux de réussite individuels des tâches est véritablement atteint. Les différents ordonnanceurs ont donc été simulés sur 10 hyperpériodes pour une charge périodique constante $U_p = 140\%$. Les figures V.16, V.17 et V.18 représentent les variations dans le temps des taux de respect individuels des tâches, pour BWP-LF, RLP-LF et RLP/T-LF respectivement.

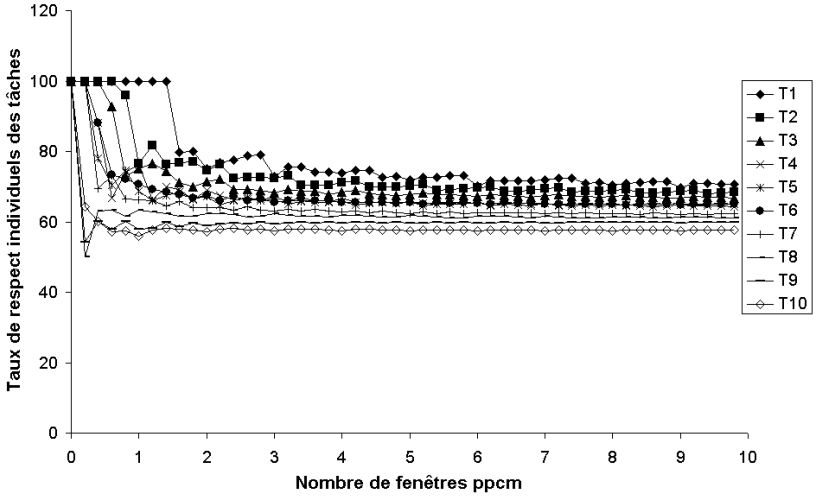

FIG. V.16 – Taux de respect individuels sous BWP-LF en fonction du temps ($s_i = 2$)

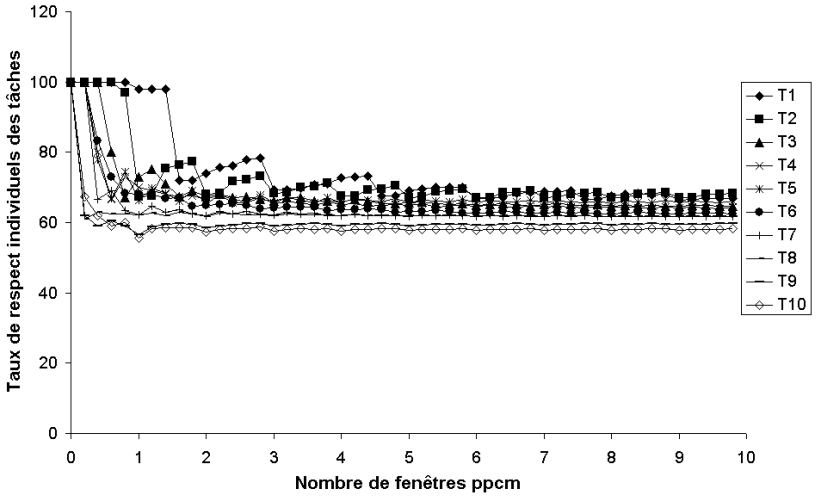

FIG. V.17 – Taux de respect individuels sous RLP-LF en fonction du temps ($s_i = 2$)

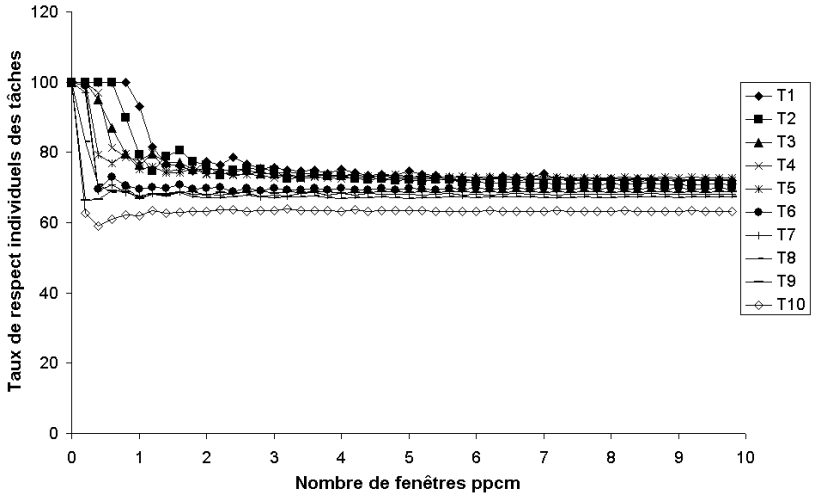

FIG. V.18 – Taux de respect individuels sous RLP/T-LF en fonction du temps ($s_i = 2$)

Nous constatons que les stratégies proposées tendent vers l'équilibre des pourcentages de réussite individuels des tâches assez rapidement. La stabilisation du taux de respect est plus élevée sous RLP/T-LF que sous BWP-LF et RLP-LF. Nous observons en effet que RLP/T-LF atteint une bonne stabilité dès la fin de la première hyperpériode tandis qu'il faut attendre la fin de la troisième hyperpériode pour les deux autres algorithmes. Ceci est dû au fait que sous RLP/T, la politique LF est appliquée au moment de l'occurrence de l'instance de tâche bleue et non au moment de son élection pour l'exécution. Une fois l'équilibre atteint, remarquons que pour l'ensemble des algorithmes considérés, les variations observées entre les pourcentages de réussite individuels sont très faibles.

5.4 Evaluation de la stabilité avec MS

5.4.1 Evaluation des taux de respect individuels

Etudions à présent les performances des variantes MS des algorithmes. Le contexte de simulation utilisé est identique à celui décrit auparavant dans ce chapitre, pour l'évaluation des algorithmes Skip-Over basés sur une politique d'ordonnancement des instances bleues selon ED ou LF. Les résultats de simulation relatifs à BWP-MS, RLP-MS et RLP/T-MS sont rapportés sur les figures V.19, V.20 et V.21 respectivement.

Nous constatons que le gain de stabilité obtenu ici grâce à l'ordonnancement MS des instances de tâches bleues, est encore plus important que dans le cas de la politique d'ordonnancement LF. Les stratégies BWP-MS et RLP/T-MS apparaissent très stables quelle que soit la charge périodique du système, affichant un écart moyen entre les pourcentages de réussite individuels de 1.17% et 2.33% respectivement. Les mesures des taux de respect individuels sous RLP-MS nous montrent en revanche que le gain de stabilité n'est pas identique pour toutes les charges périodiques. Ce dernier est en effet moins stable lorsque le système éprouve une surcharge légère, c'est-à-dire dans notre

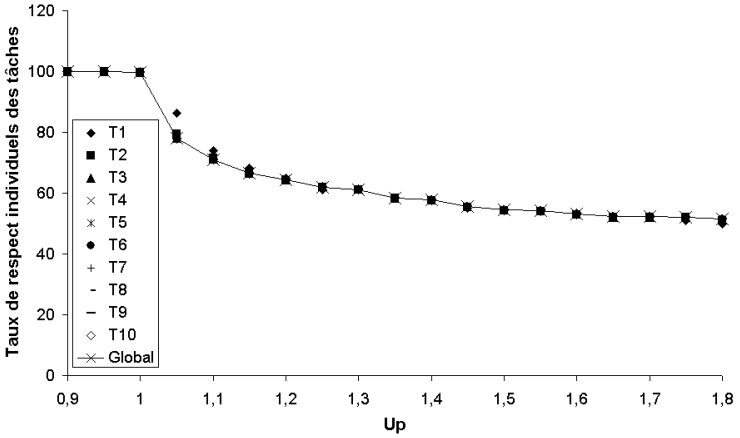

FIG. V.19 – Taux de respect individuels sous BWP-MS en fonction de U_p ($s_i = 2$)

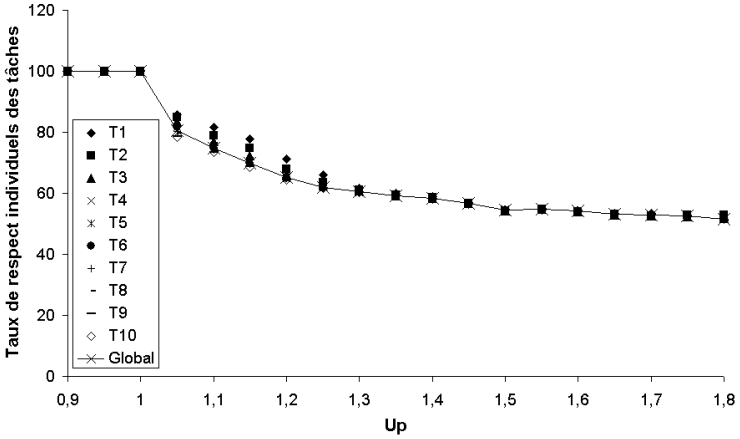

FIG. V.20 – Taux de respect individuels sous RLP-MS en fonction de U_p ($s_i = 2$)

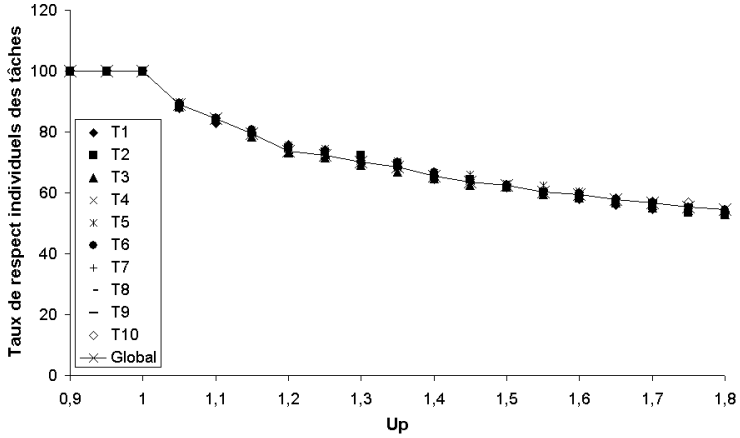

FIG. V.21 – Taux de respect individuels sous RLP/T-MS en fonction de U_p ($s_i = 2$)

cas, pour $U_p < 130\%$. Au-delà, la stabilité obtenue est proche de celle observée sous BWP-MS avec un écart moyen entre les taux de succès individuels de l'ordre de 1%.

L'ensemble des résultats nous montre donc que la stabilité du système est fortement améliorée en utilisant MS conjointement aux différents ordonnanceurs Skip-Over.

5.4.2 Evaluation du temps moyen d'obtention de la stabilité

Comme dans le cas de LF, nous évaluons à présent la performance dynamique de MS, en mesurant la durée de l'état transitoire précédant le fonctionnement stable de l'algorithme dans lequel le nombre d'instances bleues réussies par rapport aux requêtes engendrées est identique pour toutes les tâches, dans une fourchette ϵ donnée.

La durée de la simulation est égale à 10 hyperpériodes et la charge périodique appliquée est constante et telle que $U_p = 140\%$. Les figures V.22, V.23 et V.24 représentent les variations dans le temps des taux de respect individuels des tâches, pour BWP-MS, RLP-MS et RLP/T-MS respectivement.

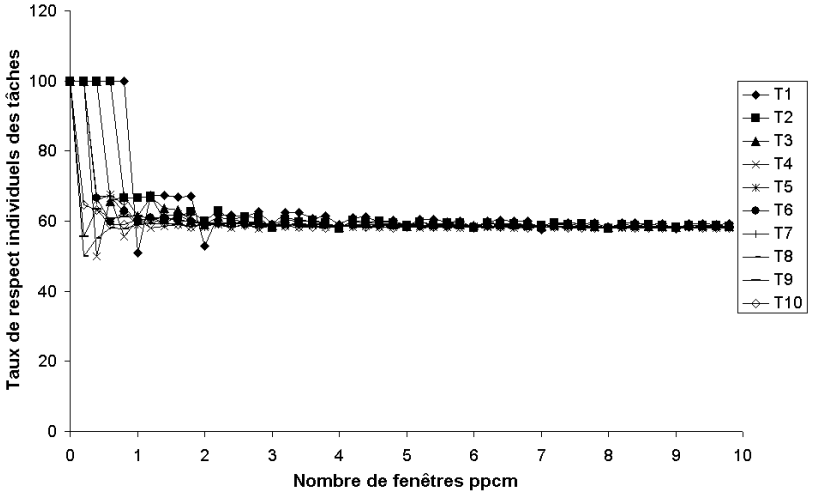

FIG. V.22 – Taux de respect individuels sous BWP-MS en fonction du temps ($s_i = 2$)

Les courbes nous montrent que l'équilibre des pourcentages de réussite individuels est atteint dans un délai assez court. Comme dans le cas de la variante précédente, la stabilité est obtenue plus rapidement sous RLP/T-MS que sous BWP-MS et RLP-MS. Cela s'explique également par le fait que la politique MS est appliquée au moment du test d'acceptation de l'instance bleue au sein du système et non au moment du dispatching des instances sur le processeur. Par ailleurs, nous constatons que RLP-MS affiche un délai d'obtention de l'équilibre légèrement inférieure à celui de BWP-MS. En effet, sur la troisième hyperpériode, l'écart maximum des taux de respect individuels des tâches est égal à 4.94% et 5.39% sous BWP-MS et RLP-MS respectivement.

Enfin, notons que les variations observées au niveau des taux de respect individuels sont minimales au-delà de la 4ème hyperpériode de simulation et ce, quel que soit l'ordonnanceur considéré.

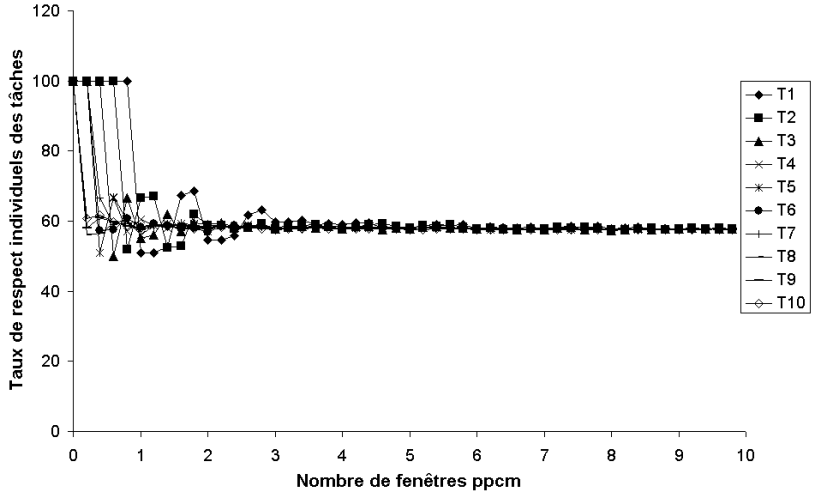

FIG. V.23 – Taux de respect individuels sous RLP-MS en fonction du temps ($s_i = 2$)

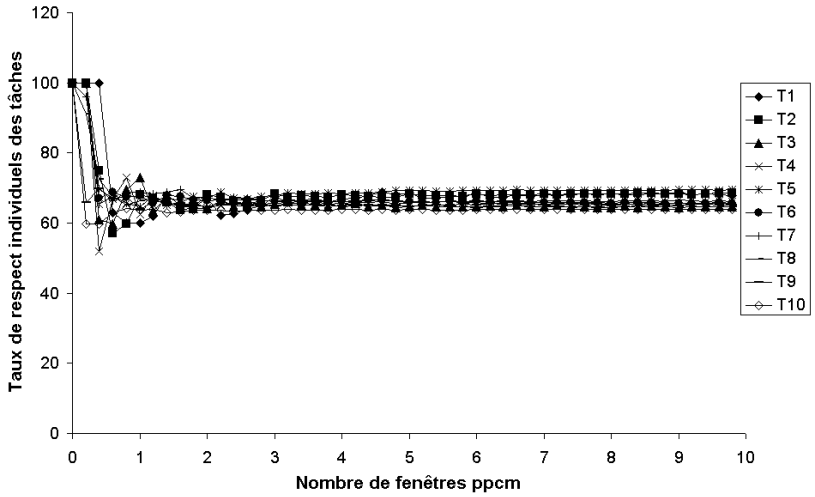

FIG. V.24 – Taux de respect individuels sous RLP/T-MS en fonction du temps ($s_i = 2$)

5.5 Evaluation de la robustesse avec LF et MS

Evaluons à présent le degré de robustesse des variantes stables des algorithmes BWP, RLP et RLP/T. Les mesures effectuées ici reposent sur la fraction de tâches périodiques qui s'accomplissent dans le respect de leur échéance. L'évaluation est menée en fonction de la charge périodique U_p appliquée au système. Les résultats obtenus pour $s_i = 2$ sont décrits sur les figures V.25, V.26, et V.27 représentant la QoS offerte sous les variantes de BWP, RLP et RLP/T respectivement.

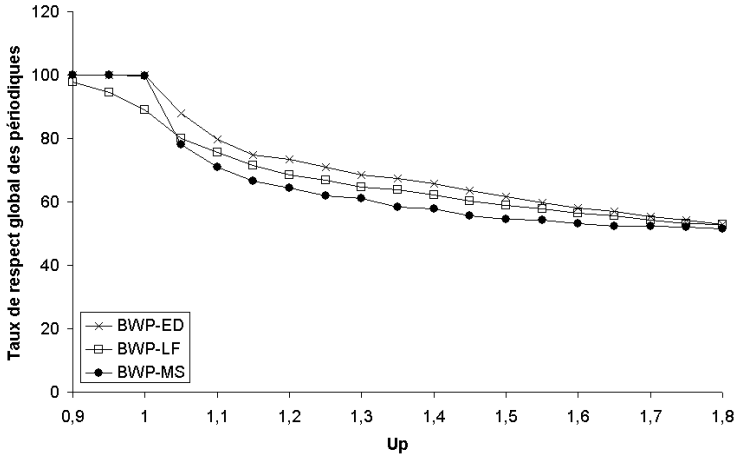

FIG. V.25 – Taux de respect global des tâches sous BWP en fonction de U_p ($s_i = 2$)

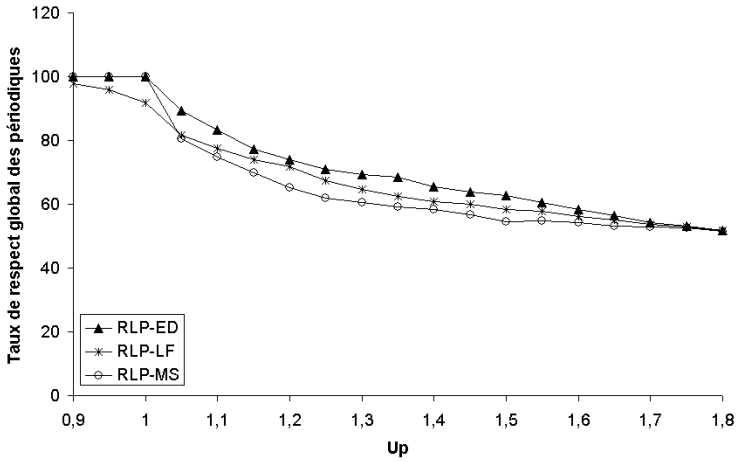

FIG. V.26 – Taux de respect global des tâches sous RLP en fonction de U_p ($s_i = 2$)

Les courbes nous montrent que les ordonnanceurs utilisant LF et MS sont moins robustes globalement que les ordonnanceurs dans lesquels les instances de tâches bleues sont ordonnancées selon ED. Nous remarquons de plus que, pour l'ensemble des algorithmes, la variante MS offre les performances les plus médiocres en termes de robustesse, dès lors que le système est surchargé. En revanche, lorsque le système n'est pas en surcharge ($U_p \leq 100\%$), la variante MS affiche les mêmes résultats de performance que la version ED (ceci est également vrai pour la variante LF de RLP/T du fait du test d'acceptation des instances bleues mis en œuvre sous cet algorithme).

En ce qui concerne les algorithmes BWP et RLP, la variante LF offre une QoS intermédiaire située entre celle de MS et celle de ED, sauf lorsque $U_p \leq 100\%$ où la courbe se situe en deçà de toutes les autres, se positionnant ainsi au taux de QoS le plus faible de l'ensemble des versions. Par ailleurs, notons que la perte de robustesse pour les

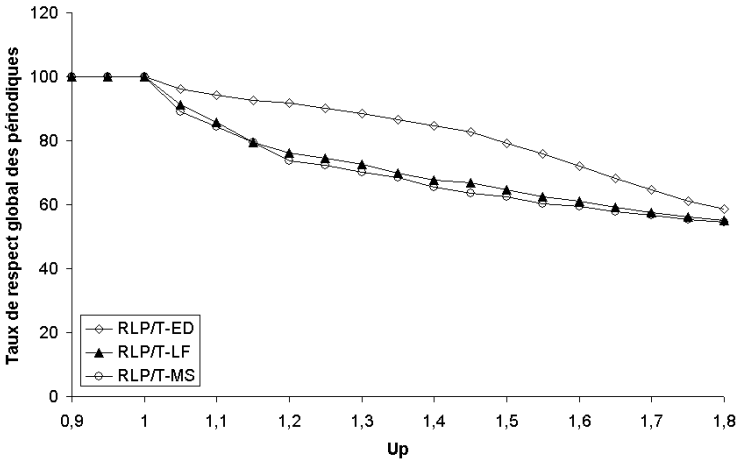

FIG. V.27 – Taux de respect global des tâches sous RLP/T en fonction de U_p ($s_i = 2$)

versions LF et MS est beaucoup plus importante sous RLP/T que celle observée sous BWP ou RLP. A titre illustratif, pour $U_p = 140\%$, l'écart de performance en termes du taux de respect global des tâches, observé entre les variantes MS et ED est de l'ordre de 8%, 7%, et 19%, sous BWP, RLP et RLP/T respectivement. Notons enfin que pour chaque algorithme, l'ensemble de ses versions tend vers le même taux de respect global lorsque la charge périodique appliquée au système devient très élevée.

En résumé, nous pouvons dire que le gain de stabilité acquis sous les variantes LF et MS s'assortit d'une perte de robustesse. Celle-ci peut être considérée comme acceptable sous BWP et RLP car assez faible en moyenne. En revanche, la perte observée sous RLP/T est plus significative.

6 Synthèse

Nous présentons dans cette partie une synthèse de la performance des ordonnanceurs BWP, RLP et RLP/T utilisés avec les différentes variantes ED, LF et MS, en termes de stabilité d'une part et de robustesse d'autre part.

6.1 Synthèse en termes de stabilité

L'ensemble des critères retenus pour l'évaluation de la stabilité des différentes stratégies ainsi que leurs mesures relatives en fonction des algorithmes étudiés, sont regroupés dans la table V.2. Les critères évalués sont les suivants : la différence moyenne d_{mean}, la plus grande différence d_{max}, et l'écart-type σ, calculés sur l'ensemble des valeurs correspondant aux taux de respect individuels des tâches.

Rappelons que l'écart-type est la mesure la plus communément rencontrée pour traduire la dispersion statistique d'un ensemble de mesures. L'écart-type indique comment en moyenne les valeurs (ici les pourcentages de réussite individuels des tâches) sont

groupées autour de la tendance centrale (moyenne arithmétique). Un faible écart-type signifie que les valeurs sont peu dispersées autour de la moyenne (série homogène), et inversement (série hétérogène).

Algorithmes	d_{mean}	d_{max}	σ
BWP-ED	30.62	43.86	8.55
BWP-LF	12.85	15.77	3.76
BWP-MS	1.17	8.71	0.34
RLP-ED	21.96	35.54	7.50
RLP-LF	11.52	17.00	3.56
RLP-MS	2.34	9.20	0.72
RLP/T-ED	28.28	46.75	9.64
RLP/T-LF	6.66	10.79	2.08
RLP/T-MS	2.34	3.62	0.72

TAB. V.2 – Critères de stabilité

Les résultats présentés dans cette table confortent les résultats de simulation précédent en soulignant le gain de stabilité obtenu grâce aux variantes LF et MS. Notons que la meilleure stabilité est acquise sous BWP-MS ($\sigma = 0.34$) tandis que RLP-MS et RLP/T-MS offre un équilibre des taux de réussite individuels des tâches, identique ($\sigma = 0.72$). Au niveau des variantes LF en revanche, c'est l'ordonnanceur RLP/T-LF qui apparaît le plus stable ($\sigma = 2.08$) suivi de RLP-LF ($\sigma = 3.56$) puis de BWP-LF ($\sigma = 3.76$). Remarquons enfin les écart-types de BWP-ED, RLP-ED et RLP/T-ED qui traduisent le comportement médiocre de ces variantes en termes d'équilibre des taux de respect individuels. Ainsi, nous pouvons dressé le tableau récapitulatif suivant (cf. V.3) résumant les performances des différents ordonnanceurs en termes de stabilité.

Algorithmes	ED	LF	MS
BWP	★	★★	★★★
RLP	★	★★	★★★
RLP-T	★	★★	★★★

TAB. V.3 – Synthèse de la stabilité des ordonnanceurs étudiés

6.2 Synthèse en termes de robustesse

Nous exposons enfin une synthèse concernant la robustesse des ordonnanceurs. Pour appuyer notre propos, nous présentons, sur un même graphique, le taux de respect global des tâches sous BWP-ED, BWP-LF, BWP-MS, RLP/T-ED, RLP/T-LF et RLP/T-MS, en fonction de la charge périodique U_p. Par souci de lisibilité, nous n'avons pas fait figurer les courbes relatives aux versions de RLP. La figure V.28 visualise les résultats obtenus.

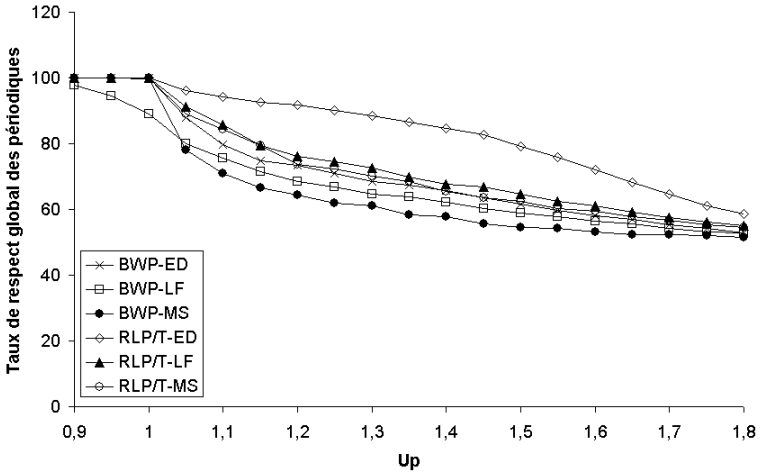

FIG. V.28 – Taux de respect global des tâches sous BWP et RLP/T en fonction de U_p ($s_i = 2$)

Ce graphique offre des résultats intéressants dans la mesure où il fait apparaître le fait que les versions stables de RLP/T, à savoir RLP/T-LF et RLP/T-MS, sont plus robustes que la version BWP-ED et ce, quelle que soit la charge périodique appliquée au système. Par ailleurs, nous observons la nette prédominance de performance de l'algorithme RLP/T-ED sur l'ensemble des algorithmes figurés. Par conséquent, les résultats obtenus en termes de robustesse peuvent être synthétisés par le tableau V.4 présenté ci-dessous.

Algorithmes	ED	LF	MS
BWP	★★★	★★	★
RLP	★★★★	★★★	★★
RLP-T	★★★★★	★★★★	★★★

TAB. V.4 – Synthèse de la robustesse des ordonnanceurs étudiés

7 Conclusion

Dans ce chapitre, nous avons étudié la stabilité et la robustesse des ordonnanceurs RTO, BWP, RLP, et RLP/T. Après avoir souligné leur manque de stabilité, au sens de l'équilibre des taux de respect individuels des tâches, nous avons proposé deux variantes d'ordonnancement des instances de tâches bleues, à savoir LF et MS et ce, en remplacement de Earliest Deadline. L'évaluation menée a permis de vérifier le gain de stabilité procuré 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 stabilité sur la QoS globale observée au niveau du système. Il est apparu que l'optimisation de la stabilité s'accompagne d'une dégradation plus ou moins importante de la robustesse selon les ordonnanceurs. Cependant, nous avons vu que les versions RLP/T-LF et RLP/T-MS, confèrent un bon compromis en garantissant à la fois robustesse et stabilité au système.

En conclusion, nous pouvons dire que les deux critères de performance que sont la stabilité et la robustesse apparaissent orthogonaux. Aussi, le choix de l'algorithme d'ordonnancement des instances de tâches bleues incombe à l'utilisateur qui doit sélectionner l'algorithme qui convient le mieux à l'application physique contrôlée, en fonction de ses exigences de stabilité et de robustesse.

Notons qu'il serait à présent intéressant d'étendre ce travail à des ensembles pour lesquels les tâches n'ont pas la même importance vis-à-vis de l'application (c'est-à-dire des tâches dont les paramètres de pertes s_i ne sont pas tous identiques).

Chapitre VI

Implantation sous Linux temps réel

Ce chapitre traite de l'intégration sous Linux temps réel des différentes stratégies d'ordonnancement sous contraintes de QoS, et de celle du serveur optimal EDL utilisé conjointement avec ces stratégies. Ce travail d'intégration sous la forme de modules chargeables dynamiquement dans le noyau, s'est effectué dans la continuité du projet RNTL CLEOPATRE dont nous décrivons les enjeux et la finalité dans une première partie. Ensuite, nous nous attachons à présenter le système d'exploitation temps réel Linux/RTAI, support d'exécution des composants développés. Puis, nous exposons les structures de données implémentées sous Linux/RTAI pour l'intégration des stratégies d'ordonnancement RTO, BWP, RLP et RLP/T, ainsi que celles relatives au serveur de tâches apériodiques EDL. Enfin, nous avons mené une évaluation des performances de l'ensemble des composants intégrés. Celle-ci concerne d'une part la mise en oeuvre d'applications de test puis la visualisation sous LTT des séquences d'ordonnancement générées, et d'autre part l'évaluation des caractéristiques des composants au niveau des empreintes mémoire et disques. Cette évaluation comprend également la présentation de résultats de mesures relatifs aux overheads des différents composants.

---

1 Contexte du travail

1.1 Le projet CLEOPATRE

1.1.1 Finalité du projet

L'émergence d'applications industrielles de plus en plus complexes montre la nécessité croissante d'un transfert des compétences universitaires en matière d'informatique temps

réel. C'est dans cette perspective que s'est inscrit le projet RNTL[1] CLEOPATRE[2] qui fut labellisé en avril 2001 et notifié en janvier 2002 par le Ministère de l'Education et de la Recherche français. L'objectif de ce projet de type exploratoire et d'une durée de 36 mois, était d'apporter des solutions au développement d'applications temps réel embarquées par la mise à disposition après vérification, de composants logiciels libres et ouverts s'appuyant sur Linux. Le but était d'enrichir les services temps réel des versions existantes de Linux pour le temps réel, telles que RTLinux [Yod04] ou Linux/RTAI [MBD+00]. La finalité du projet était donc de mettre à disposition du monde industriel et universitaire, une bibliothèque de composants logiciels s'appuyant sur des résultats scientifiques reconnus et validés de façon formelle.

1.1.2 Acteurs du projet

Le projet CLEOPATRE est le fruit de la collaboration de six laboratoires et industriels. Le consortium établi était le suivant :

- **LINA** *(Laboratoire d'Informatique de Nantes Atlantique)* de l'université de Nantes,
- **LRV** *(Laboratoire de Robotique de Versailles)* rattaché au CNRS,
- **L2TI** *(Laboratoire de Traitement et de Transport de l'Information)* de l'université de Paris Nord,
- **CEA** *(Commissariat à l'Energie Atomique)*,
- **CRTTI** *(Centre de Recherche et de Transfert Technologique Industriel)* de l'université de Nantes,
- **ROBOSOFT S.A**.

Chaque partenaire était tenu de contribuer au projet selon un cahier des charges précis. La contribution des 3 premiers partenaires précédemment cités, consistait à fournir un ensemble de composants sélectionnables (COTS[3]) selon le logiciel d'application visé. Le LINA était ainsi chargé de proposer une bibliothèque de composants de *niveau noyau* en matière d'ordonnancement dynamique et de tolérance aux fautes temporelles, tandis que le cahier des charges du L2TI et du LRV comprenait le développement de bibliothèques de composants de *niveau applicatif* en matière d'algorithmique de vision et de contrôle/commande respectivement. Le CEA devait s'assurer que les partenaires s'entendent sur les interfaces entre les différentes bibliothèques développées ainsi que sur les méthodes de développement (archivage, gestion des versions, gestion documentaire, etc.). Le CRTTI quant à lui, s'est vu responsable de la conception d'une plate-forme robotique mobile de démonstration, visant à montrer, sur une application réelle, l'adéquation de Linux temps réel muni de ses composants noyau et de sa librairie d'utilitaires, relativement aux exigences des développeurs. Enfin, la société ROBOSOFT en tant que fournisseur de technologie, était en charge de la documentation et de la dissémination via Internet des travaux réalisés dans le cadre du projet.

[1] Réseau National des Technologies Logicielles
[2] Composants Logiciels sur Etagères Ouverts Pour les Applications temps réel Embarquées
[3] Commercial-Off-The-Shelf

Dans la suite, nous limitons notre exposé au sous-projet dont était en charge le LINA, à savoir la bibliothèque de composants intégrés au niveau noyau.

1.1.3 Description de la librairie CLEOPATRE

La structure de la librairie CLEOPATRE [SG06] représentée sur la figure VI.1, offre un panel de composants sélectionnables dédiés à l'ordonnancement dynamique, au service de tâches apériodiques, à la gestion de l'accès aux ressources, à la tolérance aux fautes temporelles, et à la gestion de surcharge.

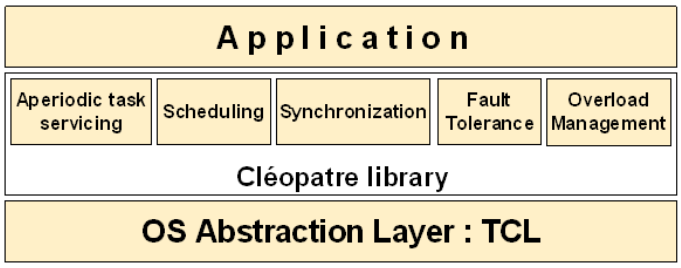

FIG. VI.1 – Structure de la librairie CLEOPATRE

La librairie CLEOPATRE s'articule autour de 5 étagères fournissant différents services temps réel. Les composants relatifs à l'ordonnancement de tâches périodiques sont basés sur 2 algorithmes : Deadline Monotonic (DM) et Earliest Deadline First (EDF). En ce qui concerne l'étagère qui permet le service des tâches apériodiques, 2 serveurs ont été implémentés dans le but de gérer l'occurrence de tâches apériodiques à contraintes strictes ou relatives : le serveur BG et le serveur TBS [CLB99]. Cinq protocoles de gestion des ressources sont disponibles : FIFO (First In First Out), Priority, SPP (Super Priority Protocol), PIP (Priority Inheritance Protocol), PCP (Priority Ceiling Protocol) et et DPCP (Dynamic Priority Ceiling Protocol) [SRL90]. Un mécanisme de tolérance aux fautes a également été implémenté : le Mécanisme à Echéance [LC86]. Enfin, le problème de la gestion des cas de surcharge a été traité par l'implémentation du modèle du Calcul Imprécis [CLL90].

Les composants sont interchangeables au sein d'une même étagère et entièrement interopérables. Par exemple, l'utilisateur peut sélectionner l'ordonnanceur EDF et le serveur de tâches apériodiques BG, pour fonctionner avec les mécanismes de gestion de surcharge ou de tolérance aux fautes.

Par ailleurs, les composants sont génériques et totalement indépendants à la fois du système d'exploitation temps réel et du matériel. Ces propriétés sont atteintes grâce à une couche d'abstraction logicielle appelée TCL (Task Layer Control) qui fournit une interface interne générique. Cette couche fait abstraction des caractéristiques spécifiques de chaque plate-forme, de manière à ce que les composants soient portables et qu'ils puissent s'adapter aisément à un processeur cible et à une architecture matérielle donnés. Cette couche est ainsi responsable de la gestion des tâches à travers diverses fonctions (*TCL_create(), TCL_destroy(), TCL_kill(), TCL_ready(), TCL_block() et TCL_schedule()*)

décrites dans [Gar05].

Le développement initial de la couche TCL a été effectué sous la variante de Linux pour le temps réel dénommée Linux/RTAI (noyau 2.2.14, RTAI 1.3). Ce choix repose principalement sur le souhait de distribuer les composants logiciels CLEOPATRE sous licence LGPL[4], licence qui correspond également à la distribution du projet RTAI. La généricité des composants développés a cependant été validée en intégrant la librairie CLEOPATRE sous la dernière version libre de RTLinux. Toutefois, à l'heure actuelle, la librairie CLEOPATRE est maintenue exclusivement sous Linux/RTAI (noyau Linux 2.4.22, RTAI 3.0). Une présentation de ce système d'exploitation temps réel (RTOS) ouvert est effectuée ci-après, de manière à expliciter ensuite l'interfaçage des composants CLEOPATRE avec RTAI.

1.2 Présentation de Linux/RTAI

1.2.1 Origine et Description

Le projet RTAI (Real-Time Abstraction Interface) a pour origine le Département d'Ingénierie Aérospatiale de l'école Polytechnique de Milan (DIAPM). Dans une optique de développement interne visant à réduire les coûts de licence sur les systèmes d'exploitation temps réel propriétaires comme QNX (alors utilisé au DIAPM), Paolo Mantegazza et son équipe décident de développer une variante de RTLinux, intégrant de nouveaux services tels que l'ordonnancement en mode périodique et la gestion des nombres flottants, alors inexistants sous RTLinux.

RTAI est intégré à Linux grâce à un patch noyau et une série de programmes additionnels (modules chargeables dynamiquement) qui étendent le noyau Linux au temps réel dur. Le système d'exploitation temps réel Linux/RTAI présente les spécifications d'un RTOS temps réel industriel. Il repose sur le concept d'une couche d'abstraction matérielle comme l'illustre la figure VI.2 [SG06].

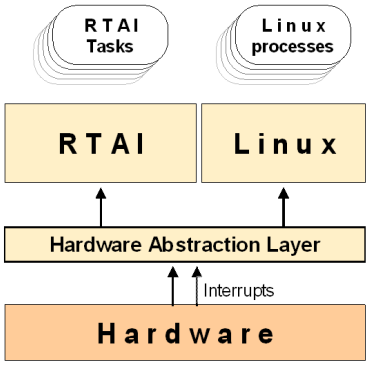

FIG. VI.2 – Modèle d'abstraction de Linux/RTAI

La couche d'abstraction peut être vue comme un dispatcher d'interruptions dans

[4]Lesser General Public License

lequel toutes les interruptions matérielles transitent en priorité vers RTAI, avant d'être redirigées si nécessaire vers Linux. Ce schéma autorise les tâches temps réel de RTAI à s'exécuter conjointement avec les processus Linux. Linux est alors considéré par RTAI comme une tâche de fond qui s'exécute lorsqu'il n'y a plus d'activité temps réel au sein du système.

Au départ, la couche d'abstraction utilisée était la même que celle observée sous RT-Linux, à savoir la couche RTHAL (Real-Time Hardware Abstraction Layer). Cependant, par la suite, RTAI se positionnant de plus en plus comme concurrent vis-à-vis du projet RTLinux qui avait quant à lui, une visée plutôt commerciale, un brevet de licence a été apposé en 2001 sur la couche RTHAL par les responsables du projet RTLinux. La position libre de RTAI étant remise en cause par ce brevet, une refonte de RTAI autour du micro-noyau ADEOS (Adaptative Domain Environment for Operating Systems) a été effectuée pour remplacer la couche initiale RTHAL. Implémenté par Philippe Gerum en 2002, ADEOS est basé sur une idée originale de Karim Yaghmour publiée dans [Yag01]. Les nouvelles versions de RTAI reposent à présent exclusivement sur ADEOS.

1.2.2 Interfaçage de CLEOPATRE avec RTAI

De multiples domaines de priorités co-existent simultanément au sein du système Linux/RTAI/CLEOPATRE, comme le montre le schéma présenté sur la figure VI.3.

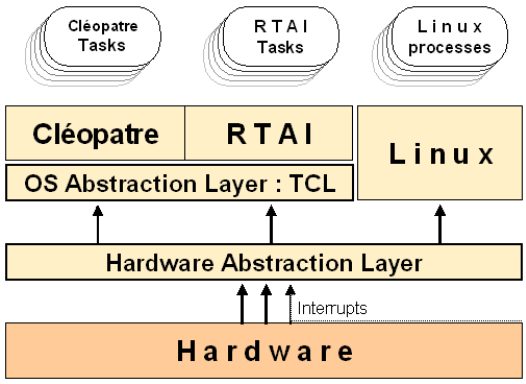

FIG. VI.3 – Modèle d'abstraction de Linux/RTAI/CLEOPATRE

CLEOPATRE est le domaine de plus haute priorité. Celui-ci intercepte les interruptions en amont de RTAI et de Linux, ce dernier correspondant au domaine de plus faible priorité. La couche d'abstraction TCL vient ici interfacer CLEOPATRE avec RTAI, en intégrant et modifiant l'ordonnanceur natif de RTAI. Les tâches CLEOPATRE sont toujours exécutées en priorité, celles de RTAI sont ordonnancées lorsqu'il n'y a plus de tâches CLEOPATRE à l'état prêt dans le système, et enfin les processus Linux obtiennent la ressource processeur dès lors qu'il n'y a plus de tâches temps réel à exécuter.

Les applications de RTAI natif fonctionnent sans modifications aucunes sous l'environnement CLEOPATRE. Ce modèle d'abstraction permet ainsi d'intégrer sous Li-

nux/RTAI de nouvelles fonctionnalités temps réel, de manière claire et efficace.

Dans la suite, nous présentons d'une part l'intégration sous CLEOPATRE des algorithmes d'ordonnancement RTO, BWP, RLP et RLP/T sous la forme d'une nouvelle étagère dénomée "QoS", et d'autre part celle du serveur EDL venant enrichir l'étagère "Serveurs de tâches apériodiques" existante.

2 Intégration d'une étagère "QoS" sous CLEOPATRE

Dans cette partie, nous nous intéressons à l'intégration dans la librairie CLEOPATRE d'une nouvelle étagère de composants d'ordonnancement appelée "Quality Of Service". Celle-ci regroupe l'ensemble des algorithmes étudiés dans le chapitre III, à savoir les algorithmes RTO, BWP, RLP et RLP/T. Après avoir décrit les structures de données internes communes aux 4 composants, nous exposons brièvement leur fonctionnement interne, avant de décrire l'interface utilisateur qui leur est associée.

2.1 Définition des structures de données utilisées

2.1.1 Descripteur de tâches étendu

La structure de données principale sur laquelle repose l'ensemble des ordonnanceurs de l'étagère "QoS" est un *descripteur de tâches* étendu pour la gestion de tâches définies sous des contraintes de pertes de type Skip-Over. Celui-ci est défini dans le fichier d'entêtes *QoS.h* accessible dans le sous-répertoire des fichiers d'entêtes de RTAI dont le chemin d'accès complet est *$RTAI_DIR/include*, où *$RTAI_DIR* représente le répertoire d'installation de RTAI. Le fichier *QoS.h* renferme un descripteur de tâches dénommé *QoSTaskType*, adapté aux tâches sous contraintes de QoS. Ce dernier est complètement décrit par la structure de données suivante :

```
typedef struct QoSTaskStruct QoSTaskType;
struct QoSTaskStruct {
  void (*fct) (QoSTaskType *);      /*pointeur sur la fonction de la tâche*/
  TaskType TCL_task;                /*descripteur de tâche bas-niveau*/
  TimeType WCET;                    /*durée d'exécution au pire-cas*/
  TimeType critical_delay;          /*délai critique*/
  TimeType period;                  /*période d'activation*/
  TimeType release_time;            /*date de réveil*/
  unsigned int max_skipvalue;       /*tolérance maximale aux pertes*/
  unsigned int current_skipvalue;   /*paramètre de pertes dynamique*/
  unsigned int current_shift;       /*décalage vis-à-vis d'une séquence RTO*/
  unsigned int slack;               /*laxité de la tâche*/
};
```

Cette structure de données comprend tout d'abord un pointeur vers la fonction correspondant au code la tâche. Le paramètre suivant nous indique que le descripteur de tâches présenté ici étend celui de TCL gérant des tâches de type *TaskType*. Notons d'ailleurs que le descripteur de tâches de TCL se définit déjà en tant qu'extension du descripteur de tâches *RT_TASK* de RTAI. Les paramètres additionnels de la structure peuvent ensuite être dissociés en deux groupes distincts : les paramètres statiques intrinsèques aux caractéristiques temporelles de la tâche et définis par l'utilisateur (durée d'exécution au pire-cas, délai critique, période d'activation, date de réveil et pertes maximales autorisées), et ceux de nature dynamique relatifs au fonctionnement interne des différents modules d'ordonnancement (pertes dynamiques, décalage dynamique dû à la réussite d'instances bleues, et laxité de la tâche à un instant donné).

2.1.2 Vecteurs de calcul des temps creux selon EDL

La seconde structure de données du fichier *QoS.h* n'est utilisée que par les composants d'ordonnancement RLP et RLP/T de l'étagère "QoS". Elle correspond à l'implémentation des vecteurs de calcul des temps creux selon EDL. Sa description en langage C est fournie ci-dessous.

```
typedef struct {
  TimeType ki;                  /*date du temps creux*/
  TimeType delta_i;             /*durée du temps creux*/
} data;

struct idle_time_cell{
  data v_idle;                  /*temps creux (ki, delta_i)*/
  struct idle_time_cell *next;  /*pointeur vers le temps creux suivant*/
  struct idle_time_cell *prev;  /*pointeur vers le temps creux précédent*/
};
```

Nous avons choisi de représenter la fonction de disponibilité du processeur f^{EDL} par une liste doublement chaînée dans laquelle chaque maillon de type *idle_time_cell* caractérise un temps creux selon le couple *(ki, delta_i)* traduisant respectivement la date et la durée d'un temps creux.

2.2 Description du fonctionnement interne

L'ordonnancement des tâches sous les modules RTO, BWP, RLP et RLP/T est assuré par une fonction propre à chaque composant et répondant au prototypage suivant :

```
void QoS_schedule(void *arg);
```

Cette fonction est définie en tant que *handler de temps* (*time handler* en anglais) via les fonctions d'initialisation *IRQ_create()* et *IRQ_register()* (fonctions internes de la couche TCL [Gar05]), au moment du chargement dynamique du module d'ordonnancement dans le noyau. Une fois le module chargé, la fonction *QoS_schedule()* du composant est exécutée à chaque interruption du timer matériel 8254, dont la fréquence est définie par l'application utilisateur. L'argument d'entrée de la fonction est un pointeur sur une variable correspondant au temps courant de l'application temps réel. Celui-ci est décrit en termes du nombre de tops horloge écoulés depuis le lancement de l'application temps réel.

Pour chacun des composants de l'étagère "QoS", c'est donc au sein de cette fonction qu'est implémenté l'algorithme d'ordonnancement qui le définit. Ainsi, les fonctions *QoS_schedule()* de RTO, BWP, RLP et RLP/T ont été implémentées selon les descriptions algorithmiques fournies précédemment dans Chapitre III.

2.3 Description de l'interface utilisateur

L'interface utilisateur des ordonnanceurs de l'étagère "QoS" est décrite ci-dessous dans la Table VI.1.

Fonctions	Description
QoS_create()	création d'une nouvelle tâche temps réel
QoS_resume()	réveil d'une tâche temps réel
QoS_wait()	attente par la tâche de sa prochaine période
QoS_delete()	suppression d'une tâche temps réel

TAB. VI.1 – Interface utilisateur des ordonnanceurs de l'étagère "QoS"

Nous précisons ci-après, les fonctionnalités de l'ensemble de ces fonctions, à travers la définition de leur prototypage respectif.

2.3.1 La fonction QoS_create()

La primitive *QoS_create()* crée une nouvelle tâche temps réel référencée par un descripteur de tâche t. Le prototypage complet de cette fonction est présentée ci-dessous :

```
int QoS_create (
  QoSTaskType *t,
  void (*fct) (QoSTaskType*),
  TimeType WCET,
  TimeType critical_delay,
  TimeType period,
  unsigned int max_skipvalue,
  TCLCreateType TCL_data) ;
```

t est un pointeur sur une structure du type *QoSTaskType* dont l'espace mémoire doit être fourni par l'application. Celui-ci doit être conservé durant toute la vie de la tâche temps réel. *fct* correspond au point d'entrée de la fonction de la tâche. Les paramètres *WCET*, *critical_delay*, et *period*, représentent les caractéristiques temporelles de la tâche, à savoir sa durée d'exécution au pire-cas C_i, son délai critique D_i et sa période P_i respectivement. *max_skipvalue* spécifie les pertes s_i autorisées au sens Skip-Over au niveau de la tâche. Enfin, *TCL_data* est une structure qui contient les données spécifiques au système d'exploitation. Sous RTAI, cette structure est constituée de quatre paramètres : la taille du tas (*heap size*) et celle de la pile (*stack size*) utilisées par la tâche, un drapeau indiquant le fait que la tâche utilise ou non l'unité arithmétique flottante (*uses fpu flag*), et un pointeur (*signal*) sur une fonction appelée à chaque fois que la tâche est élue suite à un changement de contexte.

En cas de succès, la valeur 0 est retournée. En cas d'échec de l'initialisation, une valeur négative est retournée :

- EINVAL : la structure de tâche pointée par la tâche est déjà utilisée,
- ENOMEM : les octets *stack size* ne peuvent pas être alloués pour la pile.

2.3.2 La fonction QoS_resume()

La primitive *QoS_resume()* permet de réveiller une tâche à un instant donné. Son prototypage est le suivant :

int *QoS_resume*(QoSTaskType **t*, TimeType *release_time*) ;

QoS_resume réveille à la date *release_time* une tâche suspendue au préalable, ou permet le passage à l'état prêt d'une tâche nouvellement créée. En cas de succès, la valeur 0 est retournée. Dans le cas contraire, la valeur négative -1 est retournée pour signifier que la tâche a déjà été réveillée.

2.3.3 La fonction QoS_wait()

L'appel de fonction *QoS_wait()* provoque une attente passive du prochain réveil de la tâche en cours. Le prototypage correspondant à cette primitive est fourni ci-après :

int *QoS_wait*(**void**) ;

Notons que l'exécution de la tâche est suspendue seulement temporairement, jusqu'à la date de prochaine activation de la tâche périodique. En cas de succès, la valeur retournée est 0. En cas d'échec, une valeur négative est retournée.

2.3.4 La fonction QoS_delete()

La primitive *QoS_delete()* supprime la tâche temps réel créée au préalable par la fonction *QoS_create()*. Son prototypage est le suivant :

int *QoS_delete*(QoSTaskType **t*) ;

t se réfère au pointeur du descripteur de la tâche. Si la tâche était en attente d'un sémaphore, celle-ci est retirée de la file d'attente. En cas de cas de succès, la valeur 0 est retournée. En cas d'échec, la valeur négative EINVAL est retournée pour signifier que le pointeur *t* ne se réfère pas à une tâche valide.

Les composants d'ordonnancement de l'étagère QoS, à savoir RTO.o, BWP.o, RLP.o et RLP_T.o, sont tous situés physiquement dans le répertoire $RTAI_DIR/modules/QoS. Leur chargement et déchargement dynamiques dans le noyau s'effectuent de manière classique sous RTAI via les commandes *insmod* et *rmmod* respectivement.

3 Enrichissement de l'étagère "Serveurs" de CLEOPATRE

Contrairement à la section précédente où nous nous sommes focalisés sur l'ordonnancement de tâche périodiques uniquement, nous nous intéressons ici aux applications temps réel impliquant à la fois des tâches périodiques et des tâches apériodiques critiques et/ou non critiques ; la gestion de ces dernières nécessitant la mise en œuvre d'un serveur de tâches apériodiques.

L'étagère "Serveurs de tâches apériodiques" de CLEOPATRE comporte actuellement deux serveurs dont le comportement a été validé : le serveur BG et le serveur TBS. Nous nous proposons dans cette partie d'enrichir cette étagère de composants par l'ajout du serveur optimal EDL. Notons que le composant a été conçu de manière à ce qu'il puisse fonctionner indifféremment avec des tâches périodiques classiques (c'est-à-dire sans pertes) et avec des tâches périodiques sous contraintes de QoS.

3.1 Définition des structures de données utilisées

L'interface commune aux différents serveurs de tâches apériodiques est contenue dans le fichier d'entêtes *$RTAI_DIR/include/ats.h*. Le descripteur de tâches étendu cette fois-ci pour la gestion d'applications impliquant à la fois des tâches périodiques et des tâches apériodiques, repose sur la structure de données suivante :

```
typedef struct AtsTaskStruct AtsTaskType;
struct AtsTaskStruct {
  RndType(AtsTaskType);
  void (*fct) (AtsTaskType *);   /*pointeur sur la fonction de la tâche*/
  DschTaskType Dsch_task;        /*support pour l'étagère "Scheduling"*/
  TimeType WCET                  /*Worst-Case Execution Time*/
  QoSTaskType QoS_task;          /*support pour l'étagère "QoS"*/
  TimeType slack;                /*laxité de la tâche (utilisé uniquement par EDL)*/
};
```

Cette structure comportait 4 paramètres uniquement à l'origine décrits dans [Gar05], assurant le fonctionnement des serveurs BG et TBS. Dans le cadre de nos travaux, nous lui avons apporté les modifications suivantes : d'une part, nous l'avons étendue au fonctionnement de EDL par l'ajout du paramètre *slack* reflétant le calcul de la laxité de la tâche par EDL et d'autre part, nous avons étendu le fonctionnement de l'ensemble des serveurs avec l'étagère des ordonnanceurs de tâches périodiques avec contraintes de QoS, par l'adjonction du paramètre *QoS_task*.

Notons que pour que les composants de l'étagère "Serveurs de tâches apériodiques" fonctionnent, toutes les tâches, y compris les tâches périodiques, doivent être déclarées par l'utilisateur selon le type *AtsTaskType* dans l'application temps réel.

3.2 Description du fonctionnement interne

L'ordonnancement des tâches sous EDL est assuré par une fonction répondant au prototypage suivant :

void *Ats_schedule*(**void** **arg*) ;

Cette fonction est définie en tant que *handler de temps* (*time handler* en anglais) via les fonctions d'initialisation *IRQ_create()* et *IRQ_register()* (fonctions internes de la couche TCL [Gar05]), au moment du chargement dynamique du module d'ordonnancement dans le noyau. Comme dans le cas des composants de l'étagère "QoS", cette fonction handler est exécutée à chaque interruption du timer matériel 8254, dont la fréquence est définie par l'application utilisateur. L'argument d'entrée de la fonction est un pointeur sur une variable correspondant au temps courant de l'application temps réel. Celui-ci est décrit en termes du nombre de tops horloge écoulés depuis le lancement de l'application temps réel. C'est donc au sein de la fonction *Ats_schedule()* qu'est implémenté l'algorithme d'ordonnancement du serveur EDL.

Le comportement du composant est par ailleurs conditionné par la valeur d'une variable externe appelée *LOADED_SCHEDULER* qui prend différentes valeurs selon le module d'ordonnancement chargé pour la gestion des tâches périodiques (cf. Table VI.2).

Ordonnanceur chargé	EDF	RTO	BWP	RLP	RLP/T
LOADED_SCHEDULER	0	1	2	3	4

TAB. VI.2 – Valeur de la variable LOADED_SCHEDULER en fonction de l'ordonnanceur utilisé

L'initialisation de la variable *LOADED_SCHEDULER* est effectuée au sein des différents modules d'ordonnancement de tâches périodiques, sa valeur d'initialisation étant distincte d'un composant à l'autre.

3.3 Description de l'interface utilisateur

L'interface utilisateur des ordonnanceurs de l'étagère "Serveurs de tâches apériodiques" est décrite ci-dessous dans la Table VI.3.

Fonctions	Description
Ats_create()	création d'une nouvelle tâche temps réel
Ats_resume()	réveil d'une tâche apériodique
Ats_begin()	réveil simultané de l'ensemble des tâches périodiques
Ats_delete()	suppression d'une tâche temps réel

TAB. VI.3 – Interface utilisateur de l'étagère "Serveurs"

Nous précisons ci-après, les fonctionnalités de l'ensemble de ces fonctions, à travers la définition de leur prototypage respectif.

3.3.1 La fonction Ats_create()

La primitive *Ats_create()* crée une nouvelle tâche temps réel référencée par un descripteur de tâche t. Le prototypage complet de cette fonction est présentée ci-dessous :

```
int Ats_create (
  AtsTaskType *t,
  void (*fct) (AtsTaskType*),
  TimeType WCET,
  TimeType critical_delay,
  TimeType period,
  unsigned int max_skipvalue,
  TCLCreateType TCL_data) ;
```

t est un pointeur sur une structure du type *AtsTaskType* dont l'espace mémoire doit être fourni par l'application. *fct* correspond au point d'entrée de la fonction de la tâche. Les paramètres *WCET*, *critical_delay*, et *period*, représentent les caractéristiques temporelles de la tâche, à savoir sa durée d'exécution au pire-cas C_i, son délai critique D_i et sa période P_i respectivement. Dans le cas de la création d'une tâche apériodique, *period* doit être initialisé à la valeur 0. De plus, si la tâche apériodique est non critique, une valeur nulle doit également être assignée au paramètre *critical_delay*. *max_skipvalue* spécifie les pertes s_i autorisées au sens Skip-Over au niveau de la tâche. Dans le cas où la tâche est apériodique, la valeur de ce paramètre n'est jamais considérée. Enfin, *TCL_data* est une structure qui contient les données spécifiques au système d'exploitation, comme nous l'avons vu précédemment pour les fonctions de l'étagère "QoS"

En cas de succès, la valeur 0 est retournée. En cas d'échec de l'initialisation, une valeur négative est retournée :

– EINVAL : la structure de tâche pointée par la tâche est déjà utilisée,
– ENOMEM : les octets *stack size* ne peuvent pas être alloués pour la pile.

3.3.2 La fonction Ats_resume()

La primitive *Ats_resume()* permet de réveiller une tâche apériodique au temps courant. Son prototypage est le suivant :

int *Ats_resume*(QoSTaskType **t*) ;

QoS_resume permet le passage à l'état prêt de la tâche apériodique dont le descripteur de tâche *t* est fourni en entrée. En cas de succès, la valeur 0 est retournée. Dans le cas contraire, la valeur négative -1 est retournée pour signifier que la tâche a déjà été réveillée.

3.3.3 La fonction Ats_begin()

Pour que les serveurs de tâches apériodiques fonctionnent d'une manière correcte, il est nécessaire que toutes les tâches périodiques soient réveillées au même instant (configuration synchrone). Ceci est réalisée par l'appel à la fonction *Ats_begin()* dont le prototypage est fourni ci-dessous :

int *Ats_begin*(TimeType *release_time*) ;

Le paramètre d'entrée spécifie que toutes les tâches périodiques seront réveillées à la date absolue *release_time*. En cas de succès, la valeur 0 est renvoyée. Dans le cas contraire, une erreur d'allocation de mémoire est signalée par le renvoi de la valeur -1.

3.3.4 La fonction Ats_delete()

La primitive *Ats_delete()* supprime la tâche temps réel créée au préalable par la fonction *Ats_create()*. Son prototypage est le suivant :

int *Ats_delete*(QoSTaskType **t*) ;

t se réfère au pointeur du descripteur de la tâche. Si la tâche était en attente d'un sémaphore, celle-ci est retirée de la file d'attente. En cas de cas de succès, la valeur 0 est retournée. En cas d'échec, la valeur négative EINVAL est retournée pour signifier que le pointeur *t* ne se réfère pas à une tâche valide.

Les composants d'ordonnancement de l'étagère "Serveurs de tâches apériodiques", à savoir BG.o, TBS.o, et EDL.o, sont tous situés physiquement dans le répertoire $RTAI_DIR/modules/Ats. Leur chargement et déchargement dynamiques dans le noyau s'effectuent de manière classique sous RTAI via les commandes *insmod* et *rmmod* respectivement.

4 Evaluation de performances

4.1 Tests des composants au niveau fonctionnel

Les tests fonctionnels des composants ont été effectués grâce à l'outil de traçage d'événements Linux Trace Toolkit (LTT). LTT est un outil de visualisation développé par la société Opersys[5]. Son code source est sous licence GPL. LTT réalise une trace en-ligne et une reconstruction hors-ligne du comportement dynamique du noyau Linux/RTAI. Parmi les événements capturés par LTT, nous pouvons citer les changements de contexte, les interruptions matérielles et logicielles ou bien les accès disques. L'outil présente par ailleurs des statistiques relatives au système sous la forme de mesures de performances temporelles (pourcentage de temps CPU alloué aux différentes tâches, durée d'exécution des tâches, temps moyen entre deux exécutions d'une tâche, etc.). L'outil offre également une vue graphique de l'ordonnancement des tâches de l'application temps réel. LTT permet ainsi de décrire de manière précise le comportement du système avec un coût de mesure (overhead) relativement faible [RPT04].

4.1.1 Test fonctionnel de RTO, BWP, RLP et RLP/T

Considérons l'ensemble $\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\}$ utilisé tout au long du chapitre III pour illustrer le comportement des algorithmes RTO, BWP, RLP et RLP/T. Les caractéristiques temporelles des tâches sont rappelées ci-dessous dans la table VI.4. Les tâches possèdent un paramètre de pertes uniforme $s_i = 2$ et le facteur d'utilisation équivalent du processeur intégrant les pertes est égal à $U_p^* = 75\%$, garantissant l'ordonnançabilité de la configuration (au sens des exécutions des instances obligatoires).

Task	T_1	T_2	T_3	T_4	T_5
c_i	3	4	1	7	2
p_i	30	20	15	12	10

TAB. VI.4 – Configuration de tâches pour le test des composants "QoS"

Le programme C implémenté sous Linux/RTAI pour le test des composants de l'étagère "QoS" est présenté ci-dessous. Il est important de noter que ce programme est utilisable indifféremment sous RTO, BWP, RLP et RLP/T. La distinction est effectuée au moment du chargement dynamique du module lorsque l'utilisateur sélectionne l'ordonnanceur qu'il souhaite utiliser pour son application.

```
/*——— Entêtes des composants nécessaires ———*/
#include <TCL.h>
#include <QoS.h>
#include <simul.h>
```

[5]http ://www.opersys.com

```
/*——- Période de l'horloge du Timer (10ms) ——-*/
#define TIMERTICKS 10000000
/*————— Déclarations des tâches QoS ————-*/
QoSTaskType T0;
QoSTaskType T1;
QoSTaskType T2;
QoSTaskType T3;
QoSTaskType T4;
/*————- Description du code des tâches QoS ——*/
void CodeT0() {simul.wait(3);}
void CodeT1() {simul.wait(4);}
void CodeT2() {simul.wait(1);}
void CodeT3() {simul.wait(7);}
void CodeT4() {simul.wait(2);}
/*—— Lancement de l'application temps réel ——-*/
int init_module(void)
{
  TCLCreateType create={0, 2000, 0, 0};
  /********* Initialisation des tâches QoS **********/
  QoS_create(&T0, CodeT0, 3, 30, 30, 2, create);
  QoS_create(&T1, CodeT1, 4, 20, 20, 2, create);
  QoS_create(&T2, CodeT2, 1, 15, 15, 2, create);
  QoS_create(&T3, CodeT3, 7, 12, 12, 2, create);
  QoS_create(&T4, CodeT4, 2, 10, 10, 2, create);
  /************* Réveil des tâches QoS **************/
  QoS_resume(&T4, 100);
  QoS_resume(&T3, 100);
  QoS_resume(&T2, 100);
  QoS_resume(&T1, 100);
  QoS_resume(&T0, 100);
  /*********** Passage en mode temps réel **********/
  TCL.begin(TIMERTICKS, 2000);
  return 0;
}
/*——- Arrêt de l'application temps réel ———*/
void cleanup_module(void)
{
  /**********Suppression des tâches QoS**********/
  QoS_delete(&T4);
  QoS_delete(&T3);
  QoS_delete(&T2);
  QoS_delete(&T1);
  QoS_delete(&T0);
```

```
  /********** Retour du mode temps réel **********/
  TCL.end() ;
}
```

Comme nous pouvons le voir, le code de l'application se découpe en plusieurs éléments distincts qu'il convient à présent d'expliciter. En premier lieu, nous pouvons observer la déclaration des fichiers d'entêtes relatifs à la couche TCL, aux ordonnanceurs de l'étagère "QoS" et au composant "simul" (ce dernier décrit dans [Gar05] est utilisé uniquement à des fins de test, il permet de simuler l'exécution des tâches). Ensuite, la constante TIMERTICKS définit en nanosecondes la fréquence du timer utilisée pour l'ordonnancement. Dans notre cas, nous l'avons initialisée de manière à ce qu'un top timer soit généré toutes les 10ms. Puis, les structures de données des tâches sont déclarées pour chacune des 5 tâches de l'application.

Le programme présente ensuite le code des 5 tâches applicatives, T_0, T_1, T_2, T_3, et T_4. La primitive *simul.wait(x)* signifie que la tâche s'exécute pendant x unités de temps. Rappelons que l'unité de temps est définie par TIMERTICKS (ici, 1 unité de temps = 10ms).

En dernier lieu, nous voyons apparaître les fonctions de contrôle du module applicatif en lui-même : *init_module()* et *cleanup_module()*. La fonction *init_module()* correspond à la fonction de lancement de l'application temps réel. Elle prépare l'invocation des fonctions du module de l'application. Elle est elle-même invoquée par la commande *insmod*. Elle regroupe la création et le réveil des différentes tâches par les primitives *QoS_create()* et *QoS_resume()* respectivement. L'appel à la primitive *TCL.begin()* permet par ailleurs le passage en mode temps réel. Le premier paramètre de cette fonction correspond à la fréquence du timer définie au début du programme. Le second représente l'initialisation du chien de garde de l'application (*watchdog* en anglais). Celui-ci permet de régler la durée maximale pendant laquelle une tâche peut s'exécuter (ici 2000*TIMERTICKS, soit 20s). Lorsque la durée d'exécution d'une tâche devient supérieure à ce paramètre, dans le cas d'un problème de boucle infinie par exemple, l'application temps réel est immédiatement interrompue.

La fonction *cleanup_module()* invoquée par la commande *rmmod*, regroupe quant à elle, la destruction des tâches et la libération de toutes les ressources système allouées durant la vie du module applicatif. La destruction des tâches est assurée par l'appel à la fonction *QoS_delete()*. La primitive temps réel *TCL.end()* permet de quitter le mode temps réel. Notons que la suppression des tâches est optionnelle sous CLEOPATRE dans la mesure où la primitive TCL.end vérifie l'intégrité du système d'exploitation en détruisant si besoin toutes les tâches et mécanismes de synchronisation latents, suite au déchargement de l'application dans le noyau.

Les figures VI.4, VI.5, VI.6, et VI.7, présentent la visualisation sous LTT des résultats d'exécution de l'application, pour les composants d'ordonnancement RTO, BWP, RLP et RLP/T respectivement. Sur la partie gauche des graphiques, l'outil présente la liste de l'ensemble des entités ayant existé durant la trace, *RTAI* représentant la couche de

contrôle de l'ordonnancement. Sur l'espace de droite, les lignes horizontales indiquent le temps écoulé à exécuter le code appartenant à la tâche située à la même hauteur. Les transitions verticales indiquent les instants où la ressource processeur commute d'une entité à une autre.

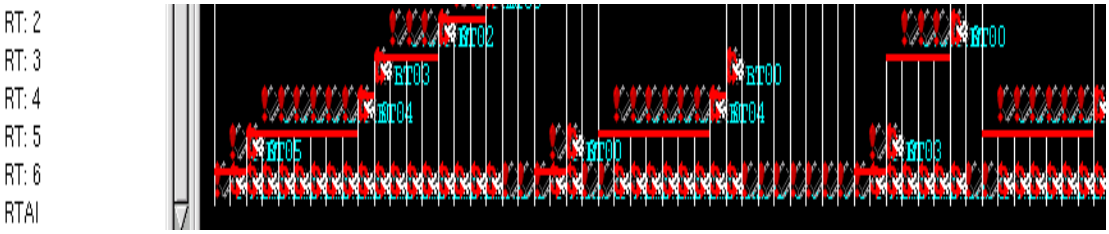

FIG. VI.4 – Comportement de RTO visualisé sous LTT

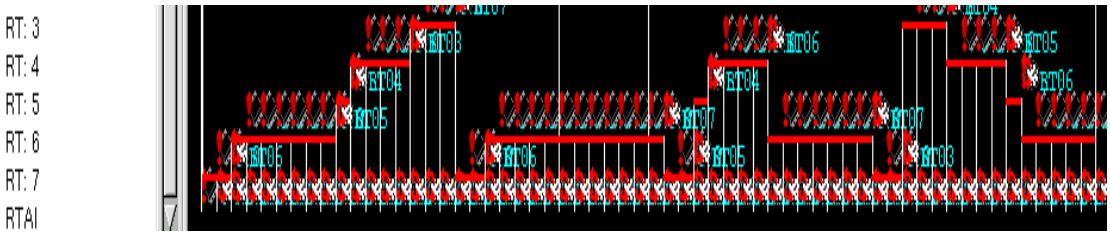

FIG. VI.5 – Comportement de BWP visualisé sous LTT

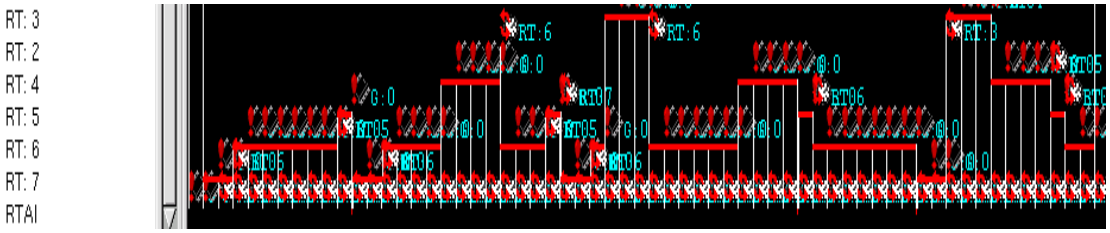

FIG. VI.6 – Comportement de RLP visualisé sous LTT

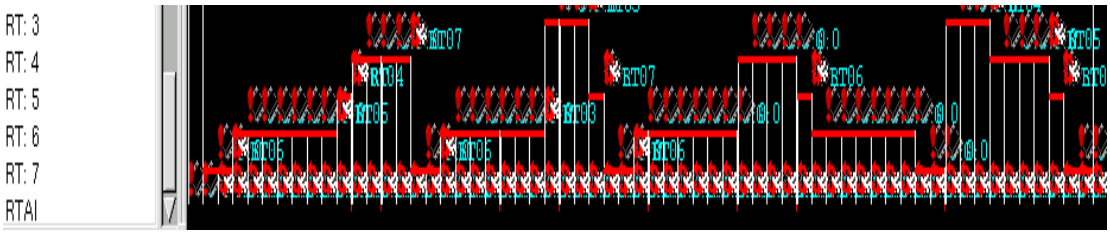

FIG. VI.7 – Comportement de RLP/T visualisé sous LTT

Les résultats visualisés sous LTT nous montrent que l'ordonnancement réalisé par les composants CLEOPATRE de l'étagère "QoS", à savoir RTO, BWP, RLP et RLP/T, est conforme à leur comportement théorique précédemment illustré dans le chapitre III avec la même configuration de tâches (cf. figures III.1, III.2, III.11, et III.13 respectivement).

4.1.2 Test fonctionnel de EDL

Considérons l'ensemble $\mathcal{T} = \{T_1, T_2\}$ utilisé tout au long du chapitre IV pour illustrer le comportement du serveur EDL avec les algorithmes d'ordonnancement RTO, BWP, RLP et RLP/T. Les caractéristiques temporelles des tâches sont rappelées ci-dessous dans la table VI.5. Les tâches possèdent un paramètre de pertes uniforme $s_i = 2$. Cette configuration de tâches est ordonnançable car le facteur d'utilisation équivalent du processeur intégrant les pertes est tel que $U_p^* = 90\%$.

Task	T_1	T_2
c_i	6	3
p_i	10	6

TAB. VI.5 – Configuration de tâches pour le test du composant EDL

Le programme C implémenté pour le test du composant EDL de l'étagère "Serveurs de tâches apériodiques" est présenté ci-dessous.

```
/*——— Entêtes des composants nécessaires ———*/
#include <TCL.h>
#include <ats.h>
#include <simul.h>
/*——- Période de l'horloge du Timer (10ms) ——-*/
#define TIMERTICKS 10000000
#define Nb_events 1
/*———— Déclarations des tâches ————*/
AtsTaskType T1 ;
AtsTaskType T2 ;
AtsTaskType R1 ;
/*—- simulation de l'occurrence de R1 à t=120 ——*/
EventType event[Nb_events]={{120,0}} ;
void event0(){
  ats_wakeup(&R1) ;
}
/*———— Description du code des tâches ———*/
void CodeT1() {simul.wait(6) ;}
void CodeT2() {simul.wait(3) ;}
void CodeR1() {simul.wait(10) ;}
```

```
/*—— Lancement de l'application temps réel ——-*/
int init_module(void)
{
  TCLCreateType create={0, 2000, 0, 0};
  /********** Initialisation des tâches ***********/
  Ats_create(&T1, CodeT1, 10, 10, 6, 2, create);
  Ats_create(&T2, CodeT2, 6, 6, 3, 2, create);
  Ats_create(&R1, CodeR1, 0, 0, 10, 0, create);
  /**** Réveil des tâches périodiques à t=100*****/
  Ats_begin(100)
  /***** Initialisation de la liste d'événements****/
  simul.create(0, event0);
  simul.event(Nb_events, event);
  /*********** Passage en mode temps réel ********/
  TCL.begin(TIMERTICKS, 2000);
  return 0;
}
/*——- Arrêt de l'application temps réel ———*/
void cleanup_module(void)
{
  /***********Suppression des tâches *************/
  Ats_destroy(&T1);
  Ats_destroy(&T2);
  Ats_destroy(&R1);
  /********** Retour du mode temps réel **********/
  TCL.end();
}
```

Comme dans le cas du test des composants de l'étagère "QoS", le code de l'application comporte plusieurs parties repérables. Nous pouvons observer ici la déclaration du fichier d'entêtes *ats.h* relatif à l'utilisation des composants de l'étagères "Serveurs de tâches apériodiques". Ensuite, en plus de la constante TIMERTICKS, nous remarquons une définition supplémentaire correspondant au nombre d'événements apériodiques simulés par le composant "simul.o". Dans notre cas, cette constante est égale à 1 et correspond à la simulation de l'occurrence d'une seule tâche apériodique.

Puis, les structures de données des tâches (de type *AtsTaskType* cette fois-ci) sont déclarées pour l'ensemble des tâches de l'application. Le programme présente ensuite le code des 3 tâches applicatives, à savoir les tâches périodiques T_1 et T_2, et la tâche apériodique R_1.

Enfin, nous voyons apparaître les fonctions de lancement et d'arrêt de l'application temps réel. La fonction *init_module()* regroupe la création de l'ensemble des tâches ainsi que le réveil simultané des deux tâches périodiques par l'appel des primitives *Ats_create()* et *Ats_begin()* respectivement. Après, la fonction procède à l'initialisation de la liste des

événements apériodiques simulés, en associant notamment l'événement n°0 à la fonction *event0()* chargée de réveiller la tâche apériodique R_1 à l'aide de la primitive *Ats_resume()*. Puis, on retrouve ensuite l'appel à la primitive *TCL.begin()* permettant le passage en mode temps réel.

Dans la fonction *cleanup_module()*, la destruction des tâches est assurée par l'appel à la fonction *Ats_destroy()*. La primitive temps réel *TCL.end()* permet de quitter le mode temps réel. Notons que la suppression des tâches est également optionnelle ici, la couche TCL se chargeant toujours de vérifier l'intégrité du système.

Les figures VI.8, VI.9, VI.10, et VI.11, présentent la visualisation sous LTT des résultats d'exécution de l'application, pour les modèles d'ordonnancement EDL-RTO, EDL-BWP, EDL-RLP et EDL-RLP/T respectivement. Notons que dans les exemples présentés et repris du chapitre IV, la tâche apériodique survient à l'instant $t = 120$ dans le cas de EDL-RTO, et à l'instant $t = 112$ dans le cas des modèles EDL-BWP, EDL-RLP et EDL-RLP/T. La tâche apériodique R_1 apparaît en première position dans la liste des tâches présentée sur la partie gauche du graphique, apparaissent ensuite les tâches périodiques T_1 et T_2.

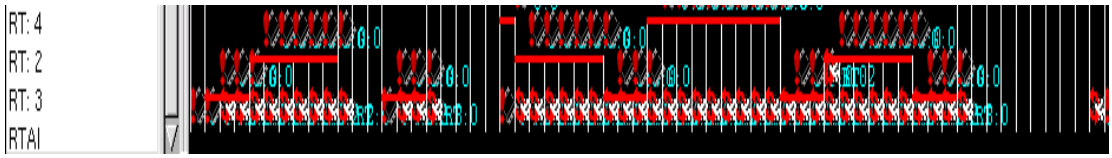

FIG. VI.8 – Comportement de EDL-RTO visualisé sous LTT

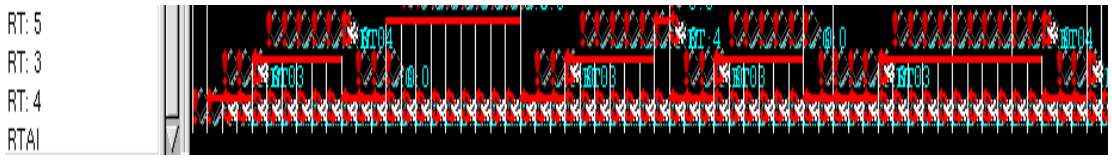

FIG. VI.9 – Comportement de EDL-BWP visualisé sous LTT

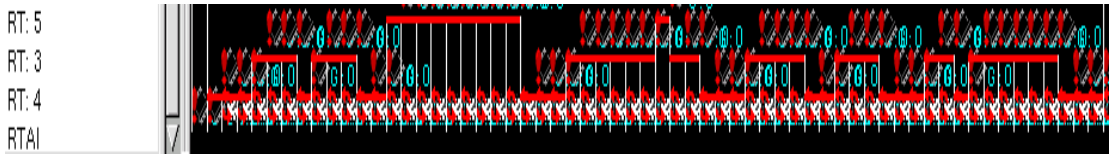

FIG. VI.10 – Comportement de EDL-RLP visualisé sous LTT

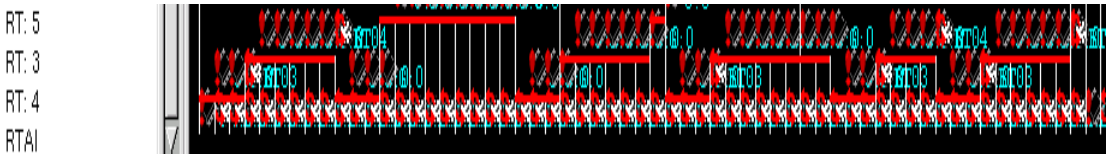

FIG. VI.11 – Comportement de EDL-RLP/T visualisé sous LTT

Les résultats visualisés sous LTT nous montrent que l'ordonnancement réalisé par le composant EDL, est conforme à son comportement théorique précédemment illustré dans le chapitre IV (cf. figures IV.7, IV.14, IV.17, et IV.20 respectivement).

4.2 Evaluation des empreintes mémoire et disque

Des précautions doivent être prises concernant l'utilisation d'un ordonnanceur dans le cadre d'applications temps réel et/ou embarquées. Au niveau des applications temps réel, nous avons vu qu'il est obligatoire de respecter toutes les échéances requises pour les tâches, tout en fournissant un résultat correct d'un point de vue logique. Ceci implique d'évaluer la durée d'exécution de l'ordonnanceur utilisé pour le contrôle de l'application, de manière à vérifier la non-interférence de l'algorithme d'ordonnancement vis-à-vis de l'exécution des tâches applicatives. Les applications temps réel embarquées présentent quant à elle des contraintes d'encombrement et d'occupation mémoire du système sur la cible.

Les empreintes mémoire et disque des différents composants intégrés sous CLEOPATRE sont résumées dans la Table VI.6.

Etagère "QoS"	Hard disk size (KB)	Memory size (KB)
RTO	3.2	2.3
BWP	4.1	3.2
RLP	9.7	7.6
RLP/T	13.3	10.8
Etagère "Serveurs"	Hard disk size (KB)	Memory size (KB)
EDL	19.7	15.2
Modules obligatoires	Hard disk size (KB)	Memory size (KB)
RTAI	27.2	23
TCL	34.8	27.1

TAB. VI.6 – Footprints of QoS components

L'empreinte minimale d'une application utilisant un ordonnanceur de l'étagère "QoS" est de 52.4 KB en mémoire (65.2 KB sur le disque), ce qui comprend le chargement de RTAI, de l'interface TCL et de l'ordonnanceur RTO. En contrepartie, l'empreinte

maximale est atteinte si l'ordonnanceur RLP/T est chargé : 60.9 KB en mémoire (75.3 KB sur le disque).

Dans les deux cas, nous pouvons voir que les tailles observées peuvent aisément être supportées par un système embarqué muni d'une mémoire flash par exemple.

4.3 Evaluation des overheads d'ordonnancement

Quelques expériences complémentaires ont été effectuées de manière à évaluer quantitativement les overheads respectifs des composants de l'étagère "QoS". Les tests mis en œuvre ont consisté à mesurer l'overhead induit par l'ordonnancement de configurations dont le nombre de tâches varie (de 0 à 20). Les périodes des tâches ont été considérées harmoniques entre elles et de *ppcm* égal à 3360. Les mesures ont été effectuées sur une durée de 33.6 secondes (l'ordonnancement des tâches étant assuré toutes les 10ms), sur un système doté d'un processeur Pentium II cadencé à 400 MHz, et possédant 384Mo de RAM.

Les résultats de mesures obtenus pour les composants d'ordonnancement de l'étagère "QoS" (cf. figure VI.12) indiquent la quantité de temps dispensée dans l'ordonnancement des tâches.

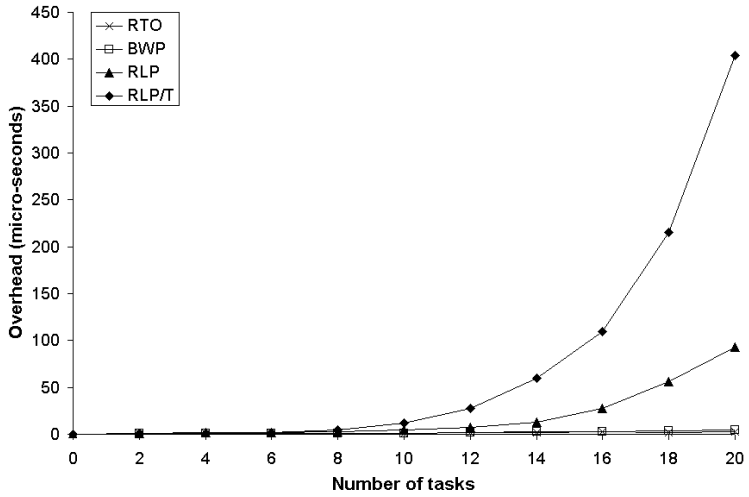

FIG. VI.12 – Overhead dynamique des ordonnanceurs de l'étagère "QoS"

Les courbes nous montrent que l'overhead des ordonnanceurs de l'étagère "QoS" varie avec le nombre de tâches installées. Le faible overhead d'exécution de l'ordonnanceur RTO est directement lié à la faible complexité de son algorithme. Notons que l'overhead de BWP est pratiquement aussi faible que celui de RTO. Par contre, la courbe obtenue pour RLP souligne la quantité de temps assez importante requise pour construire la séquence EDL (calcul des temps creux qui d'ailleurs n'est effectué que lorsqu'une tâche bleue survient alors qu'il n'y en avait aucune auparavant dans le système). L'overhead observé pour RLP/T augmente quant à lui fortement en fonction du nombre de tâches

présentes dans le système. Ceci s'explique par le fait que la détermination des temps creux sous RLP/T est effectuée à chaque fois qu'une tâche bleue survient ou se termine avec succès. Pour un total de 6 tâches périodiques, RLP/T atteint ainsi l'overhead observé sous RTO pour 20 tâches périodiques. Nous pouvons donc en conclure, qu'en pratique, RLP/T s'avère intéressant à utiliser pour des applications impliquant moins de 15 tâches.

5 Synthèse

A la fin de ce chapitre, la librairie CLEOPATRE présente une nouvelle étagère de composants innovants relatifs à l'ordonnancement sous contraintes de QoS, ainsi qu'un nouveau composant de service optimal des tâches apériodiques. Le détail des composants CLEOPATRE disponibles dans chaque étagère, est présenté sur la figure VI.13.

Serveurs apériodiques	*Ordonnanceurs*	*Protocoles de synchronisation*	*Tolérance aux fautes*	*Gestion de la surcharge*	*Qualité de Service*
- BG - TBS - EDL	- DM - EDF	- FIFO - SPP - PIP - PCP - DPCP	- Mécanisme à Echéance	- Calcul imprécis	- RTO - BWP - RLP - RLP/T

FIG. VI.13 – Description de la librairie CLEOPATRE étendue

6 Conclusion

Dans ce chapitre, nous avons traité de l'intégration sous Linux temps réel, des ordonnanceurs RTO, BWP, RLP et RLP/T, et de celle du serveur de tâches apériodiques optimal EDL. Après avoir présenté le contexte du travail mené, notamment par la présentation du projet CLEOPATRE et du système d'exploitation temps réel Linux/RTAI, nous nous sommes intéressés, pour chacun des composants intégrés, à la description de son fonctionnement et de ses interfaces utilisateur. Ensuite, nous avons validé leur comportement fonctionnel sous l'outil de traçage LTT. Puis, nous avons étudié les caractéristiques externes des composants (empreinte mémoire et disque) et nous avons évalué leurs performances temporelles en termes d'overhead d'exécution.

Grâce à la librairie CLEOPATRE étendue, la communauté temps réel dispose à présent de nouvelles solutions temps réel plus flexibles dédiées au développement d'applications temps réel présentant des contraintes temporelles strictes et ou fermes, et pouvant conduire à des situations de surcharge de traitement.

Conclusion générale

Ce rapport a proposé des solutions au problème de l'ordonnancement dynamique de tâches présentant des contraintes de qualité de service. Le travail effectué s'inscrit dans une perspective de gestion des cas de surcharge dans les systèmes centralisés monoprocesseur. Cette recherche vise à répondre aux besoins des nouvelles applications temps réel pour lesquelles l'ordonnancement TRCS semble trop restrictif.

Après avoir mené une analyse des concepts relatifs à l'ordonnancement dans les systèmes temps réel et une synthèse des modèles de résolution dédiés à l'ordonnancement de systèmes surchargés, nous avons dégagé des idées clés qui nous ont servi dans les développements réalisés.

Dans un premier temps, nous avons proposé les résultats théoriques relatifs à deux nouveaux ordonnanceurs monoprocesseur, RLP et RLP/T, issus de la fusion de deux approches existantes que sont le modèle Skip-Over et le serveur à priorités dynamiques EDL. Ces nouveaux algorithmes se sont montrés capables d'engendrer un ordonnancement fiable et performant en termes de QoS effective. Les ordonnanceurs de base RTO et BWP du modèle Skip-Over ont ainsi été améliorés.

L'objectif poursuivi ensuite consistait à fournir un modèle d'ordonnancement flexible visant à minimiser le temps de réponse des requêtes apériodiques non critiques, et à maximiser le taux d'acceptation des requêtes apériodiques critiques, en tirant profit des pertes autorisées au niveau des tâches périodiques. C'est pourquoi nous nous sommes intéressés à l'adaptation des serveurs de tâches apériodiques BG, TBS, TB* et EDL aux modèles de tâches RTO et BWP. Puis, la prédominance des performances du serveur EDL sur les autres modèles ayant été soulignée à l'aide de simulations, nous nous sommes focalisés sur l'utilisation conjointe de ce serveur optimal avec les ordonnanceurs RLP et RLP/T. Les simulations mises en œuvre nous ont permis de confirmer la supériorité des performances de EDL-RLP/T sur l'ensemble des modèles étudiés et ce, aussi bien au niveau de la QoS observée au niveau des tâches périodiques, qu'au niveau du service offert pour les tâches apériodiques.

Nous avons ensuite repris deux notions, la stabilité et la robustesse précédemment utilisées pour gérer la surcharge dans un système tolérant aux fautes. Sur la base de ces définitions, nous avons pu souligner le manque de stabilité, au sens de l'équilibre des taux de respect individuels des tâches, des ordonnanceurs BWP, RLP et RLP/T pour lesquels l'ordonnancement des instances de tâches bleues est assuré par l'algorithme ED. C'est pourquoi nous avons présenté deux nouvelles heuristiques pour l'ordonnancement des instances bleues, à savoir les variantes LF et MS. Celles-ci se sont révélées améliorer de façon significative la stabilité des ordonnanceurs précédents.

Enfin, après avoir proposé, simulé et validé formellement les ordonnanceurs développés dans les chapitres 3 et 4, nous avons procédé à leur intégration sous Linux temps réel, de manière à démontrer leur faisabilité. Les modules RTO, BWP, RLP, RLP/T et EDL sont ainsi venus enrichir une librairie de composants open-source existante, appelée CLEOPATRE. Le comportement fonctionnel des nouveaux composants a été validé à l'aide d'applications de test basiques, et des mesures liées à leurs coûts de mise en œuvre et d'implémentation ont été réalisées.

Les travaux de recherche futurs reposent sur l'extension des résultats obtenus avec le modèle Skip-Over à d'autres modèles de tâches destinés aux applications temps-réel fermes, en particulier le modèle (m,k)-firm. Par ailleurs, ce document traitant uniquement de l'ordonnancement centralisé monoprocesseur sous des contraintes de QoS, les travaux futurs vont s'orienter vers la conception d'ordonnanceurs multiprocesseurs répondant aux mêmes besoins. En effet, ceux-ci sont de plus en plus employés dans un grand nombre d'applications de traitement informatique, non seulement dans le domaine du traitement d'information, mais également pour le contrôle de robots et la simulation temps réel de systèmes dynamiques, du fait qu'ils procurent une excellente efficacité de coût.

Bibliographie

[AB90] Audsley, N., Burns, A., *Deadline-monotonic scheduling.* Technical Report YCS146, Department of Computer Science, University of York, UK, 1990.

[AB98] Abeni, L., Buttazzo, G., *Integrating Multimedia Applications in Hardware Real-Time Systems.* In Proceedings of the 19th IEEE Real-Time Systems Symposium, 4-14, 1998.

[ABR+91] Audsley, N.-C., Burns, A., Richardson, M., Wellins, A.-J., *Hard Real-Time Scheduling : The Deadline Monotonic Approach.* In Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software, Atalanta, 1991.

[ABR+93] Audsley, N.-C., Burns, A., Richardson, M., Tindell, K., Wellins, A., *Applying new scheduling theory to static priority preemptive scheduling.* Software Engineering Journal, 8(5) : 284-292, 1993.

[AHU74] Aho, A.-V., Hopcroft, J.-E., Ullman, J.-D., *The design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass, 1974.

[Bar95] Baruah, S.-K., *Fairness in Periodic Real-Time Scheduling.* In Proceedings of the 16th IEEE Real-Time Systems Symposium, 200-209, 1995.

[Bat98] Bates, I.-J., *Scheduling and Timing Analysis for Safety Critical Real-Time Systems.* PhD thesis, University of York, 1998.

[Bla76] Blazewicz, J., *Deadline scheduling of tasks - A survey.* Foundations of control engineering, 1(4), 1976.

[Bou91] Bouchentouf, T., *Ordonnancement sous contraintes de précédence dans les systèmes temps-réel.* PhD thesis, Université de Nantes, 1991.

[But97] Buttazzo, G.-C., *Hard Real-Time Computing Systems.* Kluwer academic, 1997.

[BA02] Buttazzo, G., Abeni, L., *Adaptive Workload Management through Elastic Scheduling.* Real-Time Systems, 23(1-2) : 7-24, 2002.

[BB97] Bernat, G., Burns, A., *Combining (n,m)-hard deadlines and dual-priority scheduling.* In Proceedings of th 18th IEEE Real-Time Systems Symposium (RTSS'97), pp 46-57, 1997.

[BB01] Banachowski, S., Brandt, S., *The BEST desktop soft real-time scheduler*, In Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01), pp 9-12, 2001.

[BB02] Banachowski, S., Brandt, S., *The BEST scheduler for integrated processing of best-effort and soft real-time processes*, In Proceedings of Multimedia Computing and Networking 2002 (MMCN'02), pp 46-60, 2002.

[BBB01] 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.

[BBL01] Bernat, G., Burns, A., Llamosi, A., *Weakly-hard real-time systems*. In IEEE Transactions on Computers, Vol. 50, No. 4 pp 308-321, 2001.

[BD99] Bonnet, C., Demeure, I., *Introduction aux systèmes temps réel*. Paris : Hermes Sciences Publications, 207p., 1999.

[BH93] Baruah, S., Haritsa, J., *ROBUST A Hardware Solution to Real-Time Overload*. In Proceedings of the 13th ACM SIGMETRICS Conference, pp 207-216, Santa Clara, California, USA, 1993.

[BHR90] Baruah, S.-K., Howell, R.-R., Rosier, L.-E., *Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor*. Real-Time Systems, 2, 1990.

[BHR93] Baruah, S.-K., Howell, R.-R., Rosier, L.-E., *Feasibility problems for recurring tasks on one processor*. Theoretical Computer Science, 118(1) : 3-20, 1993.

[BKM+91] Baruah, S., Koren, G., Mishra, B., Ragunathan, A., Rosier, L., Shasha, D., *On-line Scheduling in the Presence of Overload*. In Proceedings of the 32nd Annual IEEE Symposium Foudations of Computer Science, 101-110, San Juan, Puerto Rico, 1991.

[BLA98] Buttazzo, G., Lipari, G., Abeni, L., *Elastic Task Model for Adaptive Rate Control*, In Proceedings of the IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain, 1998.

[BS93] Buttazzo, G., Stankovic, J. *RED Robust Earliest Deadline Scheduling*. In Proceedings of The 3rd International Workshop on Responsive Computing Systems, Austin, USA, 1993.

[BS94] 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.

[BS99] 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.

[BSS95] Buttazzo, G.-C., Spuri, M., Sensini, F., *Value vs. deadline Scheduling in Overload Conditions*. In Proceedings of the 15th Real-Time Systems Symposium, Pisa, Italy, 1995.

[CAM79] Campbell, R.-H., Horton, K.-H., Belford, G.-G., *Simulations of a fault-tolerant deadline mechanism*. Digest of papers FTCS-9, pp 95-101, 1979.

[CB97] Caccamo, M., Buttazzo, G., *Exploiting skips in periodic tasks for enhancing aperiodic responsivess.* In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97), 1997.

[CB98] Caccamo, M., Buttazzo, G., *Optimal Scheduling for Fault-Tolerant and Firm Real-Time Systems.* In Proceedings of the IEEE Real-Time Computing Systems and Applications, 1998.

[CC89] Chetto, H., Chetto, M., *Some results of the earliest deadline scheduling algorithm.* IEEE Transactions on Software Engineering, 15(10) : 1261-1269, 1989.

[CC90] Chetto, H., Chetto, M., *A feasibility test for scheduling tasks in a distributed hard real-time system.* APII, 239-252, 1990.

[CCE88] Chetto, H., Chetto, M., Elyounsi, N., *On Designing a Real-Time System Exempt of any Timing Failures.* In Proceedings of the 8th IFAC Workshop on Distributed Computer Control Systems, Vitznau, Suisse, 1988.

[CDK+00] Cottet, F., Delacroix, J., Kaiser C., Mammeri, Z., *Ordonnancement temps-réel.* Paris : Hermes Sciences Publications, 2000.

[CEN+02] Chrobak, M., Epstein, L., Noga, J. Sgall, J., van Stee, R., Tichy, T., Vakhania, N., *Preemptive Scheduling in Overloaded Systems.* In Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP2002), pp 800-811, Málaga, Spain, 2002.

[Cla90] Clark, R.-K., *Scheduling Dependent Real-Time Activities*, Ph.D. Thesis. 1990.

[Clo88] Clouard, A., *Implémentation et évaluation d'un mécanisme de tolérance aux fautes pour les systèmes temps réel.* Rapport de DEA, Ecole Nationale Supérieure de Mécanique, Nantes, France, 1988.

[CLB99] Caccamo, M., Lipari, G., Butazzo, G., *Sharing resources among periodic and aperiodic tasks with dynamic deadlines.* In Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.

[CLL90] Chung, J.-Y., Liu, J.-W.-S., Lin, K., *Scheduling periodic jobs that allow imprecise results.* In IEEE Transactions on Computers, 39(9) : 1156-1174, 1990.

[CS93] Chetto, H., Silly, M., *Dynamic scheduling with laxities in a hard real-time system.* LAN Research Report, 1993.

[Dar03] Dardenne, M., *Ordonnancement à réservations dans un Linux Temps-Réel.* Mémoire Ingénieur Civil en Informatique, Département d'ingénierie informatique, Université catholique de Louvain, 2003.

[Dec02] Decotigny, D., *Introduction à l'ordonnancement dans les systèmes temps-réel.* Rapport bibliographique, IRISA, Rennes, 2002.

[Del94] Delacroix, J., *Un contrôleur d'ordonnancement temps-réel pour la stabilité de Earliest Deadline en surcharge : le Régisseur.* PhD Thesis, Université Pierre et Marie Curie, 1994.

[Der74] Dertouzos, M.-L., *Control Robotics : the Procedural Control of Physical Processes.* Information Processing 74, North-Holland Publishing Compagny, 1974.

[DM89] Dertouzos, M.-L., Mok, A.-K., *Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks*. IEEE Trans. Software Eng. 15(12) : 1497-1506, 1989.

[DP91] Dorseuil, A., Pillot, P., *Le temps-réel en milieu industriel, concepts, environnements, multitâches*. Paris, Bordas, 296p., 1991.

[DTB93] Davis, R.-I., Tindell, K.-W., Burns, A., *Scheduling slack time in fixed priority preemptive systems*. In Proceedings of the Real-Time Systems Symposium, 222-231, 1993.

[DW95] Davis, R., Wellings, A., *Dual priority scheduling*. In Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), pp 100-109, Pisa, Italy, 1995.

[Ely91] Elyounsi, N., *Ordonnancement et reconfiguration dynamique dans un système temps-réel réparti à contraintes strictes*. Thèse de Doctorat, Ecole Supérieure de Mécanique, Université de Nantes, 1991.

[Gar05] Garcia-Fernandez, T., *Conception et développement de composants pour logiciels temps-réel embarqués*. PhD Thesis, Université de Nantes, 2005.

[GB95] Ghazalie, T.-M., Baker, T.-P., *Aperiodic servers in a deadline scheduling environment*. The Journal of Real-Time Systems, 9 : 21-36, 1995.

[GLL+79] Graham, R.-L., Lawler, E.-L., Lenstra, J.-K., Rinnoy Kan, A.-H.-G., *Optimization and Approximation in deterministic sequencing and scheduling*. Ann. Discrete. Math. 5, 287-326, 1979.

[GMR95] George, L., Muhlethaler, P., Rivierre N., *Optimality and Non-Preemptive Real-Time Scheduling Revisited*. Research Report RR-2516, INRIA, Le Chesnay Cedex, France, 1995.

[GRS96] George, L., Rivierre, N., Spuri M., *Preemptive and Non-Premmptive Real-Time Uni-Processor Scheduling*. Research Report RR-2966, INRIA, Le Chesnay Cedex, France, 1996.

[Hor74] Horn, W.-A., *Some simple scheduling algorithms*. Naval Res-Logist. Quart. 21, 177-185, 1974.

[HLL+03] Hu, H.-S., Liu, D., Lemmon, M.-D., Ling, Q., *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.

[HLW92] Ho, K.-I.-J., Leung, J.-Y.-T., Wei, W.-D., *Minimizing maximum weighted error of imprecise computation tasks*, Technical Report, Dep. Of Computer Science and Engineering. Univ. of Nebraska, USA, 1992.

[HR95] Hamdaoui, M., Ramanathan, P., *A dynamic priority assignment technique for streams with (m,k)-firm deadlines*. IEEE Transactions on Computers, 44(4) : 1443-1451, 1995.

[HR97] Hamdaoui, M., Ramanathan, P., *Evaluating Dynamic Failure Probability for Streams with (m,k)-Firm Deadlines*. IEEE Transactions on Computers, 46(12) : 1325-1337, 1997.

[HS76] Horowitz, E., Sahni, S., *Exact and Approximate algorithms for non identical processors.* JACM, 23 : 317-327, 1976.

[HTS00] Hansson, J., Thuresson, M., Son, S.-H., *Imprecise Task Scheduling and Overload Management using OR-ULD.* In the International Conference on Real-Time Computing Systems and Applications, Cheju Island, Korea, 2000.

[ISO95] ISO ITU-T Recommendation X.902 — ISO/IEC 10746-2 : Information technology - Open Distribution Processing - Reference model : Foudations. Genève, ISO, 20p., 1995.

[ISO98] ISO ITU-T Recommendation X.641 — ISO/IEC 13236 : Information technology - Quality of Service -Framework. Genève, ISO, 48p., 1998.

[Jac55] Jackson, J.-R., *Scheduling a production line to minimize maximum tardiness.* Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[JG99] Jeffay, K., Godard, S., *A theory of rate-based execution.* In Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona (USA), pp 304-314, 1999.

[JLT85] Jensen, E.-D., Locke, C.-D., Tokuda, H., *A time-driven scheduling model for real-time operating systems.* In Proceedings of the IEEE Real-Time Systems Symposium, 112-122, 1985.

[JNC+89] Jensen, E.-D., Northcutt, J.-D., Clark, R.-K., Shipman, S.-E., Reynolds, F.-D., Maynard, D.-P., Loepfere, K.-P., *Alpha : An Operating System for the Mission-Critical Integration and Operation of Large, Complex, Distributed Real-Time Systems - An Overview.* OSMCC, 1989.

[JP86] Joseph, M., Pandya, P., *Finding response time in a real-time system.* The Computer Journal, 29(5) : 390-395, 1986.

[Kou04] Koubâa, A., *Gestion de la Qualité de Service temporelle selon la contrainte (m,k)-firm dans les réseaux à commutation de paquets.* Thèse Institut National Polytechnique de Lorraine, 2004.

[KS92] Koren, G., Shasha, D., *Dover : An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems.* In Proceedings 13th Real-Time Systems Symposium, Phoenix, Arizona, USA, 1992.

[KS95] Koren, G., Shasha, D., *Skip-Over algorithms and complexity for overloaded systems that allow skips.* In Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy, 1995.

[Law83] Lawler, E.-L., *Recent Results in the Theory of Machine Scheduling.* Mathematical Programming : the Stae of the Art, A. Bachen et al. (eds.), Springer-Verlag, New York, 1983.

[Loc86] Locke, C.-D., *Best-effort Decision Making for Real-Time Scheduling.* PhD Thesis, Computer Science Department, Carnegie-Mellon University, 1986.

[LC86] Liestman, A.-L., Campbell, R.-H., *A Fault Tolerant Scheduling problem*. In IEEE transactions on software engineering, 12(10) : 1089-1095, 1986.

[LL73] Liu, C.-L., Layland, J.-W., *Scheduling algorithms for multiprogramming in a hard real-time environnement*. Journal of the Association for Computer Machinery, 20(1) : 46-61, 1973.

[LLB+94] Liu, J.-W.-S., Lin, K.-J., Bettati, R.,Hull, D., Yu, A., *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.

[LLN87] Liu, J.-W.-S., Lin, J.-K., Natarajan, S., *Scheduling algorithms for multiprogramming in a hard real-time environment*. In Proceedings of the 8th Real-Time System Symposium, San Francisco, USA, pp. 252-260, 1987.

[LLR76] Lageweg, B.-J., Lenstra, J.-K., Rinnoy Kan, A.-H.-G., *Minimizing maximum lateness on one machine : computational experience and some applciation*. Statistica Neerlandica, 30(1), 1976.

[LM80] Leung, J., Merril, M., *A Note on Preemprive Scheduling of Periodic Real-Time Tasks*. Information Processing Letters, 11(3) : 115-118, 1980.

[LNL87] Lin, J.-K., Natarajan, S., Liu, J.-W.-S., *Condor : A distributed system making use of imprecise results*. In Proceedings of COMPSAC'87, Tokyo, Japan, 1987.

[LR92] Lehozcky, J.-P., Ramos-Thuel, S., *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.

[LSD89] Lehozcky, J.-P., Sha, L., Ding, Y., *The rate-monotonic scheduling algorithm : exact characterization and average case behaviour*. In Proceedings of the IEEE Real-Time Systems Symposium, pp. 166-171, 1989.

[LSS87] Lehozcky, J.-P., Sha, L., Strosnider, K.-K., *Enhanced aperiodic responsivess in hard real-time environments*. In Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 261-270, 1987.

[LT01] Lam, T.-W., To, K.-K., *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.

[LW82] Leung, J.-Y.-T., Whitehead, J., *On the complexity of fixed-priority scheduling of periodic, real-time tasks*. Performance evaluation, 2 :237-250, 1982.

[LW90] Leung, J.-Y.-T., Wong, C.S., *Minimizing the number of late tasks with error constraints*. In Proceedings of the 11th Real-Time Systems Symposium (RTSS'90), Orlando, USA, 1990.

[Mar94] Martineau, P., *Ordonnancement en-ligne dans les systèmes informatiques temps-réel*. PhD Dissertation, University of Nantes, 1994.

[MBD+00] Mantegazza, P., Bianchi, E., Dozio, L., Angelo, M., Beal, D., *DIAPM RTAI Programming Guide 1.0*. Lineo Inc., 2000.

[Mok83] Mok, A.-K., *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD Dissertation, MIT, 1983.

[MD78] Mok, A.-K., Dertouzos, M.-L., *Multiprocessor scheduling in a hard real-time environmnent*. In Proceedings of the 7th Texas Conference on Computer Systems, 1978.

[MF02] Montez, C., Fraga, J. *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.

[MMM00] Mejia-Alvarez, P., Melhem, R., Mosse, D., *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.

[MPR99] Mosse, D., Pollack M., Ronen, Y., *Value Density Algorithms to Handle Transient Overloads in Scheduling*. In Proceedings of the 11th Euromicro Conference on Real-Time Systems, York, England, 1999.

[NT87] Nassehi, M.-M., Tobagi, F.-A., *Transmission scheduling policies and their implementation in integrated-services high-speed local area networks*. In the 5th Annual European Fibre Optic Communications and Local Area Networks Exposition, pp 185-192, Basel, Switzerland, 1987.

[PSK+03] Poggi, E., Song, Y.-Q., Koubaa, A., Wang, Z., *Matrix-DBP for (m,k)-firm guarantee*. In Real-Tme and Embedded Systems, Paris, 2003.

[QH00] Quan, G., Hu, X., *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.

[Ram99] Ramanathan, P., *Overload management in real-time control applications using (m,k)-firm guarantee*. IEEE Transactions on Parallel and Distributed Systems, 10(6), 1999.

[RMP03] Ronen, Y., Mosse, D., Pollack, M.-E., *Using Value-Density Algorithms to Handle Transient Overloads in Deliberation Scheduling*. In IEEE Expert, Special Issue on Real-Time Intelligent Systems, 2003.

[RPT04] Ramya, B.-B., Pavithra, V., Thangaraju, B., *Process Tracing with the Linux Trace Tollkit*. Sys Admin magazine, 2004.

[RRC05] Ridouard, F., Richard, P., Cottet, F., *Ordonnancement de tâches indépendantes avec suspension*. In Proceedings of the13th International Conference on Real-Time Systems, 2005.

[SBS95] Spuri, M., Buttazzo, G.-C., Sensini, F., *Robust Aperiodic Scheduling under Dynamic Priority Systems*. In Proceedings IEEE Real-Time Systems Symposium, Pisa, Italy, 1995.

[SCE90] Silly, M., Chetto, H., Elyounsi, N., *An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems*. IEEE Symposium on Parallel and Distributed Processing, 578-585, 1990.

[Ser72] Serlin, O., *Scheduling of time critical process*. Proceedings of the Joint Computer conference, 40 : 925-932, 1972.

[Sil86] Silly, M., *La tolérance aux fautes dans un système temps réel à contraintes strictes*. INRIA, rapport de recherche N°512, 1986.

[Sil93] Silly-Chetto, M., *Sur la problématique de l'ordonnancement dans les systèmes informatiques temps-réel*. Rapport d'HDR, Université de Nantes, 1993.

[Sil96] Silly-Chetto, M., *On the stability of scheduling algorithms for real-time control*. IMACS/IEEE-SMC Computational Engineering in Systems Applications Multiconference, Lille, France, 1996.

[Sil99] Silly, M., *The EDL server for scheduling periodic and soft aperiodic tasks with resource constraints*. The Journal of Real-Time Systems, 17 : 1-25, 1999.

[SG06] Silly-Chetto, M., Garcia-Fernandez, T., *Cléopatre : Open-sourec Operating System Facilities for Real-Time Embedded Applications*. The Journal of Computing and Information Technology, 2006.

[SK03] Song, Y.-Q., Koubaa, A., *Gestion dynamique de la QdS temps-réel selon (m,k)-firm*. Ecole Temps Réel (ETR'03), Toulouse, 2003.

[SL95] Shih, W.-K., Liu, W.-S., *Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error*, In Proceedings of the IEEE Transactions on Computers, 44(3), 1995.

[SLS95] Strosnider, J.-K., Lehoczky, J.-P., Sha, L., *The Deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments*. IEEE Transactions on Computer, 44(1), 1995.

[Sor74] Sorenson, P.-G., *A methodology for real-time system developement*. PhD Thesis, University of Toronto, Canada, 1974.

[SP94] Spuri, M., Buttazzo, G.-C., *Efficient aperiodic service under earliest deadline scheduling*. In Proceedings of the IEEE Real-Time Systems Symposium, 1994.

[SP96] Spuri, M., Buttazzo, G.-C., *Scheduling aperiodic tasks in dynamic priority systems*. The Journal of Real-Time Systems, 10(2), 1996.

[SRL90] Sha, L., Rajkumar, R., Lehoczky, J.-P., *Priority inheritance protocols : An approach to real-time synchronisation*. IEEE Transactions on Computers, 578-585, 1990.

[SSL89] Sprunt, B., Sha, L., Lehoczky, J.-P., *Aperiodic task scheduling for hard real-time systems*. The Journal of Real-Time Systems, 1 : 27-60, 1989.

[SSR+98] Stankovic, J.-A., Spuri, M., Ramamritham, K., Buttazzo, G.-C., *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 273p., 1998.

[Sta85] Stankovic, J-A., *Stability and distributed scheduling algorithms*. IEEE Transactions on Software Engineering, SE-11(10) :1141–1152, 1985.

[Sta88] Stankovic, J-A., *A Serious Problem for Next-Generation Systems*. IEEE Computer, 21(10) :10–19, 1988.

[Str94] Streich, H., *TaskPair-scheduling : An approach for dynamic real-time systems.* In Proceedings of the 2nd workshop on Parallel and Distributed Real-Time Systems, Cancun, Mexico, 1994.

[TDS+95] Tia, T.-S., Deng, Z., Shankar, M., Storch, M., Sun, J., Wu, L.-C., Liu, J.-W.-S., *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.

[TLS95] Tia, T-S., Liu, J.-W.-S., Shankar, M., *Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems.* The Journal of Real-Time Systems, 1995.

[TLS+94] Tia, T., Liu, J., Sun, J., Ha, R., *A Linear-Time Optimal Acceptance Test for Scheduling of Hard Real-Time Tasks.* Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1994.

[WGS01] 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.

[WS99] West, R., Schawn, K., *Dynamic window-constrained scheduling for multimedia applications.* In Proceedings of the 6th IEEE International Conference on Multimedia Computing and Systems (ICMCS'99), 1999.

[Yag01] Yaghmour, K., *Adaptive Domain Environment for Operating Systems.* Opersys Inc., 2001.

[Yod04] Yodaiken, V., *The RTLinux Approach to Real-Time.* FSMLabs Inc., 2004.

Bibliographie personnelle

Revues Internationales

[1] **A. Marchand** and M. Silly-Chetto, *Dynamic Real-Time Scheduling of Firm Periodic Tasks with Hard and Soft Aperiodic Tasks*, The Journal of Real-Time Systems, 32 (1-2) : 21-47, February 2006.

[2] M. Silly-Chetto and **A. Marchand**, *A Dynamic Scheduler for Real-Time Periodic Tasks with Quality of Service Requirements*, WSEAS Transactions on Computers (ISSN 1109-2750), 12 (5), December 2006.

[3] M. Silly-Chetto, T. Garcia-Fernandez and **A. Marchand**, *CLEOPATRE : Open-source Operating System Facilities for Real-Time Embedded Applications*, The Journal of Computing and Information Technology, to appear.

Conférences Internationales

[4] **A. Marchand** and M. Silly-Chetto, *Stability and Robustness Issues in Scheduling Periodic Tasks with Firm Real-Time Requirements*, In Proceedings 18th Euromicro Conference on Real-Time Systems Work-In-Progress Session, Dresden (Deutschland), July 2006.

[5] **A. Marchand** and M. Silly-Chetto, *RLP : Enhanced QoS Support for Real-Time Applications*, In Proceedings 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), Hong-Kong, August 2005.

[6] **A. Marchand** and M. Silly-Chetto, *QoS Scheduling Components based on Firm Real-Time Requirements*, In Proceedings ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05), Le Caire (Egypt), January 2005.

[7] **A. Marchand** and M. Silly-Chetto, *Dynamic Scheduling of Soft Aperiodic Tasks and Periodic Tasks with Skips*, In Proceedings 25th IEEE International Real-Time Systems Symposium Work-In-Progress Session (WIP RTSS'04), Lisbon (Portugal), December 2004.

[8] **A. Marchand** and M. Silly-Chetto, *QoS and Aperiodic Tasks Scheduling for Real-Time Linux Applications*, In Proceedings 6th Real-Time Linux Workshop (RTLW'04), Singapour, November 2004.

[9] T. Garcia, **A. Marchand** and M. Silly-Chetto, *CLEOPATRE : A R&D project for providing new real-time functionalities to Linux/RTAI*, In Proceedings 5th Real-Time Linux Workshop (RTLW'03), Valencia (Spain), November 2003.

[10] M. Silly-Chetto, **A. Marchand**, T. Garcia and C. Plot, *Scheduling and Fault-Tolerance with Free Open-Source Components for Real-Time Applications*, In Proceedings 11th International Conference on High Performance Scientific Computing (HiPC'03), Hanoï (Vietnam), March 2003.

[11] **A. Marchand**, C. Plot and M. Silly-Chetto, *Real-time mobile robot navigation with LINUX/RTAI*, In Proceedings IEEE International Conference on Control and Automation (ICCA'02), Xiamen (China), June 2002.

Conférences nationales

[12] T. Garcia, **A. Marchand** and M. Silly-Chetto, *Développement et intégration sous Linux/RTAI de composants logiciels temps-réel open-source génériques*, In Proceedings International Conference on Real-Time Systems, Paris, March 2004.

Journées scientifiques

[13] **A. Marchand** and M. Silly-Chetto, *Ordonnancement temps-réel dynamique avec contraintes de qualité de service*, 5ème Journée des Doctorants (JDOC), Nantes, April 2005.

Glossaire

L'objet de ce glossaire qui s'appuie sur celui présenté dans [Sil93], est de récapituler les notations et définitions introduites tout au long de ce document.

Notations

Stratégies d'ordonnancement

ED : Earliest Deadline
EDS : Earliest Deadline as Soon as possible
EDL : Earliest Deadline as Late as possible
RTO : Red Tasks Only
BWP : Blue When Possible
RLP : Red Tasks as Late as possible
RLP/T : Red Tasks as Late as possible with blue acceptance Test
LF : Last Failure
MS : Minimum Success

Serveurs de tâches apériodiques

BG : Background server
TBS : Total Bandwidth Server
TB* : Optimal Total Bandwidth Server
EDL : Earliest Deadline as Late as possible

Configurations de tâches

▶ $\mathcal{T}$: configuration de tâches périodiques Skip-Over

n : nombre de tâches dans $\mathcal{T}$
T_i : tâche périodique n°i
r_i : date de réveil de T_i
C_i : durée d'exécution au pire-cas de T_i
P_i : période d'activation de T_i
D_i : délai critique de T_i

d_i : échéance relative de T_i
s_i : paramètre de pertes de T_i

▶ $\mathcal{R}$: configuration de tâches apériodiques non critiques

m : nombre de tâches dans $\mathcal{R}$
R_i : tâche apériodique n°i
r_i : date de réveil de R_i
C_i : durée d'exécution au pire-cas de R_i

▶ $\mathcal{S}$: configuration de tâches apériodiques critiques

p : nombre de tâches dans $\mathcal{S}$
S_i : tâche apériodique n°i
r_i : date de réveil de S_i
C_i : durée d'exécution au pire-cas de S_i

Calcul des temps creux d'une séquence d'ordonnancement

f_Y^X : fonction de disponibilité d'un processeur supportant une configuration de tâches Y ordonnancée par un algorithme X
$\Omega_Y^X(t_1, t_2)$: temps d'oisiveté du processeur entre t_1 et t_2 si Y est ordonnancée par X
P' : PPCM$\{s_i P_i; T_i \in \mathcal{T}\}$, et N' : nombre de requêtes dans $[0, P'[$
P : PPCM$\{P_i; T_i \in \mathcal{T}\}$, et N : nombre de requêtes dans $[0, P[$
τ : instant courant

▶ Calcul des temps creux statiques selon EDS

$\mathcal{E}$: vecteur statique des dates de réveil dans $[0, P[$
e_i : ième date de réveil pour $\mathcal{T}$ dans $[0, P[$
$\mathcal{D}$: vecteur statique des temps creux dans $[0, P[$
Δ_i : longueur du temps creux qui précède e_i dans la séquence produite par EDS

▶ Calcul des temps creux dynamiques selon EDS

$\mathcal{E}(\tau)$: vecteur dynamique des dates de réveil dans $[\tau, kP[$
$e_i(\tau)$: ième date de réveil pour $\mathcal{T}$ dans $[\tau, kP[$
$\mathcal{D}(\tau)$: vecteur dynamique des temps creux dans $[\tau, kP[$
$\Delta_i(\tau)$: longueur du temps creux qui précède $e_i(\tau)$ dans la séquence produite par EDS

▶ Calcul des temps creux statiques selon EDL

$\mathcal{K}$: vecteur statique des échéances dans $[0, P[$
k_i : ième échéance pour $\mathcal{T}$ dans $[0, P[$
$\mathcal{D}^*$: vecteur statique des temps creux dans $[0, P[$
Δ_i^* : longueur du temps creux qui suit k_i dans la séquence produite par EDL

► **Calcul des temps creux dynamiques selon EDL**

$\mathcal{K}(\tau)$: vecteur dynamiques des échéances dans $[\tau, kP[$
$k_i(\tau)$: ième échéance pour $\mathcal{T}$ dans $[\tau, kP[$
$\mathcal{D}^*(\tau)$: vecteur dynamique des temps creux dans $[\tau, kP[$
$\Delta_i^*(\tau)$: longueur du temps creux qui suit $k_i(\tau)$ dans la séquence produite par EDL

Définitions

► **Acceptabilité (test d')** : ensemble des relations à satisfaire pour assurer à une configuration de tâches d'être faiblement ordonnancée par un algorithme d'ordonnancement donné.
► **Acceptation (d'une tâche)** : le fait qu'une tâche ou un groupe de tâches soient admises à s'exécuter sur une machine en garantissant le respect de leurs contraintes.

► **ACET (d'une tâche)** : durée d'exécution effective (Average Case Execution Time) d'une tâche, inférieure à sa durée d'exécution au pire-cas (voir WCET).

► **Algorithme (d'ordonnancement)** : algorithme de planification temporelle de l'activité d'un ou plusieurs processeurs sur une configuration de tâches.

► **Application temps réel** : application dont l'objectif est de contrôler ou commander des activités caractérisées par des impératifs temporels.

► **Configuration (de tâches)** : modélisation de l'ensemble des tâches qui constitue un logiciel d'application.

► **Criticité** : caractère d'une tâche temps réel dont la violation des contraintes temporelles a de graves conséquences.

► **Délai critique (d'une tâche)** : intervalle de temps séparant la date de réveil de l'échéance d'une tâche.

► **Date critique (d'une tâche)** : date de fin d'exécution au plus tard d'une tâche.

► **Date d'arrivée (d'une tâche)** : date à laquelle une tâche apériodique fait connaître son désir de s'exécuter.

► **Date de réveil (d'une tâche)** : date de début d'exécution au plus tôt d'une tâche.

► **Dispatcheur** : module du système d'exploitation qui est responsable de l'assignation immédiate de l'unité centrale aux tâches prêtes à s'exécuter.

▶ **Durée d'exécution (d'une tâche)** : intervalle de temps maximal séparant le début de la fin d'exécution (sans interruption) d'une tâche sur un processeur donné.

▶ **Echéance (d'une tâche)** : voir Date critique.

▶ **Fenêtre** : se dit d'un intervalle de temps de référence. Pour une configuration de tâches périodiques, correspond à $[kP, (k+1)P]$, $k \geq 0$.

▶ **Faisabilité** : caractère d'un problème d'ordonnancement pour lequel il existe au moins une solution.

▶ **Fonction de disponibilité** : fonction du temps à valeurs binaires représentatives de l'activité du processeur.

▶ **Laxité (d'un processeur)** : intervalle de temps maximal pendant lequel un processeur peut rester inactif sans induire un manquement au respect des contraintes temporelles.

▶ **Laxité (d'une tâche à date critique)** : intervalle de temps séparant sa date de fin d'exécution de son échéance, calculé en tenant compte de l'exécution des tâches plus prioritaires.

▶ **Modèle d'ordonnancement** : association combinée d'un serveur de tâches apériodiques et d'un ordonnanceur de tâches périodiques, pour la gestion d'un ensemble hybride de tâches.

▶ **Optimalité (d'un algorithme d'ordonnancement)** : se dit d'un algorithme capable d'ordonnancer correctement une configuration de tâches chaque fois que celle-ci est ordonnançable.

▶ **Ordonnancement** : détermination de l'ordre dans lequel les ressources et plus spécialement les processeurs sont assignés aux tâches.

▶ **Ordonnanceur** : composant du système d'exploitation responsable de l'ordonnancement.

▶ **Ordonnançabilité** : caractère d'une configuration de tâches pour laquelle le respect des contraintes temporelles peut être assuré par au moins un algorithme d'ordonnancement.

▶ **Overhead** : surplus de temps processeur requis par l'exécutif pour gérer l'ensemble des tâches.

▶ **Priorité** : paramètre associé à toute tâche pour établir un ordre relatif d'exécution. La priorité peut être fixe (calculée initialement et indépendante du temps) ou dynamique (évoluant au cours du temps).

▶ **Qualité de Service (QoS)** : ensemble d'exigences de qualité à respecter. En temps réel, désigne la capacité du système à respecter des contraintes temporelles plus ou moins fortes.

▶ **Requête (d'une tâche périodique)** : désigne une instances d'exécution d'une tâche périodique.

▶ **Robustesse** : capacité d'un système à maintenir un niveau de performance considéré comme acceptable en toutes circonstances.

▶ **Séquence** : description du travail effectué par un (ou plusieurs) processeur(s) sur une configuration de tâches.

▶ **Stabilité** : critère de performance exprimé en termes d'équilibre d'un paramètre particulier (par exemple, le taux de respect individuels des tâches).

▶ **Surcharge (d'un système)** : cas où la somme des exécutions des tâches au sein du système dépasse la capacité de traitement du processeur.

▶ **Système temps réel** : système informatique utilisé pour la mise en oeuvre d'une application temps réel.

▶ **Tâche** : entité logicielle réalisant une fonction particulière.

▶ **Temps creux (d'un processeur)** : intervalle de temps pendant lequel un processeur est inactif.

▶ **Temps de réponse (d'une tâche)** : intervalle de temps séparant le réveil d'une tâche de sa fin d'exécution.

▶ **WCET (d'une tâche)** : durée d'exécution au pire-cas (Worst Case Execution Time) d'une tâche.

Résumé en français

Les travaux de thèse consistent à étudier et à proposer des stratégies d'ordonnancement novatrices en matière de gestion de surcharge du processeur, puis ensuite à les tester en simulation pour permettre leur intégration sous Linux temps réel. Cette intégration sous forme de composants doit avoir pour objectif l'aide à la conception d'un système temps réel sûr de fonctionnement offrant la "meilleure" qualité de service (QoS). Dans un premier temps, les travaux se concentrent sur l'amélioration des stratégies d'ordonnancement Skip-Over impliquant des tâches périodiques avec contraintes de QoS. Les travaux portent ensuite sur les modèles d'ordonnancement hybrides de tâches constitués de tâches périodiques avec contraintes de QoS et de tâches apériodiques critiques ou non critiques. L'étude a été menée pour les serveurs de tâches apériodiques BG, TBS, TB* et EDL. Enfin, des variantes des stratégies d'ordonnancement précédemment décrites ont été proposées sur la base de l'amélioration de la robustesse et de la stabilité des algorithmes. L'ensemble des algorithmes validés par la simulation, a été intégré et testé sous une variante de Linux pour le temps réel, Linux/RTAI (Real-Time Application Interface).

Titre et Résumé en anglais

REAL-TIME SCHEDULING UNDER QUALITY OF SERVICE CONSTRAINTS

The thesis work reported in this document aims at providing innovative scheduling solutions for processor overload cases. The work consists in testing scheduling algorithms via simulation so as to integrate them into a version of Linux for real-time. This integration is performed with components whose objective is helping users to design safe real-time systems offering the "best" quality of service (QoS). First, we propose two novel scheduling strategies of real-time periodic tasks defined under QoS constraints using the Skip-Over model. After, we study the problem of scheduling hybrid task sets consisting of aperiodic tasks and periodic tasks defined under QoS constraints. A performance study of the different aperiodic task servers considered (BG, TBS, TB*, EDL) is reported. Then, we are interested in variants improving the stability and robustness of the proposed algorithms. Finally, we present the integration of the scheduling strategies studied and proposed within the framework of the thesis, under a version of Linux for real-time, Linux/RTAI (Real-Time Application Interface).

Mots-clés :

Temps réel, ordonnancement, Qualité de Service, Earliest Deadline, tâches périodiques, Linux temps réel, gestion de surcharge

Discipline : Automatique et Informatique Appliquée

N°
(Ne rien inscrire, attribué par la bibliothèque)