Bibliographie d'introduction à l'ordonnancement dans les systèmes informatiques temps-réel

David Decotigny

Novembre 2002

Introduction

Un point commun entre une usine, un avion, une centrale nucléaire, une machine à laver, une auto, et un appareil multimédia de salon, est qu'ils possèdent tous, de nos jours, un ou plusieurs processeur(s) ou microcontrôleur(s) pour les gérer. Un autre point commun entre ces *systèmes*, est que le *logiciel* qui y est déroulé, a conscience de l'écoulement du temps, et prend des décisions en fonction de celui-ci. On parle alors de système *temps-réel*, sachant que ce terme revêt un sens extrêmement large (chapitre 1).

Dans ce travail, la notion de temps-réel dans les systèmes que nous étudions possède deux caractéristiques (chapitre 2) : *i)* elle correspond à une série de *contraintes temporelles* (absolues, relatives, moyennes) que le système doit absolument vérifier, sous peine de conséquence catastrophique sinon (perte de vies humaines, destruction de matériel, impact financier important, ...), *ii)* le temps est perçu comme une *grandeur physique mesurable*, et le système a *conscience* qu'elle affecte tous les traitements logiciels effectués.

Dans ce contexte, la problématique qui nous intéresse est celle qui consiste à *vérifier* que le système respectera toutes les *contraintes temporelles spécifiées* avant toute exécution réelle du système ; on parle d'approche par *analyse* (à la différence d'approches par *exécution* réelle ou simulée, qui ne peuvent que démontrer que le système est défaillant, et jamais qu'il est correct). *Prouver* de telles *propriétés* nécessite deux choses : *i)* être capable de caractériser le comportement temporel de chaque entité du système prise indépendamment, *ii)* organiser l'ensemble des traitements associés à chacune de ces entités, de sorte que les contraintes temporelles imposées soient toutes respectées. Dans cette étude, nous nous intéressons au deuxième point, qui porte le nom d'*ordonnancement temps-réel* (chapitre 3). Le cœur de la problématique associée provient du fait qu'il y a *compétition* entre les entités pour accéder aux *ressources* (dont le ou les processeur(s) fait/font partie) disponibles en nombre et en capacité restreints, et que par conséquent un arbitrage des accès à ces ressources est nécessaire. Dans ce chapitre, nous présentons quelques résultats scientifiques (propriétés, algorithmes, approches, ...) liés à cette problématique.

Ce document est extrait d'une version préliminaire de mon rapport de thèse, qui est désormais allégé d'une grosse partie de ce qui constitue la présente étude.

Chapitre 1

Définitions et problématique du temps-réel

1.1 Différentes définitions du *temps-réel*

La notion de "*temps-réel*" suggère bien évidemment celle de *temps*. L'acception du concept de temps est très large, mais nous nous concentrons dans ce travail sur la définition du temps en tant que donnée physique mesurable. Intuitivement, les systèmes informatiques "temps-réel" seraient donc ceux pour lesquels le comportement temporel, non seulement en terme de séquence d'opérations (propriété logique), mais également en terme de quantification de l'écoulement de ces opérations (propriété physique), a une importance.

Temps-réel et temps logique

Selon la définition officielle du temps-réel par la délégation générale à la langue française et aux langues de France[1] [4], le temps-réel est un "*mode de traitement qui permet l'admission des données à un instant quelconque et l'obtention immédiate des résultats*". Cette définition repose principalement sur l'adjectif "immédiat". Elle introduit la notion de *temps logique*, et s'applique aux systèmes où on s'intéresse aux successions logiques d'événements et de décisions (domaine des systèmes *réactifs* qui reposent sur l'hypothèse de synchronisme fort) [Hal98, Bon92], mais ne convient pas aux problèmes où on s'intéresse aux propriétés quantitatives de l'écoulement des traitements qui ont un coût temporel non nul.

Temps-réel et temps physique

D'autres définitions existent qui relativisent l'adjectif "immédiat" de la précédente, et confèrent une existence *physique* au temps, dont celle-ci (canadienne) : "*mode de traitement des données selon lequel le traitement s'effectue avec un léger décalage [temporel]*" [3]. Mais sans précision sur l'adjectif "léger", l'idée qu'on se fait de tels systèmes

[1] Ministère de la Culture, arrêté du 22/12/1981.

"temps-réel" est qu'ils sont "rapides", c'est à dire que le "décalage" est le plus petit possible (on parle aussi de systèmes *interactifs*). Cette définition n'est pas un grand progrès par rapport à la problématique de la conception de systèmes informatiques classiques, puisqu'un grand pan de l'effort en développement informatique consiste à faire les bons choix algorithmiques afin d'obtenir des temps moyens d'exécution convenables, voire minimaux.

Notion de contrainte de temps

D'autres définitions précisent la précédente en donnant une valeur numérique à l'adjectif "léger" (typiquement : 0,5 secondes dans la définition de "temps-réel dur" de [3]). Ces définitions introduisent la notion de *contrainte* temporelle. Cependant, ce relatif progrès en précision se révèle n'être d'aucune utilité : pour certaines applications (contrôle d'un avion par exemple), un "décalage" de 0,5 secondes peut être synonyme de catastrophe, et pour d'autres applications il peut se révéler totalement inadapté ou impertinent (contrôle de le température d'une pièce par exemple). Il en découle que les contraintes temporelles sont dépendantes de l'application considérée.

Définition du temps-réel *adoptée et problématique*

Dans le présent travail, nous reposons sur la définition de temps-réel suivante, largement adoptée dans le domaine [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*". Cette définition implique que les contraintes temporelles à respecter (par exemples les échéances des traitements) sont relatives à un temps physique mesurable, et font partie de la spécification du système à implanter. En outre, elle signifie que la seule rapidité moyenne d'exécution du logiciel ne conditionne pas la validité du système. Elle suggère que dans tous les cas, même les *pires*, toutes les contraintes temporelles doivent être respectées, sans quoi le système est *défaillant*.

Dans ce contexte, la problématique qui nous intéresse est celle de la conception et de la validation de systèmes informatiques qui sont soumis à des contraintes temporelles en plus des contraintes de correction fonctionnelles usuelles. Parmi les contraintes temporelles courantes, citons par exemple les échéances temporelles strictes des traitements, ou la gigue de démarrage des traitement ; nous reviendrons sur ces notions en 2.1.1.2.

1.2 Classification sommaire des systèmes informatiques temps-réel

L'utilisation de l'outil informatique apparaît de plus en plus fréquemment dans les domaines où l'interaction avec l'environnement constitue une raison d'être essentielle du système. L'intérêt de recourir à l'informatique dans ce genre de système est dû au fait que les fonctionnalités offertes le sont à coût moindre (à fonctionnalités équivalentes), en terme de temps de développement, de temps et de coûts de fabrication, d'encombrement

ou de poids, par rapport aux solutions électroniques ou mécaniques. En outre, l'ajout de nouvelles fonctionnalités, la constitution de mises à jour, ou la personnalisation du produit se révèlent souvent être des tâches moins lourdes.

Temps-réel critique

On trouve aujourd'hui des gros systèmes informatiques temps-réel dans les domaines de l'aéronautique, de l'aérospatiale, des transports ferroviaires, du contrôle de procédés industriels, de supervision de centrales nucléaires, voire de gestion de salles de marchés ou de télémédecine. On trouve aussi de beaucoup plus petits systèmes *embarqués* dans l'automobile (système de contrôle des freins, du moteur) par exemple. Pour ces exemples, une des caractéristiques est que le système informatique se voit confier une grande responsabilité en terme de vies humaines, de conséquences sur l'environnement, voire de conséquences économiques. On parle alors de systèmes *critiques*, qui sont alors soumis à des contraintes de fiabilité. Cette caractéristique amène les problématiques orthogonales de sûreté de fonctionnement et de tolérance aux fautes [CP99b, CP99a] que nous ne détaillerons pas dans ce travail.

Temps-réel strict et temps-réel souple

La majorité des systèmes temps-réel critiques est exclusivement constituée de traitements qui ont des contraintes temporelles *strictes* : on parle de systèmes *temps-réel strict*. C'est à dire qu'en condition nominale de fonctionnement du système, **tous** les traitements du système doivent impérativement respecter toutes leurs contraintes temporelles ; on parle alors de traitements *temps-réel strict* ou *dur* (*hard* en anglais). Ceci suppose deux choses : *i)* qu'on est capable de définir les conditions de fonctionnement nominales en terme d'hypothèses sur l'environnement avec lequel le système interagit ; *ii)* qu'on est capable de garantir avant exécution que tous les scénarios d'exécution possibles dans ces conditions respecteront leurs contraintes temporelles. Ceci suppose à son tour qu'on puisse extraire ou disposer de suffisamment d'informations sur le système pour déterminer tous les scénarios possibles.

Une autre classe de systèmes est moins exigeante quant au respect absolu de *toutes* les contraintes temporelles. Les systèmes de cette classe, dits *temps-réel souple* (*soft* en anglais), peuvent souffrir un taux "acceptable" de *fautes temporelles* (non respect transitoire des contraintes) de la part d'une partie des traitements (eux-mêmes dits "*temps-réel souple*"), et sans que cela ait des conséquences catastrophiques. Cette classe comprend entre autres les systèmes où la qualité est appréciée par nos sens : on parle dans ce cas de systèmes et d'applications multimédia (téléphonie, vidéo, rendu visuel interactif par exemple). La mesure du respect des contraintes temporelles prend la forme d'une donnée probabiliste : la *qualité de service* relative à un service particulier (nombre d'images ou nombre d'échantillons sonores rendus par secondes, par exemple), ou relative au comportement du système dans son ensemble (nombre de traitements qui ont pu être rendus dans les temps, tous services confondus, par exemple), ou les deux combinés. Une problématique de cette classe de systèmes est d'évaluer la qualité de service, avant ou pendant le fonctionnement, que le système offre ou va pouvoir offrir

en cours de fonctionnement, en fonction des caractéristiques de l'environnement et du système. On distingue une sous-famille dans celle des traitements temps-réel souple, suivant que le non-respect d'une contrainte temporelle continue d'apporter quelque chose au système, ou non : si le traitement dépasse une contrainte temporelle, et si le continuer dans ces conditions jusqu'à son terme n'apporte rien au système, on parle de traitement *temps-réel ferme* (*firm* en anglais).

Il est également possible d'avoir des systèmes temps-réel strict non critiques, ou des systèmes temps-réel critiques qui contiennent des traitements temps-réel souple. Nous nous intéressons dans tout ce qui suit aux systèmes qui possèdent une composante temps-réel strict.

Chapitre 2

Modèle considéré et contraintes de conception pour le temps-réel

Il y a de nombreux formalismes de modélisation de systèmes informatiques qui font intervenir le temps. Citons en particulier les méthodes formelles (logiques temporelles, algèbres de processus), les méthodes structurées (systèmes réactifs synchrones et asynchrones, réseaux de Pétri temporels et temporisés par exemples).

Pour notre part, nous nous intéressons au formalisme utilisé dans la communauté de l'ordonnancement temps-réel. Dans ce formalisme, le temps est une grandeur physique mesurable (espace discret ou dense) qui permet de caractériser les éléments du système, dont les traitements font partie (on parle de *tâches*). Le système est modélisé sous la forme de demandeurs de ressources d'un côté, et de serveurs de ressources de l'autre, le tout étant arbitré par un composant central : le *support d'exécution* (en général un système d'exploitation). La capacité des serveurs peut correspondre à une densité temporelle (*ressources actives* : processeurs, réseau), ou à une capacité scalaire (*ressources passives* : verrous, sémaphores, rendez-vous par exemple). Et les demandeurs en ressources (les *traitements*) ont des contraintes de validité fonctionnelles et temporelles, en même temps que des besoins en ressources quantifiés.

2.1 Modèle de système considéré

Le formalisme adopté possède l'avantage d'être proche des implantations concrètes, ce qui permet un dimensionnement simple du système d'après le modèle, et ce qui permet de valider l'implantation avec les mêmes outils que pour la vérification du modèle.

Plus précisément, nous nous appuyons sur le modèle de système de la figure 2.1.

Le *système* est soumis à des *événements* provenant de l'*environnement* du système, et générés par du matériel spécialisé (réseau, capteurs ou générateur d'impulsions par exemple). Il effectue des *traitements* en tenant compte ou non de ces événements, qui peuvent décider d'*actions* à entreprendre sur l'environnement au moyen de témoins (écrans ou diodes par exemples), d'actionneurs (moteurs ou relais par exemple), ou de

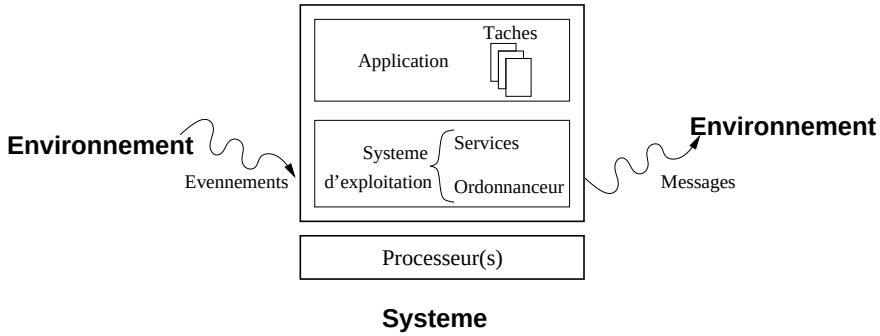


FIG. 2.1: Modèle de système considéré

messages (réseau par exemples).

2.1.0.1 Niveau logiciel

Les traitements forment l'*application* et le *support d'exécution* temps-réel. L'application effectue les traitements utiles du système pour lesquels elle a des besoins en *ressources* logiques ou matérielles (entre autres des ressources processeur), et est soumises à des *contraintes* liées aux spécifications, dont les contraintes temporelles font partie. Elle est structurée en *tâches*, c'est à dire en flots de contrôle (*i.e.* séquences d'opérations du processeur).

Le support d'exécution est chargé de répartir les demandes en ressources de l'application entre les ressources matérielles ou logiques disponibles, tout en veillant au respect de toutes les contraintes. Il peut prendre la forme d'un système d'exploitation contrôlant directement le matériel, ou d'un *intergiciel* (*middleware* en anglais) s'intercalant entre le système d'exploitation et l'application. La particularité d'un support d'exécution temps-réel est qu'il prend les contraintes temporelles ainsi que le comportement temporel de l'application et du système en compte. Tout particulièrement, c'est une composante du support d'exécution qui décide quelle tâche doit s'exécuter sur quel processeur : l'*ordonnanceur*.

Dans les sections 2.1.1 et 2.2 qui suivent, nous revenons plus précisément sur la caractérisation temporelle des tâches et du support d'exécution.

2.1.0.2 Niveau matériel

Le système repose sur un ou plusieurs *processeurs* pour effectuer les traitements. Il est découpé en *nœuds* qui communiquent (par messages ou par mémoire partagée), chaque nœud correspondant à un processeur et aux traitements qui sont exécutés par le processeur au cours de la vie du système.

En temps-réel, l'interaction entre logiciel et matériel est fine, puisque le comportement temporel des traitements exécutés par les processeurs dépend des capacités des processeurs (puissance de calcul). Nous voyons dans la suite que cette donnée est importante pour l'établissement des propriétés temporelles du système.

2.1.1 Caractérisation des tâches

2.1.1.1 Description et définitions

Une tâche est en charge de fournir un des services de l'application. Elle correspond aux exécutions d'une séquence d'opérations donnée sur le processeur. Que les tâches partagent un unique espace d'adressage, ou que le cloisonnement mémoire soit possible (notion de processus Unix par exemple), est indifférent pour la suite.

Dans le modèle canonique du domaine, le service fourni par la tâche peut être rendu plusieurs fois au cours de la vie du système. Ce qui signifie que la séquence d'opérations d'une tâche peut être ré-exécuté plusieurs fois. Pour cette raison, chacune de ces exécutions de la séquence d'opérations est considérée individuellement sous la forme de *travaux* ("*jobs*" en anglais).

Plusieurs travaux de plusieurs tâches (distinctes ou non) sont susceptibles de s'exécuter sur un même processeur, mais, par définition, ce dernier ne peut en exécuter qu'un seul à la fois. Un travail est donc associé à une structure de données qui renferme en particulier son *état* d'exécution. Le système d'exploitation s'occupe entre autre de passer l'état des travaux d'un état à un autre suivant le diagramme de transition des travaux (la figure 2.2 propose un exemple), en même temps qu'il choisit d'exécuter un travail plutôt qu'un autre. Généralement, le diagramme d'état des travaux fait couramment intervenir les états suivants :

ready : travail prêt à démarrer, *i.e.* pas encore démarré, ou dont l'exécution a été interrompue d'autorité (suite à un événement matériel par exemple) pour laisser la place à un autre travail.

running : travail en cours d'exécution sur le processeur ;

blocked : travail en attente de la disponibilité d'une ou plusieurs ressources ;

sleeping : travail en sommeil, en attente d'un événement de réveil ;

stopped : travail terminé.

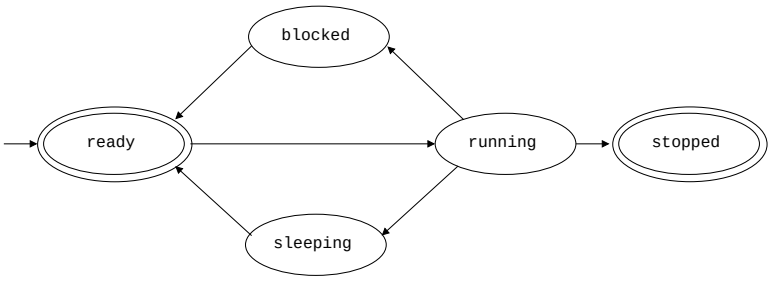


FIG. 2.2: Diagramme d'état des travaux

La vie d'un travail peut également être représentée suivant un *chronogramme* ou *diagramme de Gantt* tel que celui de la figure 2.3, ce qui permet d'introduire la terminologie suivante :

① **: date de création ou d'activation.** Le travail est créé et prêt à être exécuté sur le processeur. *Arrival* ou *Request time* en anglais.

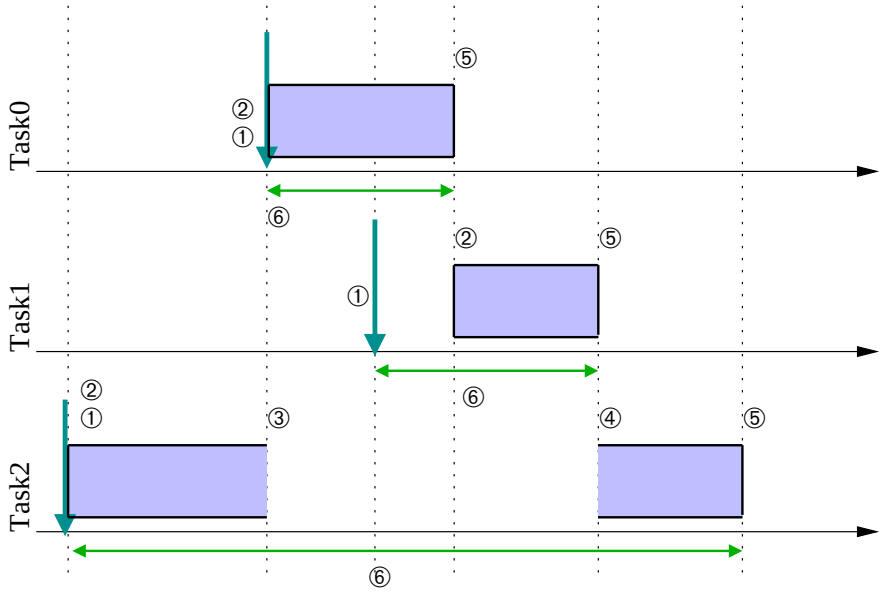


FIG. 2.3: Événements au cours de la vie d'un travail

② **: date de démarrage.** Le travail commence à être exécuté sur le processeur pour la première fois. *Start time* en anglais.

③ **: dates de *préemption*.** Le travail est momentanément interrompu au profit d'un autre, plus *prioritaire* (plusieurs possibles durant la vie du travail).

④ **: dates de *reprise*.** Le processeur reprend l'exécution du travail là où il avait été précédemment préempté (plusieurs possible durant la vie du travail).

⑤ **: date de terminaison.** Le travail termine de s'exécuter sur le processeur. *Finishing* ou *completion time* en anglais.

⑥ **: temps de réponse.** L'écart entre la date de terminaison et la date d'activation d'un travail. *Response time* en anglais.

2.1.1.2 Caractéristiques et contraintes temporelles

Le système d'exploitation doit connaître les caractéristiques temporelles des travaux de chaque tâche, formant une partie du *modèle de tâche*, puisqu'il doit garantir que les contraintes temporelles sont respectées. Dans le modèle de tâche figurent habituellement les caractéristiques temporelles suivantes :

Le temps d'exécution : il s'agit du temps d'exécution d'un travail de la tâche sur le processeur et considéré de manière isolé. En général, il s'agit d'un *temps d'exécution pire-cas* (ou *WCET* pour *Worst Case Execution Time* en anglais), c'est à dire un majorant sur tous les temps d'exécution possibles de tous les travaux de la tâche, chacun pris en isolement du reste du système. Cette donnée constitue un paramètre d'entrée important aux méthodes et outils de vérification

du respect des échéances. Quelques systèmes temps-réel souple considèrent un temps d'exécution moyen, ou minimal.

La loi d'arrivée : Il s'agit de la répartition dans le temps des dates de création des travaux de la tâche. On distingue couramment trois lois :

périodique :

Les travaux de la tâche sont créés de façon rigoureusement périodique, et la période est précisée. Si les périodes T_i des tâches du système sont telles que $\forall i, j,\ T_i > T_j \Rightarrow T_i$ multiple de T_j, alors le système est *harmonique*. Si la date d'activation du premier travail est donnée, la tâche est dite *concrète*. De plus, si toutes les tâches sont périodiques concrètes, et si les dates d'activation du premier travail sont identiques, les tâches sont dites *synchrones*. Si toutes les tâches du système sont synchrones, le motif d'activation des travaux se répète à l'identique par intervalles de durée multiple de l'*hyperpériode* (le plus petit commun multiple des périodes des tâches du système, *i.e.* la plus grande période dans le cas particulier d'un système harmonique). Par extension, on conserve cette définition de l'hyperpériode pour des tâches non concrètes.

sporadique :

Les travaux de la tâche sont créés de sorte qu'une durée minimale sépare deux travaux successifs : le *délai d'inter-arrivée*, qui est précisé. Une tâche périodique est un cas particulier de tâche sporadique.

apériodique :

Un travail de la tâche peut être créé à tout instant. Une tâche sporadique est un cas particulier de tâche apériodique.

D'autres lois d'arrivée moins usuelles sont possibles. La connaissance de cette donnée est un paramètre d'entrée important aux outils de vérifications de la garantie du respect des échéances.

Le modèle de tâche indique également les contraintes temporelles des travaux. Les contraintes temporelles courantes sont (voir la figure 2.4) :

① **: date d'échéance :** il s'agit de la date à laquelle les travaux doivent être terminés, relativement à leur date de création, ou sous forme d'un vecteur de dates absolues. *Relative* ou *absolute deadline* en anglais.

② **: gigue de démarrage :** il s'agit de la mesure de dispersion (écart-type, variance) des délais entre la date de création et la date de démarrage des travaux d'une tâche. *Release jitter* en anglais.

③ **: précédence :** avant de pouvoir démarrer, un travail doit attendre le résultat d'un travail d'une autre tâche. On représente ce type de contrainte sous la forme d'un

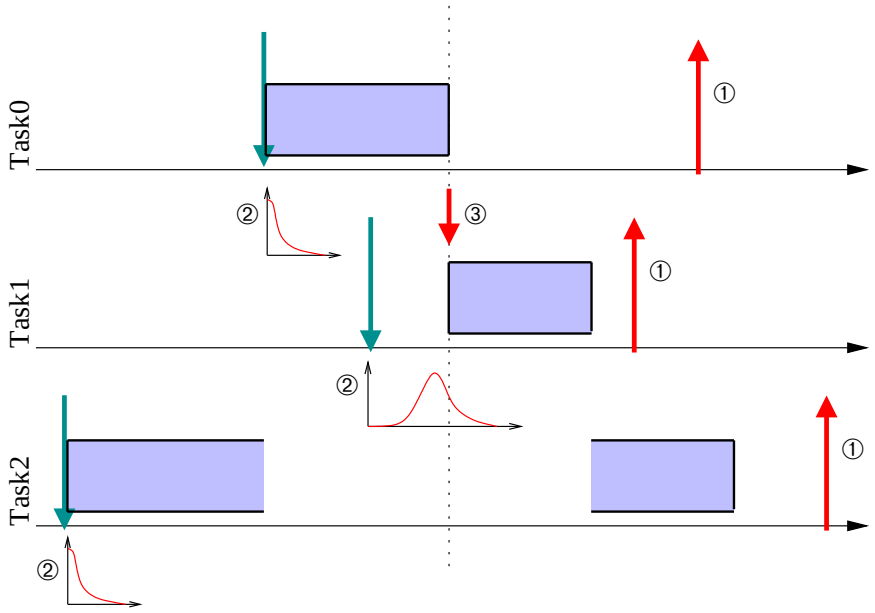


FIG. 2.4: Contraintes temporelles courantes

graphe de précédence, *i.e.* un graphe orienté dont les nœuds sont les tâches et dont les arcs sont les relations de précédence (éventuellement augmentées des messages qui transitent).

Quelques mesures pour caractériser le comportement temporel sont dérivées des contraintes et des caractéristiques temporelles :

L'utilisation (ou *taux d'utilisation*) : c'est la proportion de ressources données occupées par les travaux d'une tâche. Le plus souvent, on s'intéresse à l'utilisation du processeur : si la tâche τ_i est périodique de période T_i et si chaque travail nécessite un temps d'exécution C_i sur le processeur, alors son taux d'utilisation est : $u_i = \frac{C_i}{T_i}$. On définit également l'utilisation processeur pour un ensemble de tâches périodiques : $U = \sum_i u_i = \sum_i \frac{C_i}{T_i}$. On montre aisément que 1 est une borne supérieure d'utilisation : au delà de ce seuil, la ressource est dite *surchargée* puisque potentiellement la demande en ressource excède la capacité disponible.

Le retard : c'est l'écart entre la date de terminaison du travail et la date d'échéance. Négatif quand le travail respecte son échéance. *Lateness* en anglais ("*tardiness*" lorsqu'on ne s'intéresse qu'aux retards positifs).

La laxité : c'est l'écart maximal entre la date d'activation et la date de démarrage du travail, de sorte que l'échéance demeure respectée. *Laxity* ou *Slack time* en anglais.

Les contraintes temporelles doivent être impérativement respectées en temps-réel dur. En temps réel souple, des *dépassements* qui ne remettent pas en cause la structure de l'application (*i.e.* tout sauf les contraintes de précédence) sont tolérées. Dans ce cas,

il est possible de définir une métrique qui mesure la *qualité de service* qui demeure assurée (par exemple le nombre de travaux qui respectent leur échéance relativement au nombre total de travaux créés pour la tâche). Cette qualité de service peut être évaluée globalement pour tout le système, ou relativement à chaque sous-système (ensemble de tâches de l'application), avant fonctionnement (par analyse) et/ou en cours de fonctionnement (exécution effective ou simulation). Dans certains systèmes, la qualité de service constitue une contrainte supplémentaire conditionnant la correction du comportement du système (systèmes dits à *réservation de ressources* en particulier).

Dans la suite, sauf précision en cas d'ambiguïté, on confond les notions de *travail* et de *tâche*, puisqu'un travail est une instance de tâche.

2.1.2 Contraintes de ressources

Les tâches peuvent être vues comme des éléments du système qui consomment la ressource processeur. D'autres ressources existent au sein du système, qui peuvent être *matérielles* (processeur(s), mémoire, réseau, capteurs, actionneurs par exemple), ou *logiques* (sémaphores, files de messages par exemple). Le système d'exploitation a pour rôle de gérer toutes les ressources, c'est-à-dire d'arbitrer entre les demandes que les tâches de l'application font, et les ressources effectivement disponibles dans le système.

Dans le cas de ressources *actives* [SL94], c'est à dire celles qui permettent aux travaux de progresser dans leur exécution, telles que les processeurs dans le domaine du temps-réel, et le réseau dans le domaine des communications par réseau, cet arbitrage consiste en un partage temporel (*i.e. multiplexage*) de l'accès aux ressources. Les décisions de partage sont faites suivant un *ordonnancement* (détaillé en 3 dans le cas de la ressource processeur).

Dans le cas des autres ressources (les ressources *passives*), l'objectif est de restreindre le nombre et les types d'accès (lecture/écriture) afin de maintenir la cohérence : c'est le rôle des mécanismes de *synchronisation* (verrous, sémaphores, conditions, moniteurs, files de messages par exemple), qui par extension permettent également aux tâches d'échanger des informations ou de se synchroniser sans qu'il y ait nécessairement de ressource à protéger (comme avec les *rendez-vous* par exemple).

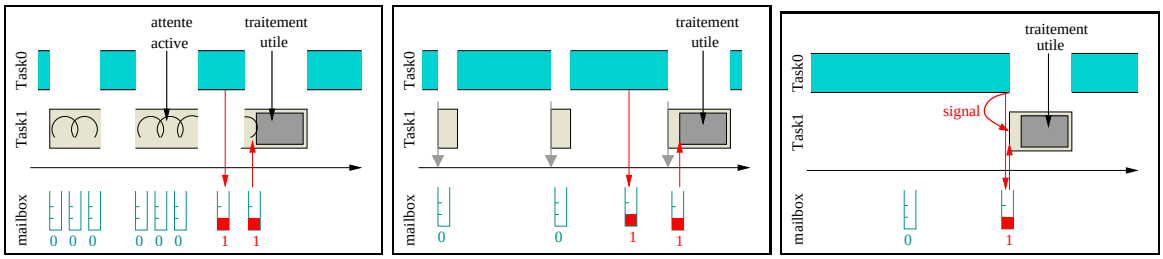
La phase de conception devra définir les profils d'accès aux ressources passives. On peut distinguer (voir la figure 2.5 pour une illustration) :

l'attente active (*busy waiting* en anglais) qui consiste à tenter d'accéder à la ressource en boucle jusqu'à ce que celle-ci soit disponible.

la consultation périodique (*polling* en anglais), ou **traitement commandé par le temps** (*time-driven* en anglais) qui consiste à tenter d'accéder périodiquement à la ressource. L'attente active est un cas particulier de ce profil d'accès.

la signalisation par événement (*event-driven* en anglais) où la tâche demandeuse de la ressource est mise en sommeil en attente qu'un événement extérieur (*via* le système d'exploitation) ou qu'une autre tâche la réveille, pour lui signaler que la ressource est disponible.

La palette des choix possibles dépend de ce que permet le système d'exploitation. Suivant le profil choisi, la réactivité du système sera plus ou moins bonne, et le com-



(a) Attente active (b) Consultation périodique (c) Commandé par événement

FIG. 2.5: Profils d'accès aux ressources

portement temporel du système plus ou moins facile à caractériser : il sera d'autant plus facile de garantir que le système respectera toutes ses contraintes temporelles que le comportement de ses tâches est déterministe (en particulier en ce qui concerne la loi d'arrivée et les temps d'exécution).

Lorsque les tâches ne sont soumises ni à des contraintes de ressources, ni à des contraintes de précédence, elles sont dites *indépendantes*.

2.1.3 Autres contraintes

En dehors des contraintes de temps ou d'accès aux ressources, la spécification de systèmes temps-réel introduit parfois d'autres contraintes. Les plus courantes sont les contraintes de :

localisation ou localité : précisent si les tâches doivent s'exécuter sur un processeur donné (par exemple pour accéder à un capteur disponible uniquement sur une machine donnée, ou pour diminuer la quantité de messages réseau échangés) ;

anti-localité : précisent si deux tâches doivent s'exécuter sur des processeurs différents (par exemple quand il s'agit d'introduire de la redondance spatiale dans les traitements afin que le système soit tolérant aux fautes physiques).

2.2 Caractérisation du support d'exécution

La phase d'implantation d'une application temps-réel repose à la fois sur les spécifications que nous venons de présenter, et sur les caractéristiques des supports langage et système pour le temps-réel sous-jacents. Ces dernières contraignent en partie la façon dont l'application doit être modélisée. Il est donc important de connaître ces supports langages et système dès la phase de conception. Nous présentons ici quelques unes des caractéristiques à prendre en compte.

2.2.1 Perception du temps

Un système d'exploitation en général, et tout particulièrement un système d'exploitation temps-réel, doit percevoir l'écoulement du temps, notamment pour prendre ses décisions d'ordonnancement.

Couramment, un système informatique s'appuie sur une *horloge* matérielle spécialisée dans la **génération d'événements** à intervalles réguliers, ce qui fournit au système d'exploitation une base de temps rudimentaire afin d'assurer au moins la vivacité du système. En contrepartie, la résolution de l'échelle de temps, utile aux décisions d'ordonnancement pour le temps-réel, est fonction de la fréquence de l'horloge. Mais, puisque les coûts de traitement associés à la prise en compte de cette horloge ne sont pas négligeables, cette fréquence n'est en pratique pas très élevée (de l'ordre de 1 à 10ms).

Certains systèmes peuvent aussi *consulter* un compteur externe ou interne au processeur, qui possède une résolution beaucoup plus élevée (jusqu'à quelques fractions de nanoseconde). Pour les systèmes temps-réel, cette fonctionnalité permet de prendre des décisions d'ordonnancement de façon plus précise, et permet de prendre en compte des contraintes temporelles beaucoup plus fines.

Dans ces deux cas, la perception du temps est locale à chaque nœud. Ceci peut poser problème lorsque le système temps-réel est distribué sur plusieurs nœuds : en général les horloges locales dévient et se désynchronisent l'une par rapport à l'autre, ce qui empêche de prendre des décisions relatives au temps et qui sont globales au système. Dans ce cas, une perception du temps cohérente entre les nœuds nécessite de connaître ou d'établir les bornes sur les écarts entre les évolutions des horloges locales aux nœuds du système. Le système doit alors soit se fonder sur une horloge commune à tous les nœuds (par exemple utiliser le système GPS), soit utiliser des mécanismes matériels ou logiciels de synchronisation des horloges locales [AP97]. Dans tous les cas, le respect des contraintes temps-réel en distribué imposent que la *précision de la synchronisation des horloges* soit connue lors de la conception.

2.2.2 Supports langage et système

2.2.2.1 Supports langage

Le modèle de système temps-réel sur lequel nous reposons fait apparaître des notions que les langages de programmation savent manipuler directement ou non.

Ainsi, quelques langages savent manipuler des tâches directement (comme Ada ou occam2). D'autres langages (`C`, `C++` par exemple) imposent de recourir directement au système d'exploitation ou au support d'exécution pour les gérer.

De la même manière, certains langages proposent des mécanismes de synchronisation (par exemple : moniteurs en Modula-1, Concurrent Pascal, Mesa ; objets protégés en Ada ; files de messages en occam2 et Ada). Sinon, il est nécessaire de faire directement appel aux services du système d'exploitation et/ou du matériel.

Enfin, il serait désirable que le langage fournisse tous les éléments qui permettent d'évaluer le comportement des tâches, ou, de façon plus réaliste, qu'il permette de l'exprimer : les temps d'exécution (par annotation du nombre d'itérations des boucles

par exemple), les ressources utilisées. Malheureusement, il existe peu de langages dans l'industrie qui supportent cette fonctionnalité ; en général, il s'agit d'une série de restrictions d'un langage courant[1] (C, Ada) et de syntaxes d'annotation (par exemple : annotation du nombre d'itérations d'une boucle) qui sont non standards et spécialisées pour un secteur ou une entreprise particulière (par exemple SPARK Ada pour l'avionique anglaise [Bat98]). Ceci amène en pratique à séparer l'implantation du système, de sa caractérisation, ce qui peut être source d'erreur (le modèle de comportement ne correspond pas à la réalité).

2.2.2.2 Supports système

Quantité de supports d'exécution pour le temps-réel existent tant dans l'industrie (VxWorks, pSos, VRTX, OSE, OS9, LynxOS/LynuxOS, RTEMS, eCos par exemple) qu'au sein de la communauté scientifique (Mars [KFG+92], Maruti [SdSA95], Spring [SRN+98], Hades [CPC+00], RTMach [TNR90], DIRECT [SG97a], Emeralds [Zub98], MaRTE [RH01], DICK [But97], Hartik/S.Ha.R.K. [GAGB01], variantes RTLinux [1][Bar97, WL99b, DFW02, Heu01] par exemple). Des normes pour les interfaces de programmation système existent également qui sont plus ou moins respectées par les systèmes d'exploitation (normes POSIX1003.1b et 1003.1c [fIT93] ou ITRON [6] par exemple).

Nous présentons ici quelques unes des fonctionnalités qu'on retrouve couramment dans les supports d'exécution temps-réel :

Interface de création des tâches. Certains systèmes proposent des mécanismes pour associer la création de tâches à l'occurrence d'événements de l'environnement (création orientée événements, ou *event-driven*). D'autres ne permettent d'implanter que des tâches périodiques (exécution commandée par le temps, ou *time-driven*). Dans tous les cas, il est possible à une tâche d'en créer d'autres.

Gestion des délais. Le système peut proposer une fonctionnalité d'alarme différée relativement au temps processeur occupé par la tâche (indépendamment des préemptions), ou de sommeil pendant une durée donnée relativement à la date de début de la mise en sommeil (*délai relatif* qui accumule les erreurs dues à la résolution de l'horloge système), ou jusqu'à une date absolue (*délai absolu*). L'implantation des tâches périodiques par exemple dépend des méthodes de mise en sommeil disponibles.

Primitives de synchronisation. Les systèmes d'exploitation proposent en général des mécanismes de synchronisation basiques identiques, à savoir les sémaphores valués ou les verrous. Au delà, les systèmes peuvent adhérer à une norme, totalement ou partiellement, dans laquelle figurent une série de mécanismes de synchronisation supplémentaires (eventflag[2], boîte aux lettres, files de messages, verrous en lecture/écriture). Les systèmes peuvent également proposer leurs propres mécanismes de synchronisation, à la place ou en plus de ceux recommandés par les normes auxquelles ils peuvent décider de se conformer (par exemple les CABs de

[1] En général : pas de boucles infinies, récursivité contrôlée.
[2] une variante des *conditions* sous forme d'un bitmap.

DICK, ou *Cyclical Asynchronous Buffers*, qui sont des boîtes aux lettres limitées à un seul élément).

Politique d'ordonnancement. Pour définir la prochaine tâche à élire, le support d'exécution peut reposer sur un calendrier, ou *plan* défini lors de la phase de conception, ou sur un ordonnancement *en-ligne* reposant sur des priorités relatives entre les tâches, ou en fonction de l'ordre d'activation (politique du premier arrivé, premier servi, ou FIFO, par exemple). Le chapitre 3 détaille plus en détails les politiques d'ordonnancement temps-réel les plus courantes.

Profil d'appel à l'ordonnanceur. Le système d'exploitation peut prendre ses décisions d'ordonnancement lors de chaque événement système (blocage sur ressource, interruption matérielle), ou ne les prendre que lors d'une interruption d'horloge (*tick scheduling*). Dans le premier cas, les surcoûts système sont plus élevés, et dans le second cas les gigues d'activation sont plus importantes.

Dans le cas de l'implantation de systèmes temps-réel strict, il est nécessaire de disposer de suffisamment d'informations sur le comportement temporel du système d'exploitation sur lequel on repose. Cela suppose soit de disposer du code source du système d'exploitation et d'être capable de l'*analyser* [CP01a, CP01b] pour en extraire les caractéristiques temporelles pertinentes ou pour en déduire le comportement de l'application dans son ensemble ; soit de disposer des caractéristiques temporelles pertinentes *fournies* avec le système d'exploitation (le plus souvent obtenues par test et mesure, ou *benchmark*) ; soit que l'utilisateur détermine les caractéristiques temporelles en *évaluant* lui-même le système d'exploitation ou l'application dans son ensemble. Parmi les caractéristiques temporelles couramment évaluées, on trouve [fIT93] les coûts liés aux appels système explicites :

Délai de prise de ressource lorsqu'une tâche acquiert un sémaphore, un verrou, un moniteur ou tout autre mécanisme d'exclusion non encore pris par une autre tâche.

Coût de changement de paramètre d'ordonnancement lorsqu'une tâche décide de changer de *priorité* (développé en 3) par exemple.

Coûts temporels d'autres services système tels que ceux de la création, suppression et terminaison de tâche, ou ceux liés au service d'ordonnancement.

D'autres coûts systèmes implicites sont également évalués :

Caractéristiques du service d'horloge système telles que sa résolution, ses paramètres de déviation ou d'irrégularité par rapport à une horloge de référence, les coûts temporels liés à l'appel des primitives de consultation et de modification, ainsi que ceux des primitives de programmation d'un réveil (*timeout*).

Délai de propagation d'un événement de réveil lorsqu'une tâche libère une ressource, signale une condition, ou envoie un événement (signal POSIX par exemple) à une autre tâche.

Coûts temporels des changements de contexte. C'est le temps nécessaire à la préemption d'un travail au profit d'un autre. Il s'agit d'une durée essentiellement dépendante de la configuration matérielle (type de processeur, cadence, taille des caches d'instructions, de données, et de traduction d'adresses par exemple).

Surcoûts temporels liés à la gestion des interruptions matérielles. Ce sont les temps nécessaires au passage de la tâche ou du traitant d'interruption en cours, vers le code de traitement de l'interruption levée ; et inversement. Il s'agit encore d'une donnée essentiellement dépendante de la configuration matérielle.

Autres coûts système. Les normes (comme POSIX1003.1b par exemple) proposent en plus de mesurer les coûts de services plus sophistiqués du système d'exploitation ou du support d'exécution, tels que ceux des services de gestion de fichiers ou d'allocation dynamique de mémoire par exemple.

La connaissance de majorants aux caractéristiques temporelles du système d'exploitation est un des paramètres d'entrée essentiels afin de garantir, avant exécution, qu'un système temps-réel strict respectera toutes ses contraintes temporelles en cours de fonctionnement. Or, certains services du système d'exploitation peuvent avoir des caractéristiques temporelles (telles que le temps d'exécution par exemple) pire-cas déraisonnables, comme par exemple l'allocation dynamique de mémoire [Pua02] ou le service de pagination par zone d'échange (*swap* en anglais), ou de gestion de fichiers sur disque. Ces services ont peu de chances d´être adaptés aux contraintes temporelles de l'ordre de la milliseconde (ce qui est courant en temps-réel), ce qui explique pourquoi ce type de service est rarement utilisé dans le contexte du temps-réel strict.

Chapitre 3

Ordonnancement et Analyse d'ordonnancement

Dans ce chapitre, nous introduisons tout d'abord la problématique de l'ordonnancement en temps-réel, dans le contexte du modèle de système décrit précédemment. Nous citons ensuite une série de résultats scientifiques qui ont été obtenus dans ce domaine central de l'ingénierie temps-réel. Nous commençons par dresser un panorama des contraintes et propriétés d'ordonnancement, avant de donner les propriétés des problèmes algorithmiques associés en termes de complexité, et les grandes classes d'approches courantes (section 3.2). Nous développons ensuite plus avant les propriétés d'une classe particulière d'ordonnancement (à priorités simples ; section 3.3) pour des systèmes constitués exclusivement de tâches temps-réel strict, avec ou sans prise en compte des coûts système, avec ou sans contraintes de ressources, avec ou sans contraintes de précédence. Puis nous levons progressivement les restrictions de ce modèle : support de tâches apériodiques (section 3.4), gestion de la *surcharge* (section 3.5), techniques d'ordonnancement qui ne sont pas à priorités (section 3.6), ordonnanceurs qui s'adaptent à un manque d'informations sur le comportement du système ou de l'environnement (section 3.7). Nous terminons par un aperçu des méthodes d'ordonnancement sur systèmes multiprocesseur ou distribués (section 3.8).

3.1 Problématique de l'ordonnancement en temps-réel

3.1.1 Description

Nous nous concentrons sur une pièce fondamentale d'un système temps-réel : l'ordonnanceur. Pour chaque processeur, l'ordonnanceur est en charge de définir l'*ordonnancement*, qui est une séquence infinie d´éléments du type : `(date, identifiant_travail)`. Chaque élément correspond à l'*élection* de la tâche à exécuter, puis à un *changement de contexte*, qui fait suite à une *décision d'ordonnancement* par l'ordonnanceur.

La problématique de l'*ordonnancement* revient à définir comment agencer les exécutions des tâches sur chaque processeur. Un ordonnancement est *faisable* (*feasible schedule* en

anglais), si **toutes** les contraintes temporelles et de ressources sont respectées (voir section 3.1.2 ci-dessous). Suivant les caractéristiques du modèle de tâches et du système, un ordonnancement faisable peut exister ou non. Lorsqu'un tel ordonnancement existe, le système temps-réel est dit *ordonnançable* (*schedulable* en anglais). Un système est dit *ordonnançable par un algorithme donné* si l'ordonnancement qui en résulte est faisable.

L'objectif que poursuit l'*analyse statique hors-ligne* est d'établir l'ordonnançabilité du modèle, et/ou la faisabilité d'un l'ordonnancement donné. Ce sont les propriétés des problèmes étudiés lors de cette phase d'analyse, et quelques solutions, que nous présentons dans la suite de cette étude.

3.1.2 Critères et métriques usuels de spécification de contraintes de validité

Pour le temps-réel strict, le critère de validité fondamental est que toutes les tâches temps-réel strict respectent toutes leurs contraintes temporelles en toute circonstance nominale de fonctionnement. Au delà de cette contrainte fondamentale que nous développons plus loin dans le présent chapitre, différents mécanismes peuvent être comparés par des mesures quantitatives sur leur comportement, en s'inspirant des travaux en temps-réel souple.

En temps-réel souple, les contraintes de validité peuvent faire appel à des données numériques caractérisant le comportement du système. Suivant la complexité du système, il est possible de les obtenir par analyse (le plus souvent statistique) ou par évaluation (le plus souvent par simulation). Parmi les métriques les plus courantes, citons :

le taux de respect : c'est la proportion de travaux des tâches qui respectent leurs contraintes temporelles. *Hit ratio* en anglais. Lorsque les tâches doivent subir un test préventif lors de leur activation et avant leur exécution, afin de vérifier que leurs contraintes temporelles pourront être respectées sans remettre en cause le reste du système (*test d'acceptation*, voir 3.4.2), on parle de *taux de garantie* (*guarantee ratio* en anglais).

la valeur dégagée : on associe à l'exécution de chaque tâche une *fonction de valeur*, et on mesure la somme (ou une autre fonction) des valeurs obtenues pendant le déroulement du système [BSS95]. *Hit value ratio* en anglais. Par exemple, on peut représenter une tâche temps-réel ferme par la fonction de valeur suivante : si la tâche se termine avant son échéance, elle dégage la valeur $X > 0$, et 0 sinon. Un cas particulier est celui où X vaut 1, auquel cas la valeur dégagée (*completion count* en anglais) est directement liée au taux de respect. D'autres fonctions de valeur font intervenir le temps d'exécution (*effective processor utilization* en anglais) [BHS01], ou d'autres paramètres [AB99].

le débit : c'est le nombre de travaux d'une tâche qui respectent leurs contraintes temporelles par fenêtre de temps donnée [WS99b]. *Throughput* en anglais. En multimédia par exemple, il peut s'agir de la mesure du nombre d'images rendues par seconde (*fps* pour *frames per second*).

le seuil d'utilisation : c'est l'utilisation du processeur au dessus de laquelle des tâches commencent à dépasser leurs contraintes temporelles [VJP97, KAS93]. *Utilization*

breakdown en anglais.

les caractéristiques temporelles (voir 2.1.1.2) telles que le retard, la gigue de démarrage, la laxité, le temps de réponse par exemple. Le cas particulier de la validation de tâches temps-réel strict consiste à garantir au moins que le retard maximal pour ces tâches est négatif ou nul.

les coûts systèmes (voir 2.2.2.2) tels que les coûts d'ordonnancement par exemple.

La majorité de ces métriques évoluent au cours du temps. L'évaluation du modèle, ou sa validation font alors intervenir les propriétés de leur distribution, telles que le maximum, le minimum, ou les caractéristiques statistiques (moyenne, écart-type par exemple). Le mode d'évaluation s'intéresse en général au régime permanent (stimuli de l'environnement réguliers), et plus rarement aux régimes transitoires [LSA+00] (délai de rétablissement, sur-utilisation temporaire). Lorsqu'il s'agit de proposer un nouveau mécanisme système pour le temps-réel, on le soumet à ces techniques d'évaluation, en intégrant dans le modèle ou dans l'implantation à évaluer soit des jeux de tests issus de systèmes réels (tels que ceux présentés dans [CFBW93, ABR+93, Tin92a, Aud91, Bur93, TC94, TBW92a, BWH93]), soit des jeu de tests générés aléatoirement.

Pour des spécifications données qui émettent des contraintes sur ces paramètres, la validation peut se faire à deux niveaux du cycle de développement : au niveau du *modèle* issu de la conception (évaluation *a priori*), et au niveau de l'implantation (évaluation *a posteriori*). Nous nous intéressons dans la partie suivante à l'évaluation et à la validation pour ces deux étapes, et présentons quelques méthodes.

3.2 Caractéristiques des problèmes et des méthodes d'ordonnancement

Après avoir introduit quelques éléments de terminologie (section 3.2.1), nous décrivons les caractéristiques de quelques problèmes d'ordonnancement (section 3.2.2), et présentons les grandes classes de solutions courantes à ces problèmes (section 3.2.3).

3.2.1 Classes des problèmes

On indique ici les caractéristiques liées aux systèmes cible de l'ordonnancement ou à l'ordonnancement lui-même, qui forment les paramètres du problème d'ordonnancement qu'on cherche à résoudre.

Monoprocesseur/multiprocesseur

L'ordonnancement est de type monoprocesseur si toutes les tâches ne peuvent s'exécuter que sur un seul et même processeur. Si plusieurs processeurs sont disponibles dans le système, l'ordonnancement est multiprocesseur.

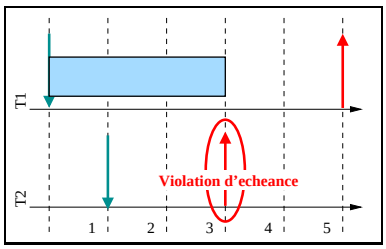
Préemptif/non-préemptif

L'ordonnancement (en-ligne ou hors-ligne) est préemptif lorsque les préemptions (voir 2.1.1.1) sont autorisées. On peut remarquer que dans le cas des ordonnanceurs monoprocesseur, la problématique de la synchronisation sur ressource n'existe pas en non-préemptif puisque il ne peut pas y avoir de concurrence sur l'accès aux ressources en l'absence de préemption.

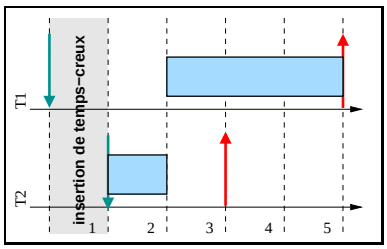
Avec/Sans insertion de temps creux : oisif/non oisif

L'ordonnanceur peut avoir la propriété suivante : à partir du moment ou au moins une tâche est prête à être exécutée, alors l'ordonnanceur en élit forcément une parmi elles. Dans ce cas, l'ordonnanceur est *non oisif*, c'est à dire qu'il fonctionne *sans insertion de temps creux* (*non-idling* ou *work-conservative* en anglais).

Dans certains cas, un système peut n'être ordonnançable qu'en n'élisant aucune tâche pendant un certain temps, alors qu'il en existe au moins une prête à être exécutée. Dans ce cas, l'ordonnanceur est *oisif*, c'est à dire qu'il fonctionne par *insertion de temps creux* (*with inserted idle times* en anglais). Par exemple, le système de tâches de la figure 3.1 (les notations sont introduites figure 2.4, section 2.1.1.2) n'est pas ordonnançable en non-préemptif non oisif (figure 3.1(a)), alors qu'il est ordonnançable en non-préemptif oisif (figure 3.1(b)). On peut remarquer que la définition d'un ordonnancement oisif repose sur la connaissance des activations à venir.



(a) Non-oisif



(b) Oisif

	T1	T2
Date d'activation	0	1
WCET	3	1
Échéance absolue	5	3

FIG. 3.1: Intérêt de l'ordonnancement oisif en non-préemptif

En-ligne/hors-ligne

Un ordonnancement hors-ligne (*off-line* en anglais) signifie que la séquence d'ordonnancement est prédéterminée à l'avance : dates de début d'exécution des tâches, de préemption/reprise éventuelles. En pratique, l'ordonnancement prend la forme d'un *plan hors-ligne* (ou *statique*), exécuté de façon répétitive (on parle aussi d'ordonnancement *cyclique*, qui définit le *cycle majeur*) : à chaque top d'horloge (on parle de cycle *mineur*) une tâche particulière du cycle majeur courant est exécutée jusqu'à terminaison ou jusqu'au top d'horloge suivant.

Un ordonnancement en-ligne correspond au déroulement d'un algorithme qui tient compte des tâches effectivement présentes dans la *file d'ordonnancement* (*run-queue* en anglais) lors de chaque décision d'ordonnancement. Les ordonnanceurs en-ligne peuvent reposer sur la construction de plans d'ordonnancements en cours de fonctionnement, auquel cas ils sont dits à plan *dynamique* [KSSR96]. Mais plus couramment, ces ordonnanceurs sont fondés sur la notion de priorité (voir 3.2.3.2).

Il est par ailleurs possible de transformer un ordonnancement à plan statique en ordonnancement en-ligne (par exemple à priorité statique préemptif [DFP01] ou non-préemptif [Bat98], ou à priorité dynamique [CA97]).

Centralisé/distribué

Lorsqu'un système est distribué, l'ordonnancement (en-ligne) est distribué si les décisions d'ordonnancement sont prises par un algorithme *localement en chaque nœud*.

Il est centralisé lorsque l'algorithme d'ordonnancement pour tout le système, distribué ou non, est déroulé *sur un nœud privilégié*.

Statique/dynamique

Les ordonnanceurs *statiques* fondent leurs décisions d'ordonnancement sur des paramètres assignés aux tâches du système avant leur activation. À l'inverse, les ordonnanceurs *dynamiques* fondent leurs décisions sur des paramètres qui varient en cours de fonctionnement du système.

Optimal/non optimal

Par définition, un algorithme d'ordonnancement *optimal* [GRS96] pour une classe de problème d'ordonnancement donné est tel que : si un système est *ordonnançable* (voir 3.1) par au moins un algorithme de la même classe, alors le système est ordonnançable par l'algorithme d'ordonnancement optimal. En conséquence, si un système n'est pas ordonnançable par un ordonnanceur optimal d'une classe donnée, alors il ne l'est par aucun ordonnanceur de la même classe.

3.2.2 Complexité des problèmes

Le problème qui consiste à définir un ordonnancement peut être ramené à un problème d'optimisation. Par exemple, beaucoup de travaux s'intéressent à minimiser le retard maximal parmi toutes les tâches ("$L_{\max}$" dans la suite). Car une fois ce retard maximal minimisé, la correction temporelle du système est équivalente à vérifier que ce retard maximal est négatif ou nul.

Nous nous intéressons ici à la complexité [Coo71, Coo83, GJ79] de quelques problèmes pertinents pour l'ordonnancement [SSNB94][2].

Pour exprimer ces problèmes, une notation abrégée a été introduite [Rie98, SSNB94, But97, GLLK79, Mer92], que nous n'utilisons pas ici.

3.2.2.1 Ordonnancement monoprocesseur

Les tableaux 3.1 et 3.1 indiquent la complexité de quelques problèmes d'ordonnancement respectivement non-preéemptifs et préemptifs sur monoprocesseur.

monoprocesseur non-préemptif						
tâches concrètes	**temps creux**	**précédences**	**ressources**	**problème**	**complexité**	
non	non	non	indifférent	minimiser $L_{\max}$	polynomiale	voir 3.2.3.2
non	oui	non		ordonnançabilité	NP-complet	[GJ79, HV95]
oui	non	non		minimiser $L_{\max}$	NP-complet	[GJ79, JSM91]
oui	non	*series-parallel*[a]		minimiser $L_{\max}$[b]	polynomiale	[SSNB94, Law78, GL95]
oui	non	oui		minimiser $L_{\max}$	NP-difficile	[SSNB94, Law78]
synchrones	non	oui		minimiser $L_{\max}$	polynomiale	[SSNB94, Law73]

monoprocesseur préemptif						
tâches concrètes	**temps creux**	**précédences**	**ressources**	**problème**	**complexité**	
non	non	non	verrous seuls	ordonnançabilité	NP-difficile	[Mok83]
non	non	non	oui	ordonnancement optimal	impossible	[Mok83], voir 3.3.6
non + temps d'exécution tous égaux	non	non	oui	minimiser $L_{\max}$	polynomiale	[Mok83]
indifférent	non	non	non	minimiser $L_{\max}$	polynomiale	voir 3.2.3.2
indifférent	non	oui	non	minimiser $L_{\max}$	polynomiale	voir 3.2.3.2
indifférent	non	oui	oui	ordonnancement	NP-difficile	[SSNB94]

TAB. 3.1: Complexité de problèmes d'ordonnancement monoprocesseur

[a] Ce type de graphe rassemble les arbres convergents et divergents, mais n'est pas du tout adapté aux cas courants où des échanges de messages peuvent se faire en travers de l'arbre [SSNB94].

[b] Ou toute fonction de chaînes de travaux vers $\mathbb{R}$ qui possède la propriété : $f(\alpha) \leq f(\beta) \Rightarrow f(a..\alpha\beta...b) \leq f(a..\beta\alpha..b)$.

On peut remarquer que le problème à résoudre est d'autant plus complexe que les préemptions sont interdites ou que les temps creux sont autorisés (insérer des temps creux dans l'ordonnancement revient à être capable de prévoir les activations à venir).

En pratique, pour résoudre les problèmes difficiles qui font intervenir des ressources et éventuellement des contraintes de précédences, il existe des ordonnancements statiques [JD90], des approches sub-optimales (telles que le *kernelized monitor* [Mok83]) ou des heuristiques (comme [Man98, MMM00b, RS94] par exemple), ou des protocoles pour accéder aux ressources en utilisant des algorithmes d'ordonnancement classiques (comme nous le verrons en 3.3.6).

3.2.2.2 Ordonnancement multiprocesseur

Nous nous intéressons ici aux systèmes multiprocesseur simples dans lesquels l'ordonnancement est centralisé et pour lesquels on suppose les temps de communication entre les nœuds nuls. Le tableau 3.2 présente quelques résultats de complexité : le problème considéré est celui qui consiste à déterminer un ordonnancement **hors-ligne** non-préemptif faisable. La plupart des résultats présentés dans ce tableau sont issus de [SSNB94].

multiprocesseur non-préemptif hors-ligne [SSNB94]				
Nombre de Processeurs	**ressources**	**précédences**	**temps d'exécution**	**complexité**
2	non	quelconques	tous égaux	polynomial
2	non	non	quelconques	NP-complet
2	non	quelconques	2 valeurs possibles	NP-complet
2	oui	non	tous égaux	polynomial [RS94]
2	1	forêt	tous égaux	NP-complet
3	1	non	tous égaux	NP-complet
N	non	forêt	tous égaux	polynomial
N	non	quelconque	tous égaux	NP-complet

TAB. 3.2: Complexité d'ordonnancement non-préemptif hors-ligne pour multiprocesseur

Contrairement au cas monoprocesseur, en ordonnancement multiprocesseur hors-ligne, pour tout ordonnancement préemptif du système qui dégage une *valeur* donnée, on peut toujours trouver des ordonnancements non-préemptifs qui dégagent une valeur encore plus faible [McN59]. C'est le cas par exemple si la mesure de la valeur de l'ordonnancement est la somme des temps d'exécution des tâches correctement ordonnancées. En multiprocesseur, autoriser les préemptions ne facilite donc rien, et rajoute des surcoûts à l'exécution (liés aux changements de contexte). Ainsi, le résultat de complexité sur la configuration la plus simple en préemptif hors-ligne [Law83] reflète bien cette propriété :

> **préemptif, N processeurs, tâches quelconques, sans ressource, sans précédence** : la définition d'un ordonnancement hors-ligne qui minimise le nombre de tâches en retard est un problème NP-difficile.

En ce qui concerne l'ordonnancement **en-ligne** pour multiprocesseur, il est prouvé [Mok83] qu'aucun algorithme d'ordonnancement en-ligne ne peut être optimal sans avoir la connaissance complète à l'avance de toutes les échéances, de tous les temps d'exécution, et de toutes les dates de démarrage des tâches. En pratique, on utilise des heuristiques (comme [Man98, MMM00b, RS94] par exemple) pour résoudre ces problèmes, comme nous le verrons en 3.8.3.3.

3.2.3 Solutions aux problèmes d'ordonnancement

Dans la suite, nous nous intéressons d'abord au problème le plus simple : l'ordonnancement monoprocesseur sans ressource et sans précédence. Nous présentons ici les solutions simples les plus courantes, et verrons comment ces solutions peuvent être aménagées pour supporter des systèmes avec ressources et contraintes de précédence.

3.2.3.1 Ordonnancement hors-ligne

L'ordonnancement hors-ligne repose sur la définition d'un plan statique. Cette politique d'ordonnancement possède l'avantage de se contenter d'un support d'exécution très réduit (l'accès à une base de temps suffit). Par là même, elle introduit un surcoût d'exécution lié à l'ordonnancement très faible [KFG+92] (il suffit que l'une horloge système incrémente la position courante dans le plan d'ordonnancement), car les traitements pour l'établissement du plan (comme [SA00, CCS02, Foh94] avec des algorithmes *ad hoc*, optimaux [JD90], ou par optimisation multi-critères [EJ00] par exemple) sont effectués hors-ligne, et peuvent se permettre d'être très complexes. En particulier, ce type d'ordonnancement se prête bien aux cas de systèmes complexes tels que les systèmes temps-réel distribués [SRG89] (le problème principal restant alors la synchronisation des horloges locales pour un déroulement cohérent du plan). Également, l'environnement n'influence pas le déroulement du plan, ce qui rend le système robuste car immunisé contre une partie des dépassements d'hypothèse dus à la mauvaise modélisation de l'environnement. Enfin, la garantie que le système respectera toutes ses contraintes temporelles est une propriété facile à appréhender : il "*suffit*" que le plan soit vérifié hors-ligne, et que les évaluations des temps d'exécution soient correctes. Ceci fait de cette solution celle qui paraît la plus rigoureuse et la plus stricte ; nous verrons cependant que dans les cas de systèmes centralisés simples il existe des solutions en-ligne qui offrent tout autant de garanties.

En contrepartie, il est nécessaire de se ramener à des plans périodiques, ou à des plans qui supposent connues toutes les dates d'arrivées (éventuellement non périodiques) de toutes les tâches. Cela conduit à sur-contraindre le système, en particulier quand on doit traiter des tâches périodiques de période non multiple du cycle mineur, ou des événements de fréquence d'occurrence maximale élevée mais de faible fréquence moyenne. Au mieux, ces contraintes excessives entraînent un surdimensionnement du système, et au pire l'impossibilité de déterminer un ordonnancement faisable. De plus, les plans générés sont très sensibles aux petites modifications du code ou des fonctionnalités qui peuvent être faites (ajout d'une tâche supplémentaire par exemple) : ces modifications affectent le plan dans son ensemble (non localité), ce qui pose problème d'intégration lorsque plusieurs acteurs (entreprises, ingénieurs) coopèrent sur le même projet (contraire aux contraintes de confidentialité par exemple). Un autre problème qui se pose est que le plan doit être capable de définir l'ordonnancement sur une durée égale à l'*hyperpériode* des tâches, qui peut être très grande, ce qui implique une grande consommation mémoire pour stocker le plan, inadaptée aux petits systèmes embarqués.

Il existe cependant des ordonnancements par plan statique qui peuvent être complétés par un algorithme d'ordonnancement en-ligne (comme [Foh94, IF99], par une méthode

similaire à la réquisition de temps creux, que nous verrons en 3.4.1.2), afin de prendre en compte l'activation de tâches sporadiques ou apériodiques par exemple.

Nous ne détaillons pas davantage ce type d'ordonnancement dans ce travail. Dans la suite, nous intéressons aux algorithmes d'ordonnancement en-ligne.

3.2.3.2 Ordonnanceurs en-ligne

La majorité des ordonnanceurs temps-réel en-ligne reposent sur la notion de *priorité* : lors de chaque décision d'ordonnancement, la tâche de plus haute priorité est élue. Les priorités peuvent être attribuées hors-ligne, auquel cas l'algorithme d'ordonnancement est dit *à priorité statique* ou *fixe* ; ou en-ligne, auquel cas il est dit *à priorité dynamique*. [1]

Ce type d'ordonnanceur est plus complexe que les ordonnanceurs à plan statique : les décisions d'ordonnancement consistent à élire la tâche prête de plus haute priorité. Dans le cas des ordonnanceurs à priorité dynamique, il s'agit en plus de déterminer les priorités relatives des différentes tâches prêtes, ce qui peut se révéler coûteux en terme de ressource processeur.

D'autre part, ce type d'ordonnanceur est plus exigeant en terme de services que doit fournit le support d'exécution : celui-ci doit entre autre pouvoir gérer des files de tâches à ordonnancer, et doit pouvoir déterminer quand une décision d'ordonnancement est nécessaire. Il en résulte par là même des appels à l'ordonnanceur plus fréquents qu'en ordonnancement hors-ligne, ce qui accroît encore le surcoût d'exécution engendré. En pratique, les supports d'exécution les plus courants et les standards disponibles les plus utilsés, fournissent les fonctionnalités nécessaires à l'ordonnancement à priorité fixe.

Comme nous le verrons par la suite, ces ordonnanceurs ont l'avantage de s'adapter aux évolutions de l'environnement (support des tâches sporadiques, admission de tâches apériodiques), ou du système (caractérisation temporelle incomplète de certaines parties du système par exemple). De plus, les algorithmes simples que nous décrivons ci-après peuvent être étendus pour supporter des modèles de tâches plus riches, des contraintes de précédence, ou des synchronisations de ressources, ce qui leur permet de rivaliser avec les ordonnancements hors-ligne de ce point de vue.

3.3 Ordonnanceurs à priorités simples

Dans cette partie, nous définissons les algorithmes d'ordonnancement à priorité fixe que nous considérons dans la suite (section 3.3.1), avant de donner quelques résultats d'optimalité les concernant (section 3.3.2). Nous indiquons ensuite les conditions d'ordonnaçabilité qui leur sont associées, pour un modèle de système très restrictif (section 3.3.3). Puis nous levons certaines restrictions du modèle (sections 3.3.4), avant de donner quelques extensions pour supporter des contraintes d'ordonnancement supplémentaires

[1] J. Migge [Mig99] remarque que l'ordonnancement à priorité statique correspond à la définition d'une priorité à l'échelle de la tâche (*i.e.* tous travaux confondus), et que l'ordonnancent à priorité dynamique revient la plupart du temps à définir la priorité à l'échelle de chaque travail (c'est le cas pour l'ordonnancement EDF par exemple).

(section 3.3.5), et pour prendre en compte les accès concurrents à des ressources protégées (section 3.3.6). Nous concluons (section 3.3.7) par une petite discussion sur les choix d'ingénierie qui émanent des familles d'ordonnancement évoquées.

3.3.1 Politiques d'ordonnancement à priorités courantes

Nous nous plaçons ici dans le cadre simple de l'ordonnancement monoprocesseur non oisif à priorité, lorsque toutes les tâches sont indépendantes (pas de précédence, pas de ressource).

Les ordonnancements à priorités les plus courants sont les suivants :

EDF (pour *Earliest Deadline First*) : ordonnancement à priorité dynamique. Une tâche est d'autant plus prioritaire que sa date d'échéance absolue est proche de la date courante. Le cas particulier où les tâches sont synchrones est parfois appelé **EDD** (pour *Earliest Due Date*), ou algorithme de Jackson.

LLF (pour *Least Laxity First*) : ordonnancement à priorité dynamique. Les tâches sont d'autant plus prioritaires que leur laxité est faible à la date courante.

RM (pour *Rate Monotonic*) : ordonnancement à priorité statique pour tâches périodiques/sporadiques avec échéance relative égale à la période/délai d'inter-arrivée. Une tâche est d'autant plus prioritaire que sa période est petite.

DM (pour *Deadline Monotonic*) : ordonnancement à priorité statique pour tâches sporadiques. Une tâche est d'autant plus prioritaire que son échéance relative est petite. DM équivaut à RM quand l'échéance relative est égale à la période.

En dépit de leur simplicité, ces ordonnanceurs possèdent des propriétés relativement fortes. C'est ainsi qu'ils servent de base à quantités de variantes, comme nous le voyons par la suite.

3.3.2 Résultats d'optimalité

Les ordonnancements simples précédents ont les propriétés intéressantes suivantes :

1. EDF et LLF sont optimaux parmi les ordonnancements préemptifs non oisifs [LL73]. EDF est également optimal relativement à la minimisation du retard maximal des tâches [But97].
2. EDF est optimal parmi les ordonnancements non-préemptifs non oisifs [GMR95, GRS96].
3. RM est optimal parmi les ordonnancements préemptifs non oisifs à priorité statique, pour tâches périodiques non-concrètes/synchrones ou sporadiques, lorsque l'échéance relative est égale à la période (ou au délai d'inter-arrivée dans le cas sporadique) [LL73].
4. DM est optimal parmi les ordonnancements préemptifs non oisifs à priorité statique, pour tâches périodiques non-concrètes/synchrones ou sporadiques à échéance relative inférieure à la période [ABRW91, But97, LW82].
5. Dans les cas où les tâches périodiques concrètes ne sont jamais synchrones, ni RM, ni DM ne sont optimaux parmi les ordonnancements préemptifs non oisifs à

priorité statique. Dans [Aud91], on donne l'affectation optimale des priorités pour l'ordonnancement préemptif non oisif à priorité statique pour le cas général (*i.e.* qui reste valable quand les échéances relatives et les périodes ne sont pas reliées).

6. DM est optimal parmi les ordonnancements non-préempifs non oisifs à priorité statique, pour tâches périodiques non-conrètes/synchrones ou sporadiques à échéance relative inférieure à la période [Bat98].

Pour l'ordonnancement à priorité dynamique, les démonstrations utilisent le fait que si on dispose d'un ordonnancement faisable, alors intervertir 2 tâches pour respecter l'ordre d'ordonnancement EDF ou LLF conserve la faisabilité de l'ordonnancement. Le même type de démonstration est utilisé avec les priorités des tâches pour démontrer l'optimalité de RM/DM.

3.3.3 Quelques conditions de faisabilité

Dans la suite, les notations suivantes seront utilisées : dans le système Γ de N tâches, chaque tâche $\tau_i, i \in [1, N]$ possède un temps d'occupation[2] sur le processeur C_i et une échéance relative D_i. Si τ_i est périodique, T_i représente la période. Si τ_i est sporadique, T_i représente le délai d'inter-arrivée.

Le taux d'utilisation de ce système de tâches s'écrit alors : $U = \sum_{i=1}^{N} \frac{C_i}{T_i}$, et on rappelle qu'une condition nécessaire pour que le système soit faisable est $U \leq 1$ (voir 2.1.1.2).

3.3.3.1 EDF préemptif

Pour un ensemble de tâches périodiques tel que $\forall i, D_i = T_i$, la condition nécessaire et suffisante de faisabilité [LL73] est :

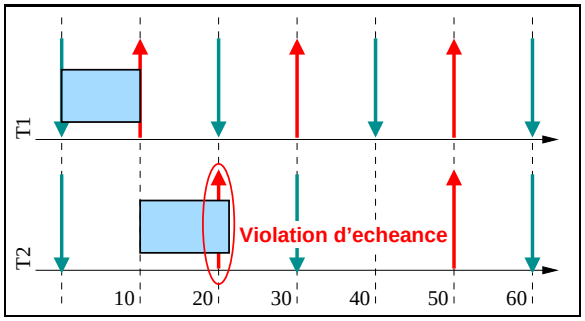
$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{3.1}$$

Cette condition reste nécessaire et suffisante dans le cas où $\forall i, D_i \geq T_i$ [BMR90].

Pour un ensemble de tâches périodiques tel que $\forall i, D_i \leq T_i$, la condition précédente n'est bien sûr plus suffisante puisque D_i peut être arbitrairement petit, comme le prouve l'exemple de la figure 3.2. Dans ce cas, une condition suffisante de faisabilité analogue existe, qui revient à considérer le système comme étant constitué de tâches sporadiques de délai inter-arrivée D_i, et qui s'écrit $\sum_{i=1}^{N} \frac{C_i}{D_i} \leq 1$. Mais cette condition pessimiste n'est pas nécessaire comme le prouve l'exemple de la figure 3.3.

Dans [BMR90] et [Spu96], deux conditions nécessaires et suffisantes pour la faisabilité d'un ensemble de tâches périodiques avec D_i et T_i arbitraires sont données. Dans [BMR90], il s'agit du déroulement de l'ordonnancement par l'intermédiaire de la

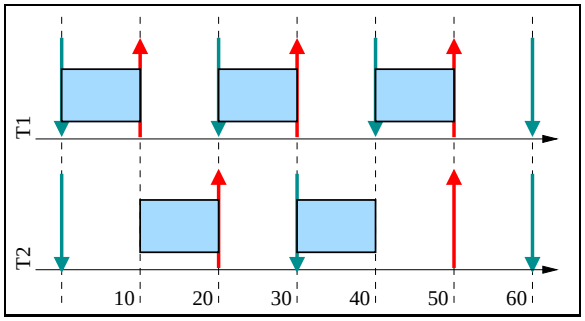
[2] Pour le moment, nous considérons qu'il s'agit du temps d'exécution effectif, toujours constant. En 3.3.4, nous nuancerons cette définition.



	τ_1	τ_2
T_i	20	30
C_i	10	11
D_i	10	20

$\Rightarrow U = 86.67 < 1$

FIG. 3.2: Système de 2 tâches périodiques synchrones non ordonnançable par EDF



	τ_1	τ_2
T_i	20	30
C_i	10	10
D_i	10	20

$\Rightarrow \sum_{i=1}^{N} \frac{C_i}{D_i} = 1.5 > 1$

$(U = 83.33 < 1)$

FIG. 3.3: Système de 2 tâches périodiques synchrones ordonnançable par EDF

demande en capacité processeur (*processor demand* en anglais) $h(t)$, définie par :

$$t \in \mathbb{R}^+, h(t) = \sum_{i=1}^{N} C_i . \max\left[0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right] = \sum_{D_i \leq t} C_i . \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right)$$

La condition de faisabilité nécessaire et suffisante pour le système de tâches est alors :

$$U \leq 1 \ \wedge \ \forall t \in \mathbb{R}^+, \ h(t) \leq t \tag{3.2}$$

Vérifier cette condition pour toutes les valeurs de t est infaisable en pratique. Dans [BMR90] puis [GRS96], on indique comment restreindre correctement le domaine de variation de t (une fraction discrète de l'hyperpériode), mais la complexité reste élevée (bien que la taille du domaine de variation de t soit pseudo-polynomiale dans la majorité des cas).
Dans [Spu96], il s'agit du calcul du temps de réponse pire cas rt_i pour chacune des tâches, reposant sur le calcul de l'interférence qu'une tâche peut subir suite à l'activation de tâches plus *urgentes* (on parle de *busy period*). Une fois ce temps déterminé, il suffit de vérifier que $\forall i, rt_i \leq D_i$. La complexité de ce calcul reste élevée.

En pratique, dans le cas général (T_i et D_i arbitraires) on utilise souvent la condition suffisante (*i.e.* pessimiste) plus simple suivante :

$$\sum_{i=1}^{N} \frac{C_i}{\min(D_i, T_i)} \leq 1$$

3.3.3.2 EDF non préemptif

Considérons d'abord le cas non oisif. La condition nécessaire et suffisante de faisabilité pour un ensemble de tâches concrètes apériodiques correspond à dérouler l'ordonnancement. Dans le cas de tâches apériodiques non-concrètes, une condition nécessaire et suffisante en $O(N^2)$ est donnée dans [GMR95].

Dans le cas de tâches sporadiques ou périodiques non-concrètes, une condition nécessaire et suffisante pseudo-polynomiale est donnée dans [JSM91] et repose sur la demande en ressource processeur.

Dans le cas de tâches périodiques concrètes, la même condition devient suffisante seulement, et le problème qui consiste à déterminer l'ordonnançabilité du système est NP-complet [GMR95, JSM91] (des conditions de faisabilité dans le cas synchrone et non synchrone sont données dans [GMR95]).

Puisque l'ordonnancement non oisif peut être vu comme un cas particulier d'ordonnancement oisif, toutes ces conditions de faisabilité restent valables dans le cas oisif, mais ne sont plus que suffisantes seulement. Dans le cas oisif (NP-complet en général), [GMR95] montre qu'à partir d'un ordonnancement oisif faisable, on peut dériver un ordonnancement par EDF oisif ; il en est déduit un algorithme de type *branch and bound* pour la détermination d'un ordonnancement EDF oisif (complexité exponentielle).

3.3.3.3 Ordonnancement préemptif à priorité statique

Dans ce paragraphe et le suivant, les tâches τ_i sont classées par ordre de priorité π_i décroissantes : $i < j \Rightarrow \pi_i \geq \pi_j$ ($\pi_i = 0$ étant la plus faible priorité possible) ; dans le cas d'une affectation des priorités suivant RM ou DM, $\pi_i = \frac{1}{\min(D_i,T_i)}$. Dans toute la suite, on ne considère que les ordonnancement à priorité fixe non oisifs.

Une première condition de faisabilité nécessaire simple est $U \leq 1$ (puisque EDF est optimal parmi tous les ordonnancements non oisifs).

Dans [LL73], la condition suffisante suivante sur la faisabilité d'un ensemble de tâches périodiques synchrones est montrée, qui suppose l'affectation des priorités suivant RM ou DM :

$$h \sum_{i=1}^{N} \frac{C_i}{\min(D_i, T_i)} \leq N.(2^{\frac{1}{N}} - 1) \tag{3.3}$$

Cette condition suffisante est adaptée dans [SLL93] pour le cas où $\forall i, D_i \geq T_i$ par modification de l'algorithme RM à 2 séries de priorités, et s'écrit $U \leq 1$ dans le cas particulier de tâches harmoniques (la condition devient alors nécessaire et suffisante).

Dans [LL73], il est également montré la propriété fondamentale selon laquelle la pire configuration de tâches périodiques non-concrètes (*i.e.* les pires temps de réponse) est obtenue pour le premier travail des tâches lorsque toutes les tâches sont activées au même instant (*instant critique*, correspondant à un ensemble de tâches synchrones). Il en découle que la condition 3.3 reste une condition de faisabilité suffisante pour des ensembles de tâches périodiques non-concrètes ou sporadiques.

Une première condition pseudo-polynomiale nécessaire et suffisante pour l'ordonnançabilité de tâches périodiques non-concrètes ou sporadiques est donnée dans [LSD89]. Cette condition, appelée TDA (pour *Time Demand Analysis*), fait intervenir une formulation de la demande en ressource processeur pour chaque tâche, est valable pour le cas simple $\forall i, D_i \leq T_i$ avec affectation des priorités quelconque, et s'écrit :

$$\forall i \in [1 \cdots N], \min_{0<t\leq D_i} \left(\sum_{j=1}^{i} \frac{C_j}{t} . \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1 \tag{3.4}$$

De par la variation "par palier" de la demande en ressource processeur (au gré des activations), on peut remarquer qu'il suffit d'évaluer l'équation 3.4 en un nombre fini de valeurs pour le paramètre t (en l'occurrence l'ensemble $S_i = \{kT_j | j \leq i, k = 1 \cdots \lfloor \frac{T_i}{T_j} \rfloor\}$).

Une condition nécessaire et suffisante équivalente, mais plus utilisée, pour la faisabilité de tâches périodiques non concrètes ou sporadiques est donnée dans [JP86, Tin92a, Bur94]. Elle repose sur le calcul du temps de réponse pire cas rt_i des tâches, s'intitule RTA (pour *Response Time Analysis*), et s'écrit $\forall i, rt_i \leq D_i$. Le calcul de rt_i ne suppose pas d'affectation des priorités particulières, et peut se calculer sans hypothèse sur la relation entre les T_i et les D_i.

Dans le cas où $\forall i, D_i \leq T_i$ [JP86], rt_i se calcule en prenant en compte les interférences venant des tâches de priorité supérieure : c'est un calcul itératif qui vise à élargir un intervalle de temps jusqu'à ce que celui-ci contienne à la fois le temps

d'exécution de τ_i, et les interférences par les tâches de priorité supérieure (*i.e.* pour une fenêtre de taille w, cette interférence s'écrit $\sum_{j<i} C_j.\lceil \frac{w}{T_j} \rceil$). Lorsque $U \leq 1$ et $\forall i, D_i \leq T_i$, rt_i est ainsi le point fixe de la suite[3] :

$$\begin{aligned} rt_i^{(0)} &= C_i \\ \forall k \geq 1, rt_i^{(k)} &= C_i + \sum_{j<i} C_j . \left\lceil \frac{rt_i^{(k-1)}}{T_j} \right\rceil \end{aligned} \tag{3.5}$$

Dans le cas général [Tin92a, Bur94] (tâches périodiques non concrètes où sporadiques, D_i et T_i non reliés), il faut en plus prendre en compte les interférences de τ_i avec elle-même quand $D_i > T_i$ (*i.e.* dans ce cas, plusieurs travaux de τ_i peuvent être prêts à s'exécuter en même temps).

Lorsque les tâches périodiques sont concrètes, ces conditions d'ordonnançabilité deviennent suffisantes seulement. En effet, lorsque les dates d'activation des premiers travaux des tâches font que les tâches ne sont jamais activées simultanément, le temps de réponse pire cas qui se manifeste effectivement peut être strictement plus petit que celui calculé dans les formules ci-dessus, qui considèrent le pire temps de réponse toutes configurations confondues. Dans [Aud91], en plus de définir l'affectation optimale des priorités, on présente à la fois un test qui permet de savoir si l'ensemble des tâches concrètes peut ou non être synchrone à un moment donné (auquel cas les conditions précédentes sont nécessaires et suffisantes), et le test de faisabilité de l'ensemble de tâches qui tient compte des dates de démarrage des premiers travaux.

3.3.3.4 Ordonnancement non préemptif à priorité statique

Dans le cas général (D_i et T_i non reliés) de tâches périodiques non concrètes ou sporadiques, [GRS96] donne une condition nécessaire et suffisante d'ordonnançabilité d'un ensemble de tâches, analogue à RTA. Il y est également montré que l'affectation des priorités suivant RM ou DM n'est plus optimale dans le cas non-préemptif (sauf si $\forall i, D_i \leq T_i$ et $\forall i \neq j, D_i < D_j \Rightarrow C_i \leq C_j$), mais que l'affectation optimale pour le cas préemptif donnée dans [Aud91] reste optimale dans le cas non préemptif non oisif.

3.3.3.5 Autres ordonnancements à priorité statique

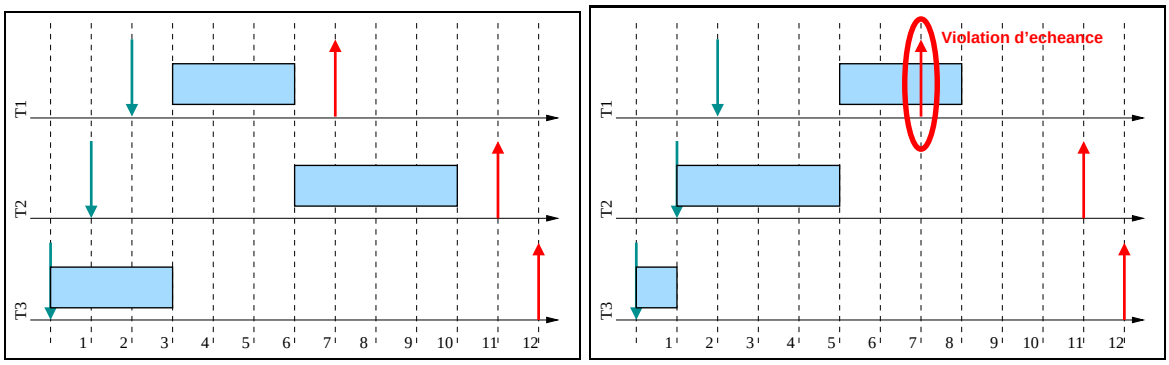
Les travaux [WS02, WS99a, GP96] proposent un modèle de tâches à priorité statique intermédiaire entre le préemptif et le non préemptif qui permet d'allier l'intérêt des deux méthodes. Il s'agit d'associer à chaque tâche deux priorités : la priorité de la tâche tant qu'elle n'a pas commencé son exécution (le *seuil de préemption* ou *preemption threshold* en anglais), et la priorité π_i quand elle a commencé son exécution. Les cas préemptif et non préemptif sont des cas particuliers avec un seuil de préemption respectivement de π_i et 0. Ces travaux proposent à la fois un test de faisabilité, et un algorithme d'affectation optimal polynomial des seuils de préemption.

[3] Cette suite admet un point fixe dès que $\sum_{j \leq i} \frac{C_j}{T_j} \leq 1$, or ceci est vrai puisque $\sum_{j \leq i} \frac{C_j}{T_j} \leq U \leq 1$.

3.3.4 Limitations

Les conditions de faisabilité qui ont été formulées ci-dessus supposent un modèle de système monoprocesseur idéal assez restrictif. En particulier :

- Toutes les tâches sont périodiques ou sporadiques, et supposées connues avant l'exécution du système.
- Toutes les tâches ont un temps d'exécution exactement connu avant exécution, toujours égal à leur temps pire-cas.
  - Dans le cas préemptif (avec tâches indépendantes), cette limitation peut être supprimée : les tâches peuvent terminer avant leur temps d'exécution pire-cas.
  - Dans le cas non-préemptif non oisif, la terminaison plus tôt d'une tâche peut conduire au cas pathologique (on parle d'*anomalie*) illustré figure 3.4, même si le système est ordonnançable lorsque les temps d'exécution pire cas sont considérés.



(a) Ordonnancement faisable (temps d'exécution pire cas pour toutes les tâches)

(b) Violation d'échéance si la tâche τ_3 est plus courte que le temps pire-cas

FIG. 3.4: Anomalie d'ordonnancement par terminaison plus tôt en non préemptif (ordonnancement de type EDF ou DM)

- Aucun mécanisme de synchronisation n'est considéré.
- Les tâches sont supposées sans relation de précédence.
- Les surcoûts inhérents au matériel (*i.e.* temps de changement de contexte par exemple) et/ou liés à l'exécution des services du système d'exploitation (dont ceux de l'ordonnanceur) sont négligés.
- Les activations des tâches périodiques sont supposées être faites à la granularité de l'horloge système.
- Comportement binaire (totalement faisable/non faisable) : aucune mesure de qualité d'aucune sorte n'est formulée lorsque le système n'est pas faisable.
- Les ordonnancements considérés sont non-oisifs.

Dans les paragraphes qui suivent, nous présentons les différentes extensions qui ont été apportés aux ordonnancements à priorité que nous venons de décrire, et qui visent à supprimer tout ou partie de ces limitations.

3.3.5 Extensions

3.3.5.1 Extensions du modèle de tâches

Gigue d'activation

En pratique, entre la date où une tâche périodique doit être théoriquement activée, et la date réelle à laquelle la tâche est mise dans la file des tâches prêtes à être exécutées, il peut y avoir un délai variable dû par exemple à la résolution de l'horloge système (en particulier pour un mode de fonctionnement de type *tick scheduling*, dans lequel les tâches ne peuvent démarrer que lors des ticks d'horloge), aux surcoûts induits par les décisions d'ordonnancement, ou au temps de génération d'un message nécessaire avant le commencement de la tâche. Les conditions de faisabilité données ci-dessus ne sont alors plus suffisantes.

Des extensions à ces conditions existent pour prendre en compte une telle gigue. Le principe est de ramener la gigue à un temps de blocage (voir 3.3.6). Ainsi, pour l'ordonnancement préemptif à priorité fixe, RTA est étendu pour supporter la gigue [ABR+93, Tin92a, Bur94]. Le même type d'extension existe pour les ordonnancements à priorité dynamique [Spu96].

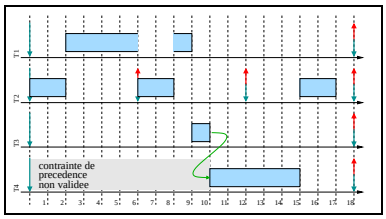
Contraintes de précédence

En ce qui concerne l'ordonnancement préemptif à priorité dynamique, [Law73] propose un algorithme qui minimise le retard maximal tout en prenant les contraintes de précédences en compte (*Latest Deadline First*). Le principe est de laisser en attente une tâche prête à être exécuter tant que sa/ses précédences n'a/n'ont pas été validée(s). Pour l'ordonnancement suivant EDF, la prise en compte des contraintes de précédence peut se faire par modification des échéances relatives [HMT90, SS93, Jef92]. L'ordonnancement résultant est alors optimal vis à vis des ordonnancements préemptifs non oisifs qui prennent en compte les précédences [HMT90].

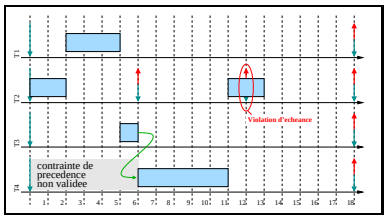
Une anomalie est remarquable quand l'ordonnancement est à priorité fixe : suivant l'affectation des priorités, la terminaison en avance d'une tâche peut rendre non ordonnançable un système qui était ordonnançable lorsque les temps d'exécution étaient tous considérés toujours égaux au temps pire-cas. La figure 3.5 propose une illustration.

On peut également remarquer que l'ajout de contraintes de précédence peut rendre ordonnançable un système qui ne l'était pas. La figure 3.6 propose une illustration.

Des travaux [Tin94, Tin92b, Tin93, ATB93] se sont attachés à étendre les conditions de faisabilité RTA de manière à intégrer des contraintes de précédence (sous la forme d'un délai minimal incompressible entre l'activation d'une tâche et celles de ses successeurs : notion de *transaction*) avec prise en compte des terminaisons en avance sous la forme d'une gigue d'activation en aval des précédences. D'autres études s'intéressent à des chaînes de tâches (*i.e.* 0 ou 1 tâche précède 0 ou 1 tâche) et vérifient le respect



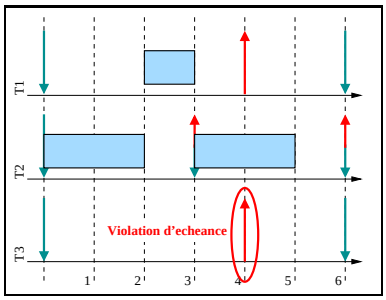
(a) Temps d'exécution égal au WCET



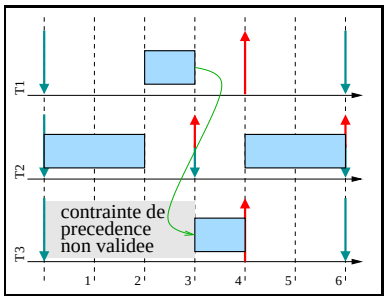
(b) τ_1 termine en avance

	τ_1	τ_2	τ_3	τ_4
C_i	5	2	1	5
$D_i = T_i$	18	6	18	18
π_i	1	2	0	3

FIG. 3.5: Contrainte de précédence $\tau_3 \prec \tau_4$ et dates de terminaison



(a) Sans précédence : infaisable en priorité statique (ici affectation DM)



(b) Prise en compte de la précédence $\tau_1 \prec \tau_3$ (ici affectation $\pi_1 = 0$, $\pi_2 = 1$, $\pi_3 = 2$)

	τ_1	τ_2	τ_3
C_i	1	2	1
D_i	4	3	4
T_i	6	3	6

FIG. 3.6: Contraintes de précédence et ordonnançabilité

des contraintes temporelles parmi tous les entrelacements des chaînes possibles (ou sur un sur-ensemble plus simple) [SNF98] et en tenant compte des variations des temps d'exécution [RRGC02, SGL97].

Modèles de tâches plus riches

Certains travaux s'intéressent à des modèles de tâches plus complexes avec les ordonnanceurs à priorité statique et dynamiques simples. Comme par exemple :

- des tâches "périodiques de façon sporadique" : une tâche est périodique avec une courte période pendant un intervalle donné, puis cet intervalle se répète de façon sporadique. Des travaux existent pour prendre en compte ce modèle de tâche en priorité statique par RTA [Tin93, ABR+93, Tin92a] et dynamique [Spu96].
- des tâches caractérisées par une fréquence d'activation (modèle de *Rate Based Execution* en anglais). Le profil d'activation d'une tâche correspond à la définition d'une densité d'activation de la tâche (une borne supérieure sur le nombre d'activations de la tâche dans une fenêtre glissante de temps donnée). Dans [JG99], une condition nécessaire et suffisante d'ordonnançabilité est donnée pour un ordonnancement suivant EDF, et il est montré qu'aucun ordonnancement à priorité fixe ne peut ordonnancer de telles tâches.
- des tâches avec "échéance avant la fin de la tâche", c'est à dire que la tâche est en deux morceaux : les traitements utiles qui doivent se terminer avant une échéance, et des opérations de maintenance exécutées immédiatement après (prise en compte des coûts de préemption par exemple, ou de libération des ressources système). Les travaux [Bur93, Tin93] proposent une extension de RTA pour intégrer ce modèle de tâches en ordonnancement à priorité statique.
- des tâches qui peuvent avoir des temps d'exécution qui suivent un motif donné (*i.e.* ce sont des vecteurs plutôt que des scalaires) : le modèle *multiframe*. Les travaux [MC96b, MC96a] et [BCM99] proposent d'étendre respectivement la condition de faisabilité 3.3 et RTA pour supporter ce modèle dans l'ordonnancement RM. Ce modèle multiframe est lui-même généralisé dans [BCG+99] afin de supporter des tâches dont les échéances relatives et les périodes sont elles même variables suivant un motif, et une condition nécessaire et suffisante de faisabilité par RTA est proposée.
- des tâches qui peuvent être constituées de sous-tâches avec des échéances relatives internes et des précédences : c'est le modèle de tâches "récurrent" de [Bar98]. Une condition suffisante de non faisabilité en préemptif y est donnée.

3.3.5.2 Prise en compte des coûts système et matériels

Prendre en compte les interruptions matérielles peut se faire en les modélisant sous la forme de tâches de haute priorité. Prendre en compte les changements de contexte (préemptions) peut se faire en rajoutant le coût du changement de contexte (dont l'estimation non pessimiste est un domaine de recherche actif) soit au temps pire-cas de

chaque tâche, soit au temps pire-cas des tâches qui peuvent être préemptées (ce qui est possible avec les algorithmes à priorité fixe, mais pas avec ceux à priorité dynamique).

En ce qui concerne l'ordonnancement à priorité dynamique (préemptif et non-préemptif), les travaux de [JS93, Jef92] et [SP97, Spu96] intègrent le surcoût des interruptions matérielles indépendantes des tâches du système, dans la condition 3.2 fondée sur la demande en ressource processeur.

En ce qui concerne l'ordonnancement à priorité fixe, les travaux de [JS93] et [SNF98] indiquent comment intégrer le surcoût des interruptions matérielles dans TDA et RTA respectivement. Le document [KAS93] détaille comment intégrer différents surcoûts fixes pire-cas (tels que le coût des préemptions, de l'ordonnanceur) dans TDA en *tick scheduling* (aisément transposable à RTA). Les travaux [Tin93, BWH93] précisent comment intégrer les surcoûts liés à l'interruption d'horloge pour du *tick scheduling*, en tenant compte de la durée d'exécution variable de l'ordonnanceur, qui est fonction du nombre de tâches à activer à chaque interruption.

3.3.5.3 Extensions du modèle de système : fonctionnement par phases

Pour certains systèmes, les tâches n'ont pas toutes besoin d'être prises en compte à tout moment. C'est le cas lorsque le système fonctionne par phase : un avion par exemple aura besoin de certaines fonctionnalités pendant le décollage, d'autres pendant la croisière, et d'autres encore à l'atterrissage. Des travaux se sont penchés sur l'intégration de tels modèles de système pour l'ordonnancement en plan statique [Foh94, KNH+97] (recours à un plan intermédiaire de transition immédiat ou différé) et à priorité statique [TBW92b] (extension de RTA) par exemple.

Le fonctionnement par phase pose le problème de la cohérence d'état global du système lors des transitions, dont la résolution est une approche propre à chaque application.

3.3.6 Prise en compte des ressources

L'intégration des contraintes de ressources avec les ordonnancements classiques que nous venons de présenter soulève un certain nombre de problèmes, qui sont résolus à l'aide de *protocoles d'accès aux ressources*.

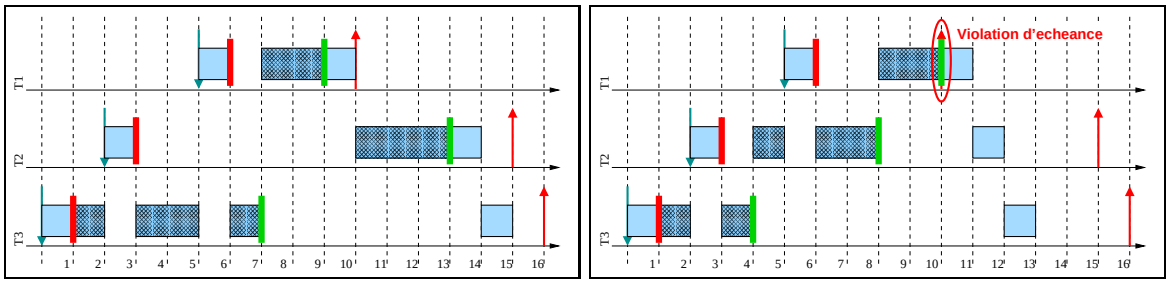
3.3.6.1 Problématique

Comme dans les systèmes informatiques non temps-réel, le principe est d'isoler l'accès à une ou plusieurs ressources sous la forme de *sections critiques*, c'est à dire de portions de code à l'intérieur des tâches.

En temps-réel, ceci s'accompagne de problèmes dus au *temps de blocage* inévitable qu'une tâche doit subir dès qu'elle demande une ressource possédée par une tâche de priorité inférieure.

Anomalie d'ordonnancement

Sans précaution particulière sur l'ordonnancement en dehors du fait que les contraintes de ressource sont prises en compte, la terminaison plus tôt d'une tâche peut rendre infaisable un ordonnancement qui l'était lorsque les temps pire-cas étaient considérés. Il s'agit d'une anomalie similaire à celle présentée en 3.3.4 et illustrée dans la figure 3.7 (une seule ressource en accès exclusif, les traits épais marquent la requête/la libération de la ressource, les zones hachurées marquent la possession de la ressource).



(a) Ordonnancement faisable

(b) Violation d'échéance si la section critique de τ_3 est plus courte

FIG. 3.7: Anomalie d'ordonnancement en présence d'une ressource

Inversion de priorité

Le jeu des conflits liés aux ressources peut entraîner qu'une tâche τ_1 soit mise en attente d'une ressource possédée par une tâche de priorité inférieure τ_3. Mais cette tâche de faible priorité peut être préemptée par une tâche τ_2 de priorité intermédiaire entre celles de τ_1 et de τ_3, et qui n'est pas en conflit ni avec τ_1, ni avec τ_3 pour la ressource. Il en découle que τ_1 a en fait été préemptée par une tâche τ_2 de priorité inférieure et qui n'est pourtant pas en conflit avec elle. Si ce phénomène d'*inversion de priorité* n'a pas été pris en compte au moment de la conception, des dépassements d'échéance sont possibles, comme cela a été le cas pour le célèbre dysfonctionnement de la mission Mars Pathfinder [5].

Comme nous le voyons par la suite, des protocoles de prise de ressource existent pour empêcher les inversions de priorité.

Interblocage

Ce problème n'est pas particulier aux systèmes temps-réel et apparaît lorsque plusieurs ressources sont présentes dans le système. Pour le cas simple à deux tâches τ_1 et τ_2 avec deux ressources en accès exclusif R_1 et R_2, il apparaît lorsque τ_1 possède R_1 et τ_2 possède R_2 et que τ_1 et τ_2 ont à leur tour besoin de R_2 et R_1 respectivement. Ce type de configuration (une boucle) entraîne le blocage permanent de τ_1 et τ_2 ainsi

que la non disponibilté permanente de R_1 et R_2. Ce problème est généralisable sur des boucles de blocage à davantage de tâches et de ressources.

Une solution simple pour empêcher ce problème d'apparaître est d'imposer un ordre d'acquisition des ressources arbitraire lors de la phase de conception. Nous verrons que certains protocoles d'accès aux ressources permettent d'éviter ce problème par d'autres voies.

3.3.6.2 Démarche

Le problème qui se pose en temps-réel en présence d'accès concurrents à des ressources, est l'intégration, dans les algorithmes d'ordonnancement à priorité classiques, d'une borne supérieure sur le temps de blocage d'une tâche par les tâches de priorités inférieures. Les protocoles d'accès aux ressources ont pour rôle de réaliser cette intégration, en rendant les temps de blocage effectivement bornés, et si possible avec une borne réduite.

Une fois la borne B_i sur le temps de blocage que peut subir un travail de τ_i déterminée, l'intégration dans les conditions de faisabilité permet de valider le comportement temporel correct du système, y compris en présence de terminaisons de tâches en avance, et s'écrit :

- Ordonnancement EDF préemptif :
  - Condition analogue à 3.1 [SS93, Bak91] :

$$\forall i \in [1 \cdot N], \Big(\sum_{k=1}^{i} \frac{C_k}{\min(T_k, D_k)} \Big) + \frac{B_i}{\min(T_i, D_i)} \leq 1$$

  - Condition analogue à 3.2 [SS93] :

$$\forall t \geq 0 \wedge \forall i \in [1 \cdot N], \left[\sum_{k=1}^{i} C_k . \Big(\Big\lfloor \frac{t - D_k}{T_k} \Big\rfloor + 1 \Big) \right] + B_i . \Big(\Big\lfloor \frac{t - D_i}{T_i} \Big\rfloor + 1 \Big) \leq t$$

  - Il existe également une condition reposant sur le calcul du temps de réponse pire cas par la détermination de la *busy period* [Spu96].
- Ordonnancement à priorité fixe préemptif :
  - Condition analogue à 3.3 [SRL90] :

$$\forall i \in [1 \cdot N], \Big(\sum_{k=1}^{i} \frac{C_k}{\min(D_k, T_k)} \Big) + \frac{B_i}{\min(D_i, T_i)} \leq i.(2^{1/i} - 1)$$

  - Condition analogue à 3.4 [SRL90]

$$\forall i \in [1 \cdots N], \min_{0 < t \leq D_i} \left[\Big(\sum_{j=1}^{i-1} \frac{C_j}{t} . \Big\lceil \frac{t}{T_j} \Big\rceil \Big) + \frac{C_i + B_i}{t} \right] \leq 1$$

– Condition analogue à 3.5 [Bur94, ABR+93, Tin92a] : le temps de réponse rt_i est le point fixe de la suite

$$\begin{aligned} rt_i^{(0)} &= C_i + B_i \\ \forall k \geq 1, rt_i^{(k)} &= C_i + B_i + \sum_{j<i} C_j . \left\lceil \frac{rt_i^{(k-1)}}{T_j} \right\rceil \end{aligned}$$

Ces tests sont tous suffisants seulement (les tâches peuvent ne subir aucun blocage en réalité), et sont plus ou moins pessimistes.

3.3.6.3 Protocoles d'accès aux ressources

Une première source de blocage qui pose des problèmes sur la détermination des bornes de blocage, correspond à l'inversion de priorité. Si on n'utilisait pas de protocole d'accès aux ressources, alors les temps de blocage seraient considérablement longs (de l'ordre de $B_i = \sum_{i<k\leq N} C_k$ en priorité statique par exemple).

La méthode la plus simple pour résoudre ce problème est d'interdire toute préemption tant qu'une tâche possède une ressource. Les temps de blocage sont ainsi parfaitement déterminés (le maximum des durées de toutes les sections critiques de toutes les tâches de priorité inférieure), mais trop pessimistes (toutes les tâches sont prises en compte, y compris les tâches qui ne rentrent pas en conflit de ressource avec la tâche τ_i considérée).

En ordonnancement à priorité statique, un autre protocole simple est le *protocole à héritage de priorité* (PIP pour *Priority Inheritance Protocol*) [SRL90], qui vise par construction à interdire l'inversion de priorité : quand une tâche de priorité L bloque un travail d'une tâche H de priorité supérieure, elle *hérite* de la priorité H jusqu'à ce qu'elle relâche les ressources pour lesquelles H est bloquée. Ce protocole exhibe des temps de blocage plus petits, mais ne gère pas les interblocages, et peut entraîner des surcoûts à cause des blocages répétés (*blocages en chaîne*) et des préemptions associées à chaque fois qu'une tâche H est bloquée par une tâche L.

Les mêmes travaux [SRL90] présentent un protocole qui résout ces deux limitations : le *protocole à seuil de priorité* ou PCP pour *Priority Ceiling Protocol*. Le principe est d'associer à chaque ressource un *seuil de priorité* égal à la plus grande priorité parmi toutes les tâches qui peuvent la demander. Lorsqu'une tâche demande une ressource, elle n'est autorisée à la prendre que si sa priorité est supérieure aux seuil de priorité de toutes les ressources acquises par d'autres tâches. Si la tâche demandeuse bloque, elle transmet sa priorité (héritage) à la tâche qui possède la ressource demandée. PCP a l'avantage d'empêcher tout interblocage et de limiter le nombre de blocages (1 blocage maximum par travail). Mais ceci se fait au prix d'un effort d'implantation et de conception plus grand : l'algorithme est relativement délicat à implanter, et la définition des seuils de priorité des ressources implique qu'on est capable de préciser hors ligne quelle tâche a besoin de quelle(s) ressource(s).

Afin d'éviter toute préemption due à des blocages, les travaux [Bak91] proposent le *protocole à priorité de pile* (SRP, pour *Stack Resource Policy*). Avec ce protocole, la tâche n'est pas autorisée à démarrer tant que toutes les ressources qui lui sont

nécessaires ne sont pas disponibles, ce qui a pour effet d'éviter tout blocage une fois que la tâche a démarré (sans toutefois supprimer l'éventuel blocage avant le démarrage). Le protocole repose sur la notion de *niveau de préemption* fixé hors-ligne et indépendant du type d'ordonnancement à priorité (statique ou dynamique). Le protocole maintient pour l'ensemble du système le seuil de priorité Π courant, et une tâche n'est autorisée à démarrer que si sa priorité est supérieure à la priorité de la tâche courante, et si son niveau de préemption est supérieur au seuil de priorité Π du système. À chaque fois que la tâche acquiert une ressource, le seuil de priorité du système Π est affecté à la valeur du plus haut niveau de préemption parmi toutes les tâches susceptibles d'acquerir la ressource. Avec SRP, le temps de blocage et les contraintes de conception sont identiques à ceux de PCP, mais le nombre de préemptions pour cause de blocage est nul : il est alors inutile de se soucier de l'accès concurrent aux ressources. De plus, il est possible de considérer des ressources non binaires (telles que des sémaphores, ou des piles).

Ces travaux peuvent également être intégrés dans un système avec ordonnancement à priorité dynamique [Spu96, SS93, Bak91]. Les travaux de Jeffay [Jef92, Jef89] proposent une autre technique, dite *par phases*, pour l'ordonnancement en priorité dynamique. Chaque phase correspond soit à une portion sans acquisition de ressource, soit à une section critique différente. Le principe est de modifier dynamiquement les échéances des tâches à chaque changement de phase, de manière à reproduire un comportement similaire à SRP : on parle de EDF/DDM (pour *Dynamic Deadline Modification*). L'auteur fournit une condition suffisante d'ordonnançabilité du système résultant.

3.3.6.4 Modes d'accès aux ressources optimistes

Plutôt que d'avoir recours à l'acquisition d'une ressource afin de modifier une structure de données (telle qu'une file par exemple), il peut être intéressant (quand c'est possible) d'utiliser des primitives qui effectuent une série d'opérations de façon optimiste (*i.e.* en ne supposant aucun conflit), et qui est répétée si il y a eu des conflits jusqu'à *validation* des opérations. L'avantage de cette méthode est que pour une série d'opérations relativement courte, les surcoûts système sont réduits. Le problème est qu'en temps-réel il est important de savoir borner le nombre de fois où les opérations de mise à jour sont répétées.

On distingue couramment 2 approches pour ce type de mécanisme optimiste [ARJ95] :

Algorithmes *sans attente* (*wait-free* en anglais). Les tâches qui ont validé leurs modifications aident celles qui sont en train d'essayer de valider les leurs, mais qui n'y ont pas encore réussi. Ce mécanisme peut être vu comme une forme d'inversion de priorité, et induit des coûts d'exécution qui peuvent être importants (copie de données en zone de transit).

Algorithmes *sans blocage* (*lock-free* en anglais). Une tâche donnée recommence d'elle-même la série d'opérations de modifications tant qu'elles n'ont pas été validées.

Les travaux [ARJ95] étendent les conditions de faisabilité pour RM et EDF pour les algorithmes sans blocage : le principe est de déterminer une borne supérieure sur le nombre de fois où les opérations de modification seront répétées.

3.3.7 Choix d'ingénierie

Par rapport à l'ordonnancement statique (voir 3.2.3.1), l'ordonnancement à priorité reste une solution simple à implanter, plus souple lorsque des modifications sont introduites, et est moins gourmande en mémoire. En contrepartie, plus il y a de contraintes (ressources, précédences par exemple), moins le processeur peut être effectivement utilisé, et dans certains cas, aucun ordonnancement en-ligne ne peut exister.

Au sein de la famille des ordonnancements en-ligne à priorité, le concepteur doit choisir entre priorité statique et dynamique. Le choix est d'autant plus difficile que les différentes extensions aux modèles simples issus de [LL73] existent en général pour les deux sous-familles.

Les ordonnancements à priorité statique s'intègrent très bien dans un système d'exploitation sans qu'il y ait besoin de lui faire gérer des contraintes de temps : il suffit que le système dispose d'un ordonnanceur à priorité, ce qui est pour le moins courant. En contrepartie, le seuil d'utilisation qu'il est possible d'atteindre (*i.e.* la rentabilité du processeur) est moins élevé qu'en priorité dynamique. De ce point de vue, il existe des versions mixtes (ordonnancements à priorité statique et dynamique combinés) qui effacent la frontière entre ordonnancement à priorité statique et dynamique [Zub98], et qui permettent de faire un compromis entre rentabilité du processeur et complexité d'ordonnancement en-ligne.

D'autre part, en priorité dynamique, lorsqu'une échéance est dépassée, elle affecte **toutes** les autres tâches, quel que soit leur niveau de *criticité*, ce qui peut amener des dépassements d'échéances en cascade non contrôlés (voir section 3.5). De ce point de vue, les ordonnancements à priorité statique ont l'avantage de mieux isoler ces fautes (les tâches de priorité supérieure à la tâche fautive ne sont pas touchées), mais moins encore que les ordonnancements hors-ligne par plan (une tâche programmée à telle date sera toujours exécutée, où l'ensemble du système sera arrêté, que la tâche précédente ait terminé dans les temps ou non).

3.4 Ordonnancement avec sous-partie dynamique

Un système temps-réel peut être en charge du contrôle d'un procédé par exemple, et en même temps avoir à fournir une palette de services sans contrainte temporelle (affichage d'une interface graphique, exécution des services d'un système d'exploitation généraliste par exemple [Bar97, GAGB01]), ou avec contrainte temporelle mais facultatifs (affichage de statistiques sur le procédé par exemple).

On se place toujours dans le cadre des ordonnancements à priorités, et on considère que le système est désormais constitué d'une sous-partie temps-réel strict garantie hors ligne, comprenant uniquement des tâches sporadiques et/ou périodiques ; et d'une sous-partie constituée de tâches apériodiques (*sous-partie dynamique* dans la suite). L'objectif est de continuer de garantir la faisabilité des tâches temps-réel strict, tout en obtenant un temps de réponse le plus petit possible pour les tâches apériodiques.

En section 3.4.1, nous étudions le cas où les tâches apériodiques sont sans contrainte temps-réel, puis le cas où de telles contraintes existent (section 3.4.2). Nous terminons

en donnant des améliorations aux mécanismes pour profiter de la terminaison plus tôt des tâches temps-réel strict (section 3.4.3).

3.4.1 Tâches apériodiques sans contrainte de temps-réel

3.4.1.1 Approches par tâches *serveurs*

La méthode la plus simple pour prendre en compte les tâches apériodiques sans contrainte de temps, est de les ordonnancer lorsqu'aucune tâche périodique/sporadique temps-réel strict n'est prête. Avec cette méthode (*ordonnancement en temps libre* ou *background scheduling* en anglais), les conditions de faisabilité et l'algorithme ordonnancement ne sont pas modifiés, mais le temps de réponse des tâches apériodiques qui découle est long.

Afin d'améliorer le temps de réponse des tâches apériodiques dans le contexte d'un ordonnancement à priorités, l'idée est de les faire prendre en charge par une tâche du système qui possède une réelle priorité (statique ou dynamique), sans toutefois remettre en cause l'ordonnancement des tâches périodiques et sporadiques temps-réel strict : on parle de tâches *serveur*. Même si nous nous intéressons ici aux ordonnancements à priorités, une technique similaire existe pour les ordonnancements à plans [Foh94, IF99].

Priorité statique

En ordonnancement à priorité statique, les premières approches [But97] considèrent une tâche serveur périodique de *capacité* fixée re-remplie à chaque début de période du serveur, et qui décroît en fonction du temps (méthode à *serveur périodique* ou *polling server* en anglais), ou en fonction de l'utilisation par les tâches apériodiques [SLS95] (DS, pour *deferrable server*). Dans le premier cas, l'introduction du serveur ne modifie pas les conditions de faisabilité (le serveur est une tâche périodique classique, ordonnancé même si il n'a aucune tâche apériodique à exécuter). En contrepartie, les tâches apériodiques ne sont pas prises en compte quand elles arrivent lorsque le serveur a été complètement ordonnancé dans la période courante. Dans le deuxième cas, les tâches apériodiques y gagnent en réactivité puisqu'elles sont prises en compte à tout moment dans la période courante : on parle de *serveur avec préservation de la bande passante* (*bandwidth preserving* en anglais). Mais l'introduction du serveur perturbe l'ordonnancement des autres tâches temps-réel (le serveur n'est pas ordonnancé pendant la période courante jusqu'à ce qu'il ait une tâche apériodique à exécuter), ce qui modifie les conditions de faisabilité (l'utilisation maximale garantie, analogue à la condition 3.3, est plus faible).

Une variante de DS, plus simple et dont les conditions de faisabilité sont analogues au cas du *polling server*, est la méthode *serveur sporadique* [Spr90] où le serveur est une tâche sporadique dans laquelle la seule contrainte est que les re-remplissages de la capacité du serveur sont séparés par un délai d'inter-arrivée du serveur. En contrepartie, le temps de réponse des tâches apériodiques est plus compliqué à déterminer. D'autres variantes plus complexes existent, comme la méthode à *échange de priorité* [But97] (*priority exchange* en anglais), qui visent à accumuler de la capacité d'exécution tant

que le serveur n'a pas de tâche apériodique à exécuter. La condition de faisabilité est moins défavorable que DS (l'utilisation maximale garantie est plus élevée), mais les tâches apériodiques ont un temps de réponse plus élevée que DS.

Priorité dynamique

En ordonnancement à priorité dynamique, une première technique est de déclarer une tâche serveur, de capacité et de période (ou délai d'inter-arrivée, définissant l'échéance relative) données. EDF peut être utilisé pour ordonnancer les serveurs. On peut utiliser les méthodes à échange de priorité ou de serveur sporadique indiquées ci-dessus, pour définir le comportement du serveur [But97]. Dans les deux cas, la condition d'ordonnancement revient à considérer le serveur comme une tâche temps-réel quelconque, la réactivité des apériodiques étant d'autant plus faible que la période du serveur est grande.

Il existe d'autres méthodes qui évitent d'avoir à attendre le début de la période suivante avant de pouvoir exécuter la tâche apériodique, de façon à réduire leur temps de réponse. Le principe est de réserver une portion de la capacité du processeur à une tâche serveur (sa capacité), et de modifier en-ligne l'échéance relative du serveur suivant la présence ou l'absence de tâches apériodiques à exécuter, tout en garantissant que les tâches temps-réel ne sont pas perturbées. C'est le fonctionnement des *serveurs à bande passante donnée* : TBS [SB94, CLB99, LB00a, SBS95] pour *Total Bandwidth Server* en anglais (pour des requêtes apériodique dont on connaît le temps pire-cas), ou CBS [AB98a, ALB99, JG01, LB01] pour *Constant Bandwidth Server* en anglais (pour des requêtes apériodiques quelconques) par exemple. La condition de faisabilité revient toujours à considérer le serveur comme une tâche temps-réel quelconque du système.

3.4.1.2 Approches par réquisition de temps creux

Priorité statique

Il existe une méthode très consommatrice en mémoire, qui minimise le temps de réponse de la première tâche apériodique en tête de la file des apériodiques sur le serveur [TLS95] : l'*ordonnancement par réquisition de la laxité* [LT94] (*slack stealing* en anglais). Il s'agit de maintenir en mémoire un calendrier (la *fonction de laxité*), établi hors-ligne, et qui couvre une hyperpériode du système. À chaque instant, le calendrier indique la capacité d'exécution que le serveur des tâches apériodiques est autorisé à prendre au détriment des tâches temps-réel strict, tout en garantissant que les échéances de ces dernières restent respectées. Les travaux [DTB93, Dav93] présentent une version complète mais complexe de cette technique, qui calcule la fonction de laxité en-ligne (pas de calendrier), ainsi que des versions approchées mais moins complexes.

Enfin, il existe une méthode aux performances (réactivité des tâches apériodiques) comparables, dite *à double priorité* (ou *dual priority*) [Dav94, GNM99], mais moins coûteuse en mémoire et en temps de calcul tant hors-ligne que en-ligne. Elle consiste à retarder au maximum les tâches temps-réel strict d'un retard calculé hors-ligne par analyse de temps de réponse (le *retard de promotion* ou *promotion time* en anglais)

si il y a des tâches apériodiques présente. Pour cela, pendant la durée de ce retard de promotion, les tâches temps-réel strict ont une priorité plus faible que les tâches apériodiques qui peuvent être présentes, ce qui assure une bonne réactivité des tâches apériodiques. Afin de continuer de garantir les tâches temps-réel strict, une fois que le retard de promotion d'une tâches temps-réel strict est écoulé, celle-ci retrouve sa priorité nominale, supérieure à la priorité de toutes les tâches apériodiques éventuellement présentes.

Priorité dynamique

Il existe également une technique similaire à l'ordonnancement par réquisition de la laxité pour EDF [TLSH94].

3.4.2 Activations en-ligne de tâches avec contraintes de temps-réel

Lorsque les tâches qui se présentent dynamiquement au cours de la vie du système ont des contraintes temporelles (traitement exceptionnel avec échéance stricte par exemple), il est impossible de garantir l'ordonnançabilité de l'intégralité du système hors-ligne (cela reviendrait à pouvoir prévoir l'avenir). En pratique, le problème se découpe en deux sous-problèmes reliés [MM97] : *1/* l'*acceptation* de la tâche, *2/* l'ordonnancement proprement dit. La phase d'acceptation a pour rôle de vérifier que si la tâche peut être ordonnancée, alors *i)* ses propres contraintes temporelles sont respectées en présence des autres tâches déjà présentes, et *ii)* il est possible d'ordonnancer la tâche sans remettre en cause les contraintes temporelles des autres tâches temps-réel (voir ci-dessous). Cette phase est dépendante de l'ordonnancement. Si ce *test d'acceptation* échoue, la tâche est *refusée*.

Tâches périodiques et sporadiques

Si la tâche à accepter est périodique ou sporadique, l'acceptation consiste en la vérification des conditions d'ordonnançabilité données en 3.3.3, ou d'une variante (comme [BS93] pour EDF, ou [McE94, RJMO98] pour l'ordonnancement à priorité statique par exemple).

Tâches apériodiques

Si la tâche à accepter est apériodique, la technique est de soumettre individuellement chaque travail au test d'acceptation, et d'ordonnancer chaque travail suivant une technique telle que celles décrites en 3.4.1. Le test d'acceptation repose sur le calcul du temps de réponse de la tâche apériodique en fonction de toutes les autres tâches déjà présentes dans le système (dépend de l'algorithme d'ordonnancement), et consiste en la vérification que ce temps de réponse est compatible avec les contraintes temporelles. Une partie de ces travaux [CLB99, LB00a, DTB93] s'intéressent en plus à la prise en compte du partage de ressources entre les tâches temps-réel strict, et celles dynamiquement activées.

3.4.3 Récupération de ressources pour favoriser l'ordonnancement de tâches activées dynamiquement

Lorsque l'ordonnancement de tâches temps-réel activées dynamiquement dépend d'un test d'acceptation, il est important de pouvoir compter sur le maximum de ressources disponibles, en particulier en ce qui concerne les ressources processeur. Ceci suppose d'être capable d'évaluer précisément les ressources disponibles et celles nécessaires : si l'on se limite à ne prendre en compte que les pires comportements des tâches en cours (temps d'exécution pire cas, fréquence d'activation maximale pour les tâches sporadiques), alors des tâches temps-réel dynamiquement activées risquent d'être refusées à tort. La *récupération de ressources* est un mécanisme complémentaire à l'acceptation dynamique en-ligne vu précédemment, qui permet de profiter des ressources réservées pour d'autres tâches, mais qui sont finalement non utilisées (par exemple : abandon de la tâche de sauvegarde quand la tâche primaire d'un couple primaire/sauvegarde termine correctement), afin de limiter le pessimisme introduit par la réservation de ressource processeur qui avait été faite sur la base d'un comportement pire-cas.

Il existe des méthodes qui réactualisent les ressources disponibles à la fin de chaque tâche, pour récupérer celles réservées mais non utilisées [SBS95, LB00a, Dav93]. Il existe d'autres méthodes plus précises, qui mesurent l'accélération par rapport au temps d'exécution pire cas, au cours de l'exécution de la tâche [HS90, MZ93, DTB93, GAGB01] : il s'agit de découper la tâche en sections (*milestones* en anglais), dont on connaît le temps pire cas, et encadrées par l'émission de signaux de repérage envoyés au système d'exploitation par exemple.

Ces techniques sont en général accompagnées d'un mécanisme de *repêchage* des tâches refusées (ou *second chance* en anglais) : lorsque la récupération des ressources libère suffisamment de ressources, il peut être possible d'ordonnancer une tache dynamiquement activées qui avait été précédemment refusée et mise dans la file de repêchage (*reject queue* en anglais).

3.4.4 Problèmes reliés

La contention sur la ressource processeur parmi les tâches temps-réel qui arrivent dynamiquement, pose le problème de l'arbitrage entre les tâches à accepter, refuser, ou *abandonner* lorsque le système entre en situation de *surcharge* (ressource processeur insuffisante). Nous verrons en 3.5 quels sont les problèmes soulevés, et comment ils peuvent être traités.

3.5 Gestion de la surcharge

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*. Le phénomène de surcharge peut être lié à un fonctionnement anormal du système, résultat d'un comportement non prévu de l'environnement (activation de tâches sporadiques sans respect du délai minimal d'inter-

arrivée par exemple), ou du système (temps d'exécution réels supérieurs aux temps pire-cas spécifiés, surcoûts du système d'exploitation ou du matériel sous-estimés ou mésestimés). Le phénomène peut aussi être le résultat d'une décision de réalisation, comme la prise en compte de tâches temps-réel apériodiques par exemple, ou la volonté de considérer des temps d'exécution moyens plutôt que des temps pire-cas dans l'ordonnancement (afin de limiter le surdimensionnement du système, quand par exemple le comportement au pire-cas n'est pas *raisonnable*, comme pour un mécanisme d'allocation mémoire par exemple).

Si aucune précaution n'est prise, la conséquence d'une surcharge est que certaines tâches ne respectent pas leur échéance, et le système peut s'écrouler : les tâches qui manquent leur échéance retardent les autres qui à leur tour manquent leur échéance. C'est l'effet *domino* [SSDB94] particulièrement sensible avec un ordonnancement de type EDF, puisque celui-ci fonde ses décisions d'ordonnancement sur la proximité de l'échéance de la tâche, donnant ainsi plus grande priorité à la tâche qui est en train de manquer son échéance.

En amont de la surcharge, il est possible d'éliminer une partie des risques. Par exemple, il est possible de filtrer les événements générés par l'environnement, pour ne pas qu'ils dépassent une certaine fréquence d'apparition [Sta94]. Mais cette approche n'est pas forcément compatible avec le système considéré.

En présence de surcharge, la solution pour contrôler le comportement du système est de borner ou de diminuer les besoins en ressources. Cela suppose :

- Qu'on est capable de détecter en-ligne le commencement de la surcharge. Ce problème est NP-complet [BHR93] dans le cas général. Dans certains cas cependant, il est possible de profiter de bonnes propriétés des algorithmes d'ordonnancement qui permettent de détecter le début de la saturation du processeur [KS93]. Sinon, on fait appel à des approximations qui consistent à mesurer le taux d'occupation du processeur, ou *charge de travail*, afin de *prévoir* la surcharge (peut-être à tort) [MS95, BBL01]. La formalisation puis la mesure de la charge sont eux-même des problèmes en tant que tels, puisqu'il s'agit de tenir compte à la fois des ressources processeur demandées *globalement*, et des contraintes temporelles qui sont *locales* à chaque tâche, et qui participent de la surcharge (pas de surcharge en l'absence de contrainte de temps). Par convention, une charge de valeur 1 marque la frontière entre absence et présence de surcharge.
- Qu'on accepte le fait que le système puisse terminer d'autorité une tâche si nécessaire.

En aval, une fois la surcharge détectée ou anticipée, il y a plusieurs méthodes pour diminuer le besoin en ressource processeur : soit certaines tâches sont volontairement abandonnées, *i.e.* terminées d'autorité (décisions microscopiques) ; soit la structure globale du système est modifiée, *i.e.* le système est *reconfiguré* (décisions macroscopiques). Nous nous intéressons maintenant à ces deux voies d'étude (sections 3.5.1 et 3.5.2 respectivement) dans le cas d'ordonnanceurs pour le temps-réel strict. En section 3.5.3, nous indiquons des modèles de systèmes qui tolèrent des dépassements d'hypothèses dans certaines limites.

3.5.1 Ordonnancement sans reconfiguration

En général, on découpe le système en deux sous-parties : la sous-partie critique définie et garantie hors-ligne, et la sous-partie non critique. La sous-partie critique n'est jamais remise en cause, seule la sous-partie non critique présente les problèmes de surcharge. Nous nous concentrons ici sur cette sous-partie non critique.

On distingue habituellement plusieurs classes d'ordonnanceurs de cette catégorie : les ordonnanceurs *au mieux* (*best effort* en anglais) sans test d'acceptation dans 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*, *i.e.* abandon (dits aussi *robustes*).

3.5.1.1 Notion de valeur

Que ce soit pour évaluer la qualité d'un ordonnanceur en présence de surcharge, pour donner des garanties numériques sur le comportement du système (tous ordonnanceurs), ou pour arbitrer entre les tâches à accepter (ordonnanceurs à garantie) ou abandonner (ordonnanceurs robustes), il est nécessaire d'introduire les notions d'*importance* des tâches représentées numériquement par des *fonctions de valeur*, et de *valeur globale* dégagée par le système, qui définit la *qualité du service* effectivement fourni par le système.

Les fonctions de valeur associées aux tâches peuvent être par exemple constantes, ou fonction du temps d'exécution, ou fonction de la date courante relativement à la date de démarrage, ou fonction de la décision qui doit être prise (acceptation, refus ou abandon ; comme dans [MB97, Del96]). La valeur globale dégagée par le système peut à son tour être la somme des valeurs dégagées par chacune des tâches, ou une fonction plus complexe [BPB+00].

Nous ne considérons pas ici le problème décisionnel lié la définition des fonctions de valeur, qui peut lui-même être un problème complexe [BPB+00].

3.5.1.2 Limites théoriques

Les travaux de Sanjoy Baruah montrent que pour des fonctions de valeur simples, il existe une limite sur la valeur globale maximale garantie qu'on peut atteindre avec un algorithme en-ligne, par rapport à ce qu'il est possible d'obtenir avec un algorithme d'ordonnancement optimal qui disposerait de toutes les informations sur les activations de tâches à venir, dit *ordonnanceur clairvoyant* (comme par exemple [AB98b]). Le quotient entre les deux valeurs est le *facteur de compétitivité*.

Dans le cas simple où la fonction de valeur pour une tâche est : 1 quand la tâche termine dans les temps, et 0 si elle dépasse son échéance où si elle est refusée ou abandonnée, les travaux [BHS01] montrent que dans le cas général, par rapport à un algorithme clairvoyant, les algorithmes d'ordonnancement en-ligne peuvent dégager

une valeur (appelée dans ce cas le *taux de réussite* ou *completion count* ou *hit ratio*) arbitrairement plus petite (*i.e.* le facteur de compétitivité vaut 0).

Des travaux analogues [BKM+91], mais qui s'intéressent à la fonction de valeur qui vaut le temps d'exécution effectif en cas de terminaison correcte, ou 0 sinon (dépassement d'échéance, refus ou abandon), montrent que, par rapport à un algorithme clairvoyant, la valeur dégagée (appelée dans ce cas l'*utilisation!effective du processeur* ou EPU) maximale garantie par un algorithme en-ligne ne peut pas être plus du quart de la valeur dégagée par un algorithme clairvoyant : le facteur de compétitivité est 1/4, ce facteur étant d'autant plus élevé que la surcharge est faible (0.4 quand la charge tend vers 1^+, et bien sûr 1 quand la charge est inférieure à 1). Ces travaux traitent également du cas plus général où la valeur dégagée par l'exécution correcte d'une tâche est fonction du temps d'exécution pondéré par un facteur d'importance de la tâche.

Les travaux [BH97] étendent cette étude en s'intéressant à l'évolution du facteur de compétitivité en fonction de du quotient $\frac{\text{echeance}}{\text{WCET}}$ (qui figure la laxité) minimal parmi toutes les tâches du système : comme on s'y attend, le facteur de compétitivité est d'autant plus élevé (tend vers 1) que ce ratio est élevé.

3.5.1.3 Ordonnanceurs au mieux

Avec ces ordonnanceurs, une tâche est ordonnancée jusqu'à son terme, ou jusqu'à dépassement de son échéance, de manière à borner le besoin en ressources processeur. Cette stratégie peut être associée à un ordonnancement à priorités classique, ou à priorités liées à une fonction de valeur.

Dans le paragraphe précédent, il était question de garanties strictes sur la valeur qu'il est possible d'obtenir. Avec ce type d'ordonnancement, beaucoup de travaux s'intéressent aussi à la valeur *moyenne* dégagée par le système (par simulation ou analyse probabiliste) avec un ordonnancement donné (par exemple RM, DM et EDF dans [Gar99], RM pour l'approche *Statistical Rate Monotonic Scheduling* [AB98d, AB98c], ou des ordonnanceurs à priorité dynamique [AB99, BSS95]), étant données plusieurs lois statistiques sur les durées d'exécution et les dates d'activation des tâches, et plusieurs fonctions de valeur.

Ce type de travail est une transposition des conditions de faisabilité classiques au domaine des probabilités : il permet par exemple d'affirmer qu'un système de tâches donné dégagera une valeur moyenne donnée. Ceci permet d'avoir une idée globale sur la qualité du service *moyen* fourni, mais sans disposer de garantie stricte sur la qualité *minimale* du service fourni. Ce qui rend ces résultats adaptés en priorité au monde du temps-réel souple.

3.5.1.4 Ordonnanceurs à garantie et ordonnanceurs robustes

En général, les *ordonnanceurs à garantie* reposent directement sur le test d'acceptation (tels que ceux évoqués en 3.4) : si celui-ci résussit, la tâche est définitivement intégrée dans le système. C'est le fonctionnement de GED (pour *Guaranteed Earliest Deadline*) [BS93], une variante de EDF avec test d'acceptation. On pourrait cepen-

dant imaginer des ordonnanceurs qui refusent une tâche alors que le test d'acceptation réussit, par anticipation de l'activation prochaine d'une tâche (sporadique par exemple) plus importante.

Les *ordonnanceurs robustes* sont souvent des ordonnanceurs à priorité classiques, avec test d'acceptation, et dans lesquels la politique de rejet n'est activée qu'en cas de refus d'une tâche : quand le test d'acceptation échoue, l'ordonnanceur tente d'abandonner une ou plusieurs tâches de moindre importance. C'est le fonctionnement de RED (pour *Robust Earliest Deadline*) [BS93], une variante robuste de GED.

D'autres ordonnanceurs robustes ne reposent pas sur un test d'acceptation, mais sur un test de démarrage, comme D^{over} [KS93] qui est issu de EDF. D^{over} repose sur une propriété fondamentale de EDF qui lui permet d'identifier la saturation transitoire du processeur (*i.e.* un risque de surcharge) au démarrage d'une tâche, auquel cas le démarrage est conditionné par un test qui optimise la valeur dégagée à hauteur de la limite théorique donnée dans [BKM+91] et évoquée précédemment.

3.5.2 Ordonnancement avec reconfiguration

Les ordonnanceurs de ce type sont capables de passer d'une configuration du système à une autre. Une configuration peut être un ensemble de tâches défini hors-ligne, ou défini en-ligne : le système est constitué d'une série de *services* ; un service étant une fonction particulière du système ou un groupement de tâches établi par un algorithme spécialisé [IMR96]. Chaque service peut être rendu avec une plus ou moins grande qualité, ou différentes contraintes en ressources, en précédences, en temps, par des ensembles de tâches différents ayant chacun leur coût d'exécution.

Ce type d'ordonnanceur repose sur deux mécanismes :

- un mécanisme de suivi de l'état courant du système, avec détermination de la configuration la mieux adaptée à la charge du processeur courante, ou à ses évolutions.
- un mécanisme de changement de configuration.

Ainsi, dans l'approche FERTS par exemple [BSS94, GSSR97], l'application est composée d'entités (*i.e.* FERTs, ou services), et chaque entité peut correspondre à plusieurs séries (compositions) de modules (*i.e.* tâches) : les *stratégies* (*i.e.* configuration) de chaque entité. Le système ne prend pas les décisions de changement de configuration à l'échelle de la tâche, mais à l'échelle de l'entité. Lorsqu'une entité est activée, en fonction de l'état courant du système, la stratégie la plus adaptée pour l'entité est choisie.

Les travaux [Del96] présentent une approche similaire : la politique d'abandon de tâches en cas de refus d'une tâche est définie par le concepteur sous la forme d'un composant logiciel dédié : le *régisseur*. Mais en plus, pour les tâches périodiques, plusieurs configurations globales du système (ensemble de tâches), ou *phases*, sont définies hors-ligne. Le concepteur fournit un automate pour passer d'une configuration à l'autre, et c'est le régisseur qui décide de changer l'état de l'automate (*i.e.* des tâches à ordonnancer) en fonction de la charge courante. Malheureusement, la notion de phase pose deux problèmes :

- la transition d'une phase à l'autre ne doit pas laisser le système dans un état incohérent. En particulier, pour tous les services du système qui sont synchronisés par ressources ou qui ont des contraintes de précédence, il est important que la transition se fasse de façon correcte (libération des ressources, validation des contraintes de précédence sur la phase à terminer).
- la transition d'une phase à l'autre doit être justifiée. En effet, elle n'est pas immédiate. Il faut donc d'une part prendre en compte les surcoûts et les délais induits par ces mécanismes, et d'autre part être assuré qu'une fois la transition effectuée, l'état courant résultant est compatible avec la phase introduite (au moins en ce qui concerne les contraintes temporelles). Sinon il y a instabilité de la configuration : au pire, les tâches de la nouvelle phase dépassent leurs contraintes temporelles ; et au mieux le système retourne à la phase précédente, induisant des oscillations entre phases.

  Une solution pour éviter ce problème est d'introduire de la marge dans les décisions de changement de phase (phénomène d'hystérésis) : il s'agit de définir le comportement temporel d'un changement de phase, et d'assurer hors-ligne, que pendant cet intervalle et étant donné les hypothèses faites sur le comportement de l'environnement, les paramètres de décision pour le passage d'une phase à l'autre n'évolueront pas dans le sens de l'établissement d'une autre phase, ou du retour à la précédente.

Des travaux avec le système distribué ordonnancé par plans statiques Maruti [SdSA95] résolvent le problème de la cohérence en établissant à l'avance une série de points où les changements de reconfiguration peuvent se faire. D'autres travaux avec le système distribué ordonnancé par plans statiques MARS [KNH+97] résolvent également ce problème, en reposant pour cela à la fois sur l'existence d'une phase de transition entre 2 modes adjacents (le mode précédent, et le mode destination), et sur le fait que l'ordonnancement est commandé par le temps, afin d'assurer la synchronisation des changements de mode. Cependant, il reste toujours le danger de l'instabilité au changement de phase, que le concepteur doit continuer de traiter.

3.5.3 Extensions au modèle de tâche

Il existe des modèles qui intègrent directement le fait que des tâches puissent être totalement ou en partie abandonnées, et ceci avec garantie en-ligne, ou hors-ligne dès la phase de validation par analyse de faisabilité.

Ainsi, il existe des modèles pour lesquels la spécification doit préciser quels travaux d'une tâche peuvent être supprimés :

- Le modèle à perte sporadique (ou *skip-over* en anglais) [KS95, CB97] : un travail est autorisé à être abandonné au maximum toutes les x activations (x spécifié). Déterminer la faisabilité d'un ensemble de tâches de ce type est NP-difficile, car il consiste en le choix des travaux à refuser préventivement à la surcharge, ce qui est analogue au problème de détection de surcharge. Par contre, des heuristiques simples qui étendent EDF et RM de façon pseudo-polynomiale existent.

– Le modèle avec pertes contraintes par fenêtre (ou *window-constrained* en anglais) [MW01] considère des tâches périodiques telles que, pour une fenêtre répétée périodiquement de k périodes, m travaux ($m \leq k$) doivent respecter leur échéance. Dans le cas général, le problème de faisabilité est également NP-difficile. Cependant, le cas particulier où toutes les tâches ont même WCET possède à la fois une condition de faisabilité polynomiale, et un algorithme d'ordonnancement approprié.
– Le modèle avec fréquence de pertes contrainte (ou *weakly-hard* en anglais) [Nic98, BBL01] est assez proche du modèle précédent, à la différence qu'il considère des fenêtres glissantes : n travaux (éventuellement successifs) parmi m successifs respectent (ou ne respectent pas) leur échéance. En ce sens, il s'agit d'une généralisation du modèle à perte sporadique. Un test de faisabilité non polynomial existe, qui est adapté à l'ordonnancement à priorité statique.

Il existe des modèles où seule une partie de la tâche est garantie hors-ligne, l'exécution du reste dépendant de la charge du système :

– Le modèle de *calcul imprécis* (ou *imprecise computation* en anglais) [LLS+91] considère des tâches qui sont en deux parties : une partie dont l'exécution correcte est garantie hors-ligne, et une partie facultative dont l'exécution peut être abandonnée en cours d'exécution si besoin : par définition, l'*erreur* d'exécution est d'autant plus grande que la partie facultative est abandonnée tôt, et est définie par la *fonction d'erreur*. Avec ce modèle, la problématique est de minimiser l'erreur totale [LLS+91], ou l'erreur maximale [SL95] avec des algorithmes dérivés de RM ou EDF. Cependant, Baruah [BH98] montre que, dans le cas général, la valeur qu'il est possible de dégager avec un tel modèle (les parties garantie et facultative ont chacune une fonction de valeur constante) peut être arbitrairement plus faible que la valeur maximale théorique (*i.e.* le facteur de compétitivité vaut 0).
– Le modèle à transformation de tâche (ou *transform-task*) [TDS+95], considèrent des tâches dont une partie est garantie hors-ligne, et le reste est ordonnancé au mieux dans un serveur pour tâche apériodique. La problématique est de déterminer hors-ligne [TDS+95], ou de mesurer par simulation [GL99, Gar99], la probabilité de dépassement d'échéance en fonction de la distribution des durées d'exécution de la partie ordonnancée par serveur, suivant le type de serveur et la politique de service des requêtes dans le serveur (FIFO, DM par exemple).
– Le modèle à exception (*taskpair scheduling* ou TPS) [Str95, Net97], qui est la démarche inverse du modèle à calcul imprécis : une tâche (dite *tâche principale*) est associée à une tâche fantôme (dite *exception*). L'exception n'est exécutée que si la tâche principale s'approche *trop* de son échéance : la tâche principale est abandonnée au profit de l'exception quand la date de démarrage au plus tard de l'exception est atteinte. Ainsi, l'exception est toujours garantie (par test d'acceptation ou hors-ligne), par contre la tâche principale ne l'est pas. Ce modèle permet d'isoler les fautes temporelles et de garantir un fonctionnement minimal. Ce modèle est également compatible avec une connaissance non sûre des temps d'exécution

de la tâche principale (par exemple statistique [GS96, NGM01, NG97b], ou par instrumentation et extrapolation [SG97a, NGS97, NG97a, NGM98, SG97b]).

3.6 Extensions du modèle de système

Nous avons présenté les méthodes d'ordonnancement classiques en temps-réel. Il existe cependant d'autres approches qui reposent sur d'autres modèles du système concernant l'ordonnancement. Les deux sections qui suivent abordent deux modèles de systèmes différents : systèmes fondés sur un ordonnancement fluide (section 3.6.1), et systèmes fondés sur une hiérarchie d'ordonnanceurs (section 3.6.2).

3.6.1 Ordonnancement fluide

En matière d'ordonnancement dynamique, dans les paragraphes précédents, le système d'exploitation fait appel aux décisions d'ordonnancement uniquement lorsque la structure du système est modifiée : une tâche est activée, dépasse une contrainte temporelle, demande/relâche une ressource système, ou se termine. Certains problèmes d'ordonnancement sont rendus complexes en partie à cause des périodes incompressibles entre deux décisions d'ordonnancement (par exemple le calcul du temps de réponse), puisqu'il s'agit alors de faire des calculs, des vérifications ou des optimisations non linéaires.

Le modèle d'ordonnancement fluide, par Pfair [SAWJ+96, JG01] (*proportional fair*), ou PSA [ALB99] (*proportional share allocation*), s'inspire des techniques de partage de bande passante en réseau ou en multimédia (comme dans le système Nemesis [oCCL00] par exemple, qui procède par réservation), et a pour but de se rapprocher le plus possible d'un système linéaire, dans lequel le rythme de progression des tâches serait régulier. Ce modèle revient à considérer qu'à tout moment les tâches ont une portion donnée et connue du processeur qui leur est réservée (*i.e.* un processeur virtuel de moindre capacité donnée connue) : la fraction de la *bande passante* de traitement du processeur. Et une fois que le rythme de progression est connu, il est possible d'assurer que les contraintes temporelles sont respectées.

Le modèle fluide parfait imposerait qu'à tout instant le processeur exécute une partie de chaque tâche, proportionnelle à la bande passante du processeur qui lui est allouée. Ce principe théorique n'est pas applicable : en pratique, l'ordonnanceur est appelé régulièrement, toutes les q unités de temps (le *quantum* d'ordonnancement). Cependant, en contrepartie à la discrétisation introduite par les quanta de q unités de temps, le rythme de progression défini par l'ordonnanceur n'est pas exactement constant, mais borné de façon sûre et connue par l'*ecart temporel@écart temporel* (*time lag* en anglais), qui correspond à l'écart maximal entre les progressions obtenues par exécution réelle et suivant le modèle théorique fluide parfait.

Dans les méthodes les plus simples répondant à ces contraintes, le concepteur associe un poids absolu ω_i à chaque tâche, et l'ordonnanceur garantit que la tâche aura une fraction $\frac{\omega_i}{\sum_j \omega_j}$ de la bande passante du processeur qui lui est allouée. Il existe par exemple une adaptation de EDF, dite EEVDF (pour *Elligible Earliest Virtual Deadline First*), qui borne l'écart temporel à q unités de temps, en travaillant dans une

échelle de temps virtuelle : dans cette échelle de temps, 1 unité de temps correspond à l'exécution de toutes les tâches de sorte que celles-ci progressent individuellement de ω_i unités de traitement processeur. Cet algorithme est accompagné d'une condition de faisabilité analogue à $U \leq 1$, et peut être adapté pour tenir compte des terminaisons plus tôt des tâches, ou pour l'ordonnancement de tâches activées dynamiquement (*via* test d'acceptation en-ligne simple). Il existe également d'autres algorithmes tels que WFQ (*Weighted Fair Queueing*) ou SFQ (*Start Fair Queueing*) plus complexes ou avec un écart temporel plus élevé.

Avec ces algorithmes, le rythme de progression de chaque tâche varie en fonction de $\sum_i \omega_i$, donc en fonction de la structure du système (tâches activées non terminées). Il existe des variantes qui fonctionnent par allocation, dès la phase d'acceptation (ou hors-ligne), d'une fraction fixée de la bande passante de traitement du processeur, indépendamment du nombre et du poids des autres tâches : il s'agit de recalculer les poids absolus des tâches à chaque changement de structure du système (activation ou fin de tâche) de manière à ce que pour chaque tâche, $\frac{\omega_i}{\sum_j \omega_j}$ corresponde à la bande passante allouée.

L'inconvénient du modèle d'ordonnancement fluide est que les surcoûts système sont plus élevés : liés d'une part aux préemptions tous les q quanta de temps (par interruption), et d'autre part aux décisions d'ordonnancement qui doivent être prises quoi qu'il arrive. Cependant, en pratique, ces surcoûts demeurent raisonnables parce que *i)* les systèmes d'exploitation gèrent une horloge système, donc on peut profiter de l'interruption d'horloge pour prendre les décisions d'ordonnancement, auquel cas le quanta de temps q correspond à la résolution de l'horloge système (ou à un multiple) ; *ii)* les algorithmes d'ordonnancement pour ce modèle sont peu coûteux (comparables à la complexité des ordonnancements à tourniquet, ou *round-robin* en anglais, des systèmes généralistes).

L'avantage de ce modèle est qu'il permet de résoudre des problèmes d'ordonnancement temps-réel complexes en multiprocesseur, comme nous le verrons en 3.8.2.

3.6.2 Ordonnanceurs multiples et hiérarchiques

Dans les paragraphes précédents, nous considérions une application unique, et le système d'exploitation était chargé d'arbitrer entre les besoins en ressources de toutes les tâches de façon globale.

Quelques travaux s'intéressent cependant aux systèmes à *ordonnanceurs multiples* (ou également *à priorités en couches*), c'est à dire au cas où plusieurs ordonnanceurs doivent cohabiter dans le système. En particulier, les travaux [Mig99] et [Nav99] présentent une technique analytique fondée sur le calcul d'une borne sur le temps de réponse des tâches, pour établir l'ordonnançabilité lorsque le système est constitué de plusieurs files d'exécution, pour lesquelles la politique d'ordonnancement peut être soit de type FIFO, soit de type tourniquet (c'est le cas de système respectant la norme Posix 1003.1b).

Les travaux sur l'*ordonnancement hiérarchique* s'attachent à rendre le modèle de système plus modulaire, d'apparence mieux conçu, de sorte que le système est composé

d'applications disposant chacune d'une fraction des ressources du système, dont la ressource processeur. L'idée est d'*isoler* les besoins et les occupations de ressources entre applications, y compris au niveau de la gestion des contraintes de temps et des occupations processeur [Sta93a], de la même manière que l'est la mémoire par exemple.

Pour cela, l'approche est de partager le travail d'ordonnancement entre plusieurs *niveaux hiérarchiques*. Dans le modèle le plus simple [WL99a, RH01, CJ98], l'ordonnancement des tâches est délégué à une composante spécialisée de l'application parente ; et le système s'occupe de l'ordonnancement des ordonnanceurs des applications. Ce modèle d'ordonnancement hiérarchique peut être étendu à plus de deux niveaux.

Dans un tel modèle, l'application devient un *composant* du système chargée elle-même de l'ordonnancement de ses tâches, avec les avantages d'abstraction et de modularité que cela apporte (aspect boîte noire), mais aussi avec toute la difficulté de prouver la composition de tels composants vis à vis des contraintes temporelles [RS01].

Pour supporter ce paradigme, plusieurs notions sont proposées, qui peuvent être composées dans le système :

- Négociation par propagation [RSH00, LMA88] : où le système d'exploitation continue de superviser globalement l'ensemble des ressources et des consommateurs de ressources. Le principe est que chaque composant (tâche, application) négocie ses ressources (telles que de la capacité processeur) avec le composant de niveau hiérarchique supérieur, et les demandes remontent jusqu'au système d'exploitation. Le principe fondamental de cette approche est que chaque composant doit être capable de rassembler le maximum d'informations concernant le comportement des composants de niveaux inférieurs, ce qui suggère la présence de propriétés de réflexivité [RSH00].
- Virtualisation [MF02, LB00b, GAGB01] : à chaque application est associée une partie de la capacité de traitement du processeur sous la forme d'une tâche serveur ou par ordonnancement fluide, définissant ainsi une forme de processeur fictif dédié à l'application. L'application est alors en charge de gérer la ressource processeur fictive qui lui est assignée (on parle de *système ouvert*). Les travaux de [MF02] montrent qu'on peut obtenir des garanties temps-réel strict avec cette approche par virtualisation, en associant à chaque application du système deux paramètres temporels, qui définissent les caractéristiques de la ressource processeur virtuelle utilisée, et qui servent de contraintes pour garantir les échéances dans une composition hiérarchique d'applications, de sous-applications, de sous-sous-applications, etc...

Quelques travaux [BM01, BM02] présentent un prototype d'implantation incluant ces deux notions, pour des mécanismes à deux niveaux hiérarchiques, fondés sur un langage dédié à la définition des ordonnanceurs.

Étant donné la complexité de prouver le comportement temporel correct de tels systèmes, ils trouvent en général bien leur place dans les systèmes temps-réel souple, et peuvent dans ce cas s'accommoder d'une connaissance minimale, incomplète, ou approximative sur les comportements temporels des applications et de leur tâches.

3.7 Relâchement d'hypothèses sur l'environnement et le système

Jusqu'à présent, le système était conçu à des fins de prévention des fautes temporelles : il était admis que toute tâche en cours d'exécution dans le système avait un comportement temporel et des besoins en ressources connus dès la phase de conception, ou au moment de l'acceptation.

Nous allégeons maintenant ces hypothèses, en considérant des systèmes pour lesquels toutes les informations sur le comportement temporel de l'application et de l'environnement ne sont pas disponibles avant l'acceptation des tâches. Cela peut être dû à une très grande complexité de modélisation (plusieurs centaines de tâches interdépendantes par exemple) qui empêche de caractériser leur comportement. Ou à une volonté délibérée de ne pas prendre en compte les comportements pire-cas systématiquement, de manière à rentabiliser au maximum les ressources système.

Face à ces lacunes, il s'agit soit de ramener le comportement réel dans les limites d'un comportement pire-cas identifié (principe de filtrage présenté en 3.5), soit de prévoir l'information manquante.

Nous présentons ici brièvement deux approches dont le rôle est de s'adapter à l'absence, avant exécution, d'informations utiles aux décisions d'ordonnancement. Ces mécanismes trouvent largement leur place en temps-réel souple.

3.7.1 Approche par prévision

Le principe de cette approche est que le système observe son propre comportement selon un certain nombre de métriques, et utilise ses observations pour prétendre prévoir l'évolution du comportement à venir, et ainsi fonder une partie des décisions (d'ordonnancement ici) sur ces prévisions. Les informations manquantes sont donc générées à partir du comportement passé.

En pratique, le fonctionnement consiste en un composant qui définit le comportement à venir d'un paramètre à partir des valeurs brutes mesurées, de leur dérivée et de leur intégrale : on parle de contrôleur PID (pour *proportional integral derivative* en anglais). Ce composant injecte ses résultats en entrée du mécanisme de décision, et l'ensemble forme ainsi une boucle de rétroaction (*feedback control* en anglais). Le choix se porte vers le contrôleur PID parce qu'il s'agit d'un composant dont le comportement (en particulier les conditions de stabilité) a été largement étudié en traitement du signal.

Ainsi, il existe une extension de EDF (FC-EDF et FC-EDF^2 pour *Feedback-Control EDF*) [SLS98, LSA+00] qui utilise une rétroaction PID afin de prévoir la charge à venir. Cette prévision conditionne le choix de la version de chaque tâche à accepter, pour des tâches qui existent en plusieurs versions.

Il existe également des extensions de TPS (présenté en 3.5.3) [GS96, SG97a, NG97a, NGM01] qui permettent d'affiner l'ordonnancement des tâches principales, par prévision des temps d'exécution en fonction des temps d'exécution observés dans le passé (stockés dans une base de données).

3.7.2 Systèmes réflexifs

Le principe de cette approche est symétrique à la précédente : le système dispose de toutes les informations qui lui permettent de reconstruire de façon précise l'information manquante. Une autre caractéristique des systèmes réflexifs est qu'ils sont capables de modifier leur propre structure si ils jugent que l'état courant l'exige. Cette deuxième caractéristique peut consister en une reconfiguration du système (voir en 3.5.2), ou en la sélection de mécanismes d'exécution à la volée (comme une stratégie de tolérance aux fautes par exemple). L'ensemble de ces deux caractéristiques peut être offert par le langage dans certains cas, ou sinon elle doit faire partie du modèle de système.

Grâce à la première propriété, il est par exemple possible de connaître l'ensemble des entités (objets, ressources) qui interviendront pendant l'exécution d'un travail, avant l'activation de celui-ci, et en fonction de l'état courant du système (par exemple si l'activation des travaux d'une tâche est modélisée par un automate). Ensuite, grâce à ces connaissances, il est possible de dérouler les tests d'acceptation et l'ordonnancement [Sta93b].

Ainsi, le système Spring [SRN+98], dont l'ordonnancement est de type plan dynamique, utilise cette approche afin de générer ses plans : la chaîne de compilation hors-ligne extrait le maximum d'informations sur le comportement de chacune des entités. Ces informations permettent également de vérifier automatiquement en-ligne un certain nombre de contraintes de synchronisation. Le même type de fonctionnement est considéré dans [RSYJ97] où il s'agit d'utiliser les informations sur les différentes entités du système, ainsi que leurs différents besoins en ressources, pour pouvoir initier des changements de configuration suivant l'évolution des utilisations de ressources, elles-mêmes conditionnées par le comportement de l'environnement.

3.8 Ordonnancement pour systèmes multiprocesseurs ou distribués

Nous considérons ici des systèmes constitués de plusieurs processeurs qui coopèrent pour exécuter l'application, et présentons très brièvement les problèmes liés à l'ordonnancement dans ce contexte, ainsi que quelques voies pour les résoudre.

Dans ce qui suit, nous disons qu'un système est *multiprocesseur* quand l'ordonnancement qui s'exécute est unique (*i.e.* les files d'attentes de tâches sont partagées entre tous les processeurs), et qu'il est *distribué* quand l'ordonnancement est distribué (*i.e.* chaque processeur est en charge de l'ordonnancement d'une partie donnée des tâches). Les décisions qui mènent à un choix de conception plutôt qu'à l'autre dépendent du degré de couplage entre les processeurs : un système à mémoire physiquement partagée sera plus naturellement régi par un ordonnanceur central unique (modèle multiprocesseur ci-dessus), qu'un système architecturé sans mémoire partagée autour d'un réseau local.

Après avoir présentés quelques difficultés liées à l'ordonnancement de ce type de systèmes (section 3.8.1), nous indiquons très brièvement quelques travaux dans le

domaine de l'ordonnancement multiprocesseur (section 3.8.2), puis distribué (section 3.8.3).

3.8.1 Problématique

Nous l'avons vu en 3.2.2.2, dans le schéma le plus simple (pas de ressource, pas de précédence), aucun ordonnanceur multiprocesseur en-ligne n'est optimal [Mok83] (en particulier aucune extension multiprocesseur de RM ou de EDF). Par contre, suivant les modèles de tâches choisis, il existe des algorithmes d'ordonnancement qui permettent d'ordonnancer les tâches (mais de façon non optimale), et qui sont accompagnés de conditions de faisabilité.

L'ordonnancement en multiprocesseur ou en distribué procède par conséquent au cas par cas, et se heurte à un certain nombre de difficultés dans l'établissement des conditions de faisabilité car, par rapport au cas centralisé, il existe un certain nombre d'"*anomalies*" [Law83, GFB01], comme par exemple :

- La terminaison en avance des tâches peut rendre infaisable un ordonnancement qui était faisable.
- Le pire temps de réponse n'est pas obtenu lorsque toutes les tâches sont démarrées au même instant.

3.8.2 Ordonnancement pour systèmes multiprocesseurs

Ce modèle de système correspond au plus simple envisageable : par rapport à l'ordonnancement centralisé, il s'agit d'ajouter une étape d'*affectation* des tâches aux processeurs, et à gérer leur *migration* vers les autres processeurs (en général à faible surcoût).

Un cas particulier à ce modèle correspond à un domaine de recherche particulièrement actif actuellement : il concerne les *multiprocesseurs uniformes*. Dans ce modèle, si un traitement met c unités de temps à s'exécuter sur un processeur, ce même traitement demandera $s.c$ unités de temps pour s'exécuter sur un autre processeur du système, avec le facteur s (la *vitesse* ou *rapidité*) constant et connu. Les *multiprocesseurs identiques* sont un cas particulier de multiprocesseur uniforme : tous les processeurs ont la même capacité de traitement, *i.e.* $s = 1$ pour tout couple de processeurs du système.

Dans ce cas, il existe une condition nécessaire et suffisante pour l'ordonnançabilité d'un ensemble de tâches périodiques indépendantes [FGB01, GBF02] (algorithme d'ordonnancement inconnu). Ce résultat est utilisé pour dériver une condition suffisante pour l'ordonnancement de tâches périodiques par une variante de EDF sur multiprocesseur. Dans cette variante, si le système comporte m processeurs, à tout instant les travaux prêts à être exécutés et de plus haute priorité sont affectés aux processeurs les plus rapides, jusqu'à concurrence de m travaux, et en assurant qu'un travail plus prioritaire qu'un autre est affecté à un processeur plus rapide. Cette définition de EDF introduit par conséquent beaucoup de migrations (au rythme des activations et terminaisons des tâches), mais aucune alternative ne semble diminuer ce nombre de migrations dans le cas général [GFB01].

D'autres travaux [ABJ01] se sont intéressés à l'aménagement analogue de RM pour le cas particulier du multiprocesseur identique, et proposent une condition suffisante pour l'ordonnançabilité.

Les algorithmes d'ordonnancement fluide ont également été étendus au cas multiprocesseur identique pour des tâches périodiques indépendantes [AS99, BGP95, RM00].

Lorsque des contraintes plus complexes sont à prendre en compte (autres ressources, précédences), peu de résultats existent concernant ces algorithmes simples. En pratique, l'ordonnancement de tels systèmes repose sur les mêmes méthodes que celles utilisées pour les systèmes distribués.

3.8.3 Ordonnancement pour systèmes distribués

Dans ces systèmes, il n'y a pas un unique mécanisme d'ordonnancement qui centralise les informations et les décisions d'ordonnancement, mais plusieurs (1 par processeur, ou 1 par multiprocesseur). Par rapport à l'ordonnancement multiprocesseur, quatre difficultés supplémentaires apparaissent :

- La difficulté de maintenir une vision cohérente de l'état du système : assurer le consensus sur les décisions d'ordonnancement introduirait un fort surcoût, en général pas compatible avec les contraintes temporelles. En pratique, chaque ordonnanceur s'occupe de gérer un sous-ensemble identifié des tâches du système, et en isolement des autres ordonnanceurs.
- Les coûts de migration sont en général élevés, ce qui entraîne que la décision d'affectation est plus déterminante qu'en multiprocesseur, et qu'elle peut difficilement être remise en cause.
- Il est nécessaire de prendre en compte l'ordonnancement de la ressource de communication (réseau) entre les processeurs.
- Pour les systèmes critiques, il est nécessaire de prendre en compte la contrainte de disponibilité et de fiabilité du système et du réseau (très sensible à l'environnement au travers des interférences électromagnétiques par exemple), par des techniques de tolérance aux fautes.

Pour pallier aux complexités théoriques, et aux difficultés de réalisation majeures, en pratique trois voies sont couramment empruntées : l'ordonnancement statique (pour la ressource processeur et/ou réseau), l'ordonnancement avec affectation statique des processeurs, et différentes heuristiques.

3.8.3.1 Ordonnancements statiques

Étant donné la complexité du problème d'ordonnancement, il est intéressant de la reléguer hors-ligne (par exemple, dans [EJ00], ou dans les systèmes Mars [KFG+92, SRG89, Foh94] et Maruti [SdSA95]), ce qui permet de définir et de valider à la fois les allocations des processeurs et du réseau dès la phase de conception.

3.8.3.2 Ordonnancements avec affectation statique des processeurs

Dans ce modèle, les tâches sont regroupées par processeur dès la phase de conception, et un algorithme d'ordonnancement dérivé du cas centralisé est employé sur chaque processeur.

Ainsi, il existe des approches qui reposent sur la définition d'un plan au niveau de l'ordonnancement pour le réseau (on parle de multiplexage temporel, ou TDMA pour *Time Division Multiple Access*). Il s'agit ensuite d'associer à chaque émission ou réception de message, une tâche sporadique dans chacun des systèmes communiquant en point à point, et à intégrer ces tâches dans le modèle de système, en intercalant entre elles les temps de propagation du messages sous la forme de contraintes de précédence, de temps de blocage, et de gigue de démarrage. Lorsqu'il s'agit d'affecter les différentes tâches aux processeurs pour ce modèle de système, les travaux [TBW92a] proposent une solution fondée sur une méthode d'optimisation globale suboptimale (le *recuit simulé*). Les travaux [Nav99] présentent une approche comparable lorsque les messages sont échangés sur un réseau à priorité (bus CAN), et s'intéressent également au taux de mon respect des échéances dû à la perte ou à l'altération de messages.

Il existe d'autres modèles de systèmes beaucoup plus simples, où les échanges de messages entre processeurs sont modélisés par des traitants d'interruption (de plus haute priorité que toutes les tâches) qu'il s'agit de prendre en compte dès la phase de conception [TH99]. Cette phase de validation est beaucoup plus complexe, puisqu'il s'agit d'étudier de manière exhaustive tous les entrelacements possibles de toutes les tâches avec les interruptions possibles, suivant les profils d'émission de messages.

3.8.3.3 Heuristiques

Dans les systèmes distribués plus complexes, qui doivent considérer par exemple à la fois des contraintes de précédence et de ressources en environnement hétérogène et de façon dynamique, il existe des approches suboptimales par heuristiques.

L'algorithme le plus répandu étant l'*algorithme myope* (*myopic algorithm* en anglais) [Man98], utilisé dans le système Spring [SRN+98]. Le système Spring est composé de multiprocesseurs à 4 processeurs à mémoire partagée, dont 1 spécialisé dédié à l'ordonnancement, et les multiprocesseurs sont reliés entre eux par un réseau dédié. L'algorithme myope est chargé d'établir des plans dynamiques pour chaque multiprocesseur, en tenant compte des précédences (éventuellement vis à vis des autres multiprocesseurs du système) et des contraintes temporelles. Il repose sur un algorithme d'allocation sur les processeurs locaux itératif, relié à un test d'acceptation, et dont le nombre d'itérations est borné à la conception.

Cet algorithme a été étendu pour supporter des modèles de tâches particuliers, tels que les FERTS évoqués en 3.5.2, ou l'intégration de tâches temps-réel souple [MMM00b] (par serveur).

3.8.3.4 Considérations de tolérance aux fautes

Une définition anecdotique des systèmes distribués (attribuée à Leslie Lamport) est : "*A distributed system is one that stops you from getting any work done when a machine you've never even heard of crashes*". En effet, dans un système distribué, la probabilité qu'au moins un élément soit défaillant est mécaniquement liée (exponentiellement) au nombre d'éléments dans le système.

En conséquence, pour les systèmes distribués temps-réel critiques, il est nécessaire de prendre en compte les fautes qui peuvent apparaître. La technique est d'utiliser la *redondance* (des traitements ou des messages de communication) temporelle ou spatiale de façon à détecter ou masquer les fautes qui peuvent apparaître.

Mais gérer la redondance de façon correcte impose que les fonctions répliquées soient cohérentes entre elles, ce qui amène à résoudre le problème du consensus, ce qui rajoute des contraintes supplémentaires à la conception du système. Notamment parce qu'une des briques de base est de déterminer, et de façon cohérente, quels nœuds sont corrects ou non, avant de pouvoir établir l'accord sur les décisions entre les nœuds corrects (on parle de *mécanisme de gestion de groupe*). Or ceci, dans le modèle de défaillance le plus simple dit à silence sur défaillance, rajoute la présence d'un délai de détection de défaillance, souvent élevé, à prendre en compte dans la conception du système.

Un autre problème qui se pose en présence de mécanisme de tolérance aux fautes des nœuds est celui du redémarrage des nœuds après défaillance, et de leur réinsertion dans le système. Cette faculté, quand elle est supportée, pose à la fois le problème de la cohérence de la décision de réinsertion vis à vis du mécanisme de gestion de groupe, et également celui de la reprise d'un état correct par le nœud qui se réinsère. Suivant l'implantation du système et le mécanisme de reprise, celui-ci peut entraîner des occupations en ressources conséquentes le temps du transfert ou de l'établissement de l'état de reprise qu'il est nécessaire de prendre en compte en particulier dans l'ordonnancement des messages réseau [Vij98].

Le système MARS [KFG+92] par exemple propose toute une architecture avec redondance du réseau et des traitements, et leur intégration avec les contraintes de temps-réel. La tolérance aux fautes est facilitée de par la conception autour d'un plan d'ordonnancement connu de tous les nœuds, permettant par exemple de détecter la défaillance d'un nœud (absence ou retard d'un message).

De même, Spring [GSSR97, MM97], Hades [CPC+00] ou Dharma [MMM00a] proposent plusieurs solutions pour la gestion de plusieurs types de redondances logicielles en contexte de temps-réel. Par exemple, Hades repose sur la décomposition des traitements du système en unités élémentaires caractérisées hors-ligne par des besoins en ressources, et liées entre elles par des contraintes de précédences éventuellement conditionnelles accompagnées de l'envoi de messages. Ce type d'architecture autorise la prise en compte des traitements répliqués pour établir le consensus même en présence de défaillances, dans les analyses d'ordonnancement.

Conclusion

La plupart des analyses hors-ligne consistent à déterminer la pire configuration possible compte-tenu de l'algorithme d'ordonnancement et des propriétés des tâches, et à vérifier que cette configuration est compatible avec les contraintes temporelles (notamment en comparant les temps de réponse obtenus avec les contraintes d'échéance). Pour plus de souplesse de conception et d'utilisation, la tendance actuelle est d'intégrer ce type de système spécifié, dimensionné et garanti hors-ligne, avec des tests d'acceptation en-ligne de tâches. Ces tests partent de la configuration courante, pour déterminer la pire configuration qui résulterait de l'activation de la nouvelle tâche, et à son tour vérifie que cette configuration est compatible avec les contraintes temporelles.

Dans ces cas, la phase d'analyse d'ordonnancement recourt le plus souvent à des démonstrations fondées sur l'étude du système dans son ensemble. Des travaux récents s'attachent à alléger cette contrainte, c'est à dire qu'ils s'intéressent aux *systèmes ouverts*. Dans ces systèmes, l'objectif est qu'une série d'applications, chacune chargée de gérer ses propres tâches suivant sa propre politique d'ordonnancement, puissent coopérer. Des résultats ont déjà été obtenus (ordonnancement fluide, hiérarchique, ressources virtuelles) allant dans le sens d'une virtualisation des ressources partagées, et une coopération sous forme *boîte noire* des traitements temps-réel effectués par le système.

D'autres axes sont encore largement ouverts, comme l'ordonnancement temps-réel en-ligne pour des systèmes temps-réel distribués ou multiprocesseurs. D'importants résultats d'impossibilité ou de limites théoriques ont été obtenus récemment, et de grands efforts d'intégration avec les problématiques connexes ont été faits (notamment en ce qui concerne la tolérance aux fautes en distribué). Mais l'ordonnancement en-ligne dynamique dans ce contexte reste un domaine à approfondir.

Articles référencés

[AB98a] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, pages 4–13, 1998.

[AB98b] Alia Atlas and Azer Bestavros. An omniscient scheduling oracle for systems with harmonic periods. Technical Report 1998-014, Sept 1998.

[AB98c] Alia K. Atlas and Azer Bestavros. Design and implementation of statistical rate monotonic scheduling in kurt linux. Technical Report 98-013, Boston University, Sept 1998.

[AB98d] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Dec 1998.

[AB99] S. Aldarmi and A. Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, Dec 1999.

[ABJ01] Bjorn Andersson, Sanjoy Baruah, , and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, pages 193–202, Dec 2001.

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sep 1993.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[ALB99] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In *Proceedings of the International Conference on Multimedia Computing and Systems*, volume II, pages 107–111, Jun 1999.

[AP97] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Technical Report PI1103, IRISA, July 1997.

[ARJ95] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Dec 1995.

[AS99] J. Anderson and A. Srinivasan. A new look at pfair priorities. *Real-time Systems Journal*, Oct 1999.

[ATB93] N.A. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proceedings of the Fifth Euromicro Workshop on Real-time Systems*, pages 36–41, Jun 1993.

[Aud91] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, U York, Nov 1991.

[Bak91] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1):67–100, 1991.

[Bar97] Michael Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, Jun 1997.

[Bar98] Sanjoy Baruah. A general model for recurring real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, pages 114–122, 1998.

[Bat98] IJ Bates. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD thesis, University of York, Nov 1998.

[BBL01] G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. In IEEE, editor, *IEEE Transactions on Computers*, number 50(4), pages 308–321, Apr 2001. An extended version is also available as a technical report YCS-99-320 September 1999.

[BCG+99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, , and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 1(17):5–22, Jul 1999.

[BCM99] S. Baruah, D. Chen, and A. Mok. Static-priority scheduling of multiframe tasks. In *Proceedings of the Euromicro Conference on Real-Time Systems*, Jun 1999.

[BGP95] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, Apr 1995.

[BH97] Sanjoy Baruah and Jayant Haritsa. Scheduling for overload in real-time systems. *IEEE Transactions on Computers*, 9(46):1034–1039, Sep 1997.

[BH98] Sanjoy Baruah and Mary Ellen Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers*, 47(9):1027–1032, Sep 1998.

[BHR93] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[BHS01] Sanjoy Baruah, Jayant Haritsa, and Nitin Sharma. On-line scheduling to maximize task completions. *Combinatorial Mathematics and Combinatorial Computing*, (39):65–78, 2001.

[BKM+91] S. Baruah, G. Koren, B. Mishra, D. Mao, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 106–115, Dec 1991.

[BM01] L Porto Barreto and G Muller. Bossa: A dsl framework for application-specific scheduling policies. Technical Report PI-1384, IRISA, March 2001. biblio d2.

[BM02] Luciano Porto Barreto and Gilles Muller. Bossa: a language-based approach for the design of real-time schedulers. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[Bon92] F. Boniol. Le temps-réel par une approche conjointe synchrone - asynchrone. Technical Report 1/3410.00/DERI, CERT, mar 1992.

[BPB+00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[BS93] G. Buttazzo and J. Stankovic. Red: Robust earliest deadline scheduling. In *Proceedings of the International Workshop on Responsive Computing Systems*, 1993.

[BSS94] A. Bondavalli, J. Stankovic, and L. Strigini. Adaptable fault tolerance for real-time systems. Technical Report UM-CS-1994-039, University of Massachusetts, Amherst, Computer Science, May, 1994.

[BSS95] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 15th Real-Time System Symposium*, pages 90–99, Dec 1995.

[Bur93] A. Burns. Fixed priority scheduling with deadlines prior to completion. Technical Report YCS212, 1993.

[Bur94] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. Technical Report 214, 1994.

[But97] G.C. Buttazzo. *ard Real-Time Computing Systems.* Kluwer academic, 1997.

[BWH93] A. Burns, A. J. Wellings, and A. D. Hutcheon. The impact of an ada runtime system's performance charactersitics on scheduling models. In *Proceedings of the 12th Ada Europe conference*, 1993.

[CA97] Seonho Choi and Ashok K. Agrawala. Scheduling aperiodic and sporadic tasks in hard real-time systems. Technical Report CS-TR-3794, 1997.

[CB97] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997.

[CCS02] L. Cucu, R. Cocif, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[CFBW93] C.Bailey, E. Fyfe, A. Burns, and A. J. Wellings. The olympus attitude and orbital control system, a case study in hard real-time system design and inplementation. Technical report, University of York, 1993.

[CJ98] George Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998. USENIX.

[CLB99] Marco Caccamo, Giuseppe Lipari, and Giorgio Buttazzo. Sharing resources among periodic and aperiodic tasks with dynamic deadlines. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[Coo83] Stephen A. Cook. An overview of computational complexity. *Communications of the ACM*, 26(6):400–408, 1983.

[CP99a] P. Chevochot and I. Puaut. An approach for fault-tolerance in hard real-time distributed systems. In *Proc. 18th IEEE Symposium on Reliable Distributed Systems (SRDS'99) (short paper)*, pages 292–293, Lausanne, Switzerland, October 1999.

[CP99b] P. Chevochot and I. Puaut. Tolérance aux fautes dans les systèmes répartis temps-réel strict. *Techniques et Sciences Informatiques (TSI)*, 18(8):837–870, October 1999.

[CP01a] A. Colin and I. Puaut. Analyse de temps d'exécution au pire cas du système d'exploitation temps-réel rtems. In *Seconde Conférence Française sur les Systèmes d'Exploitation (CFSE2)*, pages 73–84, Paris, France, April 2001.

[CP01b] A. Colin and I. Puaut. Worst-case execution time analysis of the rtems real-time operating system. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 191–198, Delft, The Netherlands, June 2001.

[CPC+00] P. Chevochot, I. Puaut, G. Cabillic, A. Colin, D. Decotigny, and M. Banâtre. Hades: a distributed system for dependable hard real-time applications built from cots components. Technical Report PI1357, IRISA, October 2000.

[Dav93] R. Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical Report 216, U. York, Dec 1993.

[Dav94] R. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS230, University of York, UK, May 1994.

[Del96] J. Delacroix. Towards a stable earliest deadline scheduling algorithm. *Real-Time Systems*, 10:263–291, 1996.

[DFP01] Radu Dobrin, Gerhard Fohler, and Peter Puschner. Translating offline schedules into task attributes for fixed priority scheduling. *22nd IEEE Real-Time Systems Symposium, December 2001, London, United Kingdom*, Dec. 2001.

[DFW02] W. Dinkel, M. Frisbie, and J. Woltersdorf. *Kurt-Linux User Manual*. UC Kansas at Lawrence, Mar 2002.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pages 222–231, 1993.

[EJ00] Cecilia Ekelin and Jan Jonsson. Solving embedded system scheduling problems using constraint programming. In *IEEE RTSS*, May 2000.

[FGB01] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, pages 183–192, Dec 2001.

[fIT93] IEEE Standard for Information Technology. *Portable Operating System Interface 1003.1b & 1003.1c*. IEEE, 1993.

[Foh94] Gerhard Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1994.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[Gar99] M. K. Gardner. *Probabilistic Analysis and Scheduling of Critical Soft Real-Time Systems*. PhD thesis, University of Illinois at Urbana-Champaign, Sep 1999.

[GBF02] J. Goossens, S. Baruah, and S. Funk. Real-time scheduling on multiprocessors. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[GFB01] Joel Goossens, Shelby Funk, and Sanjoy Baruah. Edf scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations. Technical Report UNC-CS TR01-033, UNC, Nov 2001.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of Np-Completeness*. Number ISBN 0716710455. W H Freeman & Co., 1979.

[GL95] Donald W. Gillies and Jane W.-S. Liu. Scheduling tasks with AND/OR precedence constraints. *SIAM J. Comput.*, 24(4):797–810, 1995.

[GL99] Mark K. Gardner and Jane W. S. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. In *Proceedings of the Eleventh Euromicro Conference on Real-Time Systems*, Jun 1999.

[GLLK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[GMR95] L. George, P. Muhletahler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. Technical Report 2516, INRIA, 1995.

[GNM99] Bruno Gaujal, Nicolas Navet, and Jorn Migge. Dual-priority versus background scheduling: A path-wise comparison. Technical-report, Inria, Institut National de Recherche en Informatique et en Automatique, July 1999.

[GP96] R. Gopalakrishnan and Guru M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1996.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report 2966, INRIA Rocquencourt, sep 1996.

[GS96] M. Gergeleit and H. Streich. Taskpair-scheduling with optimistic case execution times – an example for an adaptive real-time system. In *Second International Workshop on Object-oriented Real-time Dependable Systems, Laguna Beach, CA*, GMD, Germany, February 1,2 1996.

[GSSR97] O. Gonzalez, H. Shrikumar, John A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time constraints. In *Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, California*. U. Mass, December 1997.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16, 1998.

[Heu01] Arnd Christian Heursch. Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4. In *Proceedings of the Real-Time & Embedded Computing Expo & Conference*, 2001.

[HMT90] Chetto H., Silly M., and Bouchentouf T. Dynamic scheduling of real-time task under precedence constraints. *Real Time Systems*, (2):181–194, 1990.

[HS90] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on Software Engineering*, 16(2):1374–1389, 1990.

[HV95] Rodney R. Howell and Muralidhar K. Venkatrao. On non-preemptive scheduling of recurring tasks using inserted idle times. *Information and Computation*, 117(1):50–62, 1995.

[IF99] Damir Isovic and Gerhard Fohler. Online handling of firm aperiodic tasks in time triggered systems. In *11th Euromicro Conference on Real-Time Systems*, York, UK, June 1999.

[IMR96] Hong Inki, Potkonjak Miodrag, and Karri Ramesh. Hetergeneous BISR-approach using system level synthesis flexibility. Technical Report 960047, University of California, Los Angeles, Computer Science Department, December 31, 1996.

[JD90] Xu J. and Parnas D. Scheduling processes with release times, deadlines, precedence, and exclusion, relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, 1990.

[Jef89] K. Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with preemption constraints. In *IEEE Real-Time Systems Symposium*, 1989.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE Real-Time Systems Symposium*, pages 89–99, 1992.

[JG99] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proceedings of the of the 20th IEEE Real-Time Systems Symposium*, Dec 1999.

[JG01] K. Jeffay and S.M. Goddard. Rate-based resource allocation models for embedded systems. In *Proceedings of the First International Workshop on Embedded Software*, pages 204–222, Oct 2001.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[JS93] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 212–221, Dec 1993.

[JSM91] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In IEEE, editor, *Proceedings of the 12 th IEEE Symposium on Real-Time Systems (December 1991)*, pages 129–139, december 1991.

[KAS93] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider. Engineering and analysis of fixed priority schedulers. *Software Engineering*, 19(9):920–934, 1993.

[KFG+92] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchotichy, and R. Zainlinger. The programmer's view of MARS. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1992*, pages 223–226, Phoenix, Arizona, USA, December 1992. IEEE Computer Society Press.

[KNH+97] Hermann Kopetz, Roman Nossal, René Hexel, Andreas Krüger, Dietmar Millinger, Roman Pallierer, Christopher Temple, and Markus Krug. Mode handling in the time-triggered architecture. *IFAC DCCS 97, June 1997, Seoul, Korea*, Jun. 1997.

[KS93] G. Koren and D. Shasha. D^{over}: An optimal online scheduling algorithm for overloaded realtime systems. In *Proceedings IEEE Real-Time Systems Symposium*, pages 290–299, 1993.

[KS95] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995.

[KSSR96] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. Technical Report UM-CS-1996-045, University of Massachusetts, Amherst, Computer Science, December, 1996.

[Law73] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19, 1973.

[Law78] E.L. Lawler. Sequencing jobs to minimize total weighted completion time. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[Law83] Eugene L. Lawler. Scheduling a single machine to minimize the number of late jobs. Technical Report CSD-83-139, UC Berkeley, 1983.

[LB00a] G. Lipari and G.C. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of System Architectures*, 46:327–338, 2000.

[LB00b] Giuseppe Lipari and Sanjoy Baruah. Efficient scheduling of real-time multitask applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 166–175, May 2000.

[LB01] G. Lipari and S.K. Baruah. A hierarchical extension to the constant bandwidth server framework. In *Proceedings of the Real-Time Technology and Application Symposium*, 2001.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLS+91] Jane W.-S. Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[LMA88] S. T. Levi, D. Mosse, and A. K. Agrawala. Allocation of real-time computations under fault-tolerance constraints. In *Proceedings IEEE Real-Time Systems Symposium*, pages 161–170, 1988.

[LSA+00] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Dec 2000.

[LSD89] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LT94] John P. Lehoczky and Sandra R. Thuel. Scheduling periodic and aperiodic tasks using the slack stealing algorithm. *Advances in Real-Time Systems*, pages 172–195, 1994.

[LW82] J. Leung and J.W. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.

[Man98] G. Manimaran. *Resource management with dynamic scheduling in parallel and distributed real-time systems*. PhD thesis, Indian Institute of Technology, Madras, jan 1998.

[MB97] C. McElhone and A. Burns. Scheduling optional computations for adaptive real-time systems. Technical Report YCS289, 1997.

[MC96a] Aloysius K. Mok and Deji Chen. A general model for real-time tasks. Technical Report CS-TR-96-24, 1996.

[MC96b] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. In *Proceedings of the Real-Time Systems Symposium*, 1996.

[McE94] C. McElhone. Adapting and evaluating algorithms for dynamic schedulability testing. Technical report, AC York, Feb 1994.

[McN59] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 12:1–12, 1959.

[Mer92] Clifford W. Mercer. An introduction to real time operating systems: Scheduling theory. Technical report, CMU, Nov 1992. Unpublished manuscript.

[MF02] Aloysius K. Mok and Alex Xiang Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *Second International Conference on Embedded Software, EMSOFT 2002, LNCS 2491*, Oct 2002.

[Mig99] Jörn Migge. *L'ordonnancement sous contraintes temps-réel : un modèle à base de trajectoires*. PhD thesis, INRIA Sophia Antipolis, Nov 1999.

[MM97] G. Manimaran and C. Siva Ram Murthy. A new scheduling approach supporting different fault-tolerant techniques for real-time multiprocessor systems. *Journal of Microprocessors and Microsystems*, 21(3):163–173, dec 1997.

[MMM00a] G. Manimaran, A. Manikutty, and C Siva Ram Murthy. Dharma: A tool for evaluating dynamic scheduling algorithms for real-time multiprocessor systems. *Journal of Systems and Software*, 50(2):131–149, Feb 2000.

[MMM00b] A. Mittal, G. Manimaran, and C. Siva Ram Murthy. Integrated dynamic scheduling of hard and qos degradable real-time tasks in multiprocessor systems. *to appear in Journal of Systems Architecture*, 2000.

[Mok83] A.K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, Jun 1983.

[MS95] M. Marucheck and J. Strosnider. An evaluation of the graceful degradation properties of real-time schedulers. In *25th Annual International Symposium on Fault-Tolerant Computing*, June 1995.

[MW01] Aloysius K. Mok and Weirong Wang. Window-constrained real-time periodic task scheduling. In *Workshop on Real-Time Embedded Systems*, Dec 2001.

[MZ93] C. E. Moron and H. Zedan. Adaptable scheduler using milestones for hard real-time systems. Technical report, UYork, 1993.

[Nav99] Nicolas Navet. *Évaluation de performances temporelles et optimisation de l'ordonnancement de tâches et messages*. PhD thesis, Institut National Polytechnique de Lorraine, Nov 1999.

[Net97] E. Nett. Real-time behaviour in a heterogeneous environment. In *Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, CA*, GMD, Germany, February 6-7 1997.

[NG97a] E. Nett and M. Gergeleit. Preserving real-time behavior in dynamic distributed systems. In *IASTED International Conference on Artificial Intelligence and Soft Computing, The Bahamas*, GMD, Germany, December 8–10 1997.

[NG97b] E. Nett and M. Gergeleit. Preserving real-time behavior un dynamic distributed systems. In IEEE, editor, *IEEE IASTED International Conference on Intelligent Information Systems*, Grand Bahama Island, The Bahams, December 1997.

[NGM98] E. Nett, M. Gergeleit, and M. Mock. An adaptive approach to object-oriented real-time computing. In *Proceedings of ISORC'98, Kyoto, Japan*, GMD, Germany, April 20–22 1998.

[NGM01] E. Nett, M. Gergeleit, and M. Mock. Enhancing oo middleware to become time-aware. *RTS Journal*, ??:??, ? 2001.

[NGS97] E. Nett, M. Gergeleit, and H. Streich. Flexible resource scheduling and control in an adaptive real-time environment. In *IASTED International Conference on Artificial Intelligence and Soft Computing, Banff, Canada*, GMD, Germany, July 27 –August1 1997.

[Nic98] Guillem Bernat Nicolau. *Specification and Analysis of Weakly Hard Real-Time Systems*. PhD thesis, Universitat de les Illes Balears, Jan 1998.

[oCCL00] University of Cambrifge Computer Laboratory. *The Nemesis System Documentation*, jan 2000.

[Pua02] I. Puaut. Real-time performance of dynamic memory allocation algorithms. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[RH01] Mario Aldea Rivas and Michael González Harbour. Posix-compatible application-defined scheduling in marte os. Technical report, Universidad de Cantabria. SP., 2001.

[Rie98] Marco Riedel. Classification of deterministic scheduling problems (survey and notation), Aug 1998.

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shui Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, Jan 1998.

[RM00] S. Ramamurthy and M. Moir. Static-priority periodic scheduling of multiprocessors. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 69–78, Dec 2000.

[RRGC02] Michael Richard, P. Richard, E. Grolleau, and F. Cottet. Contraintes de précédences et ordonnancement mono-processeur. In *Proceedings of the Conférence Internationale sur les Systèmes Temps-Réel*, Mar 2002.

[RS94] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In IEEE, editor, *Proceedings of the IEEE*, volume 82, January 1994.

[RS01] J. Regehr and J. Stankovic. Hls: A framework for composing soft real-time schedulers. In *Proceedings of the Real-Time Systems Symposium*, pages 3–14, Dec 2001.

[RSH00] John Regehr, Jack Stankovic, and Marty Humphrey. The case for hierarchical schedulers with performance guarantees. Technical Report CS-2000-07, CS Virginia, 14, 2000.

[RSYJ97] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In IEEE, editor, *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, volume GIT-CC-97-26, dec 1997.

[SA00] K. Subramani and Ashok Agrawala. A dual interpretation of "standard constraints" in parametric scheduling. In *Proceedings of the Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 121–133, Sep 2000.

[SAWJ+96] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[SB94] M. Spuri and G. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time System Symposium*, pages 2–21, 1994.

[SBS95] M. Spuri, G. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *Proc. of the IEEE Real-Time Systems Symposium, Pisa, Italy*, dec 1995.

[SdSA95] M. Saksena, J. da Silva, and A. K. Agrawala. Design and implementation of maruti-ii. *Advances in Real-Time Systems*, pages 72–102, 1995.

[SG97a] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In *11th International Parallel Processing Symposium, University of Geneva, Switzerland*, GMD, Germany, April 1-5 1997. University of Geneva, Switzerland.

[SG97b] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In IEEE, editor, *IEEE Workshop Parallel Distributed Real-Time Systems (WPDRTS)*, Geneva, Switzerland, 1997.

[SGL97] Jun Sun, Mark K. Gardner, , and Jane W. S. Liu. Bounding completion times of jobs with arbitrary release times, variable execution times, and resource sharing. *IEEE Transactions on Software Engineering*, 23(10):603–615, Oct 1997.

[SL94] Matthew F. Storch and Jane W.-S. Liu. A simulation environment for distributed real-time systems. In *Proceedings of the SCS Simulation Multiconference*, April 1994.

[SL95] WK Shih and WS Liu. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *Proceedings of the IEEE Transactions on Computers*, 44(3), March 1995.

[SLL93] Wei Kuan Shih, Jane W.-S. Liu, and C. L. Liu. Modified rate-monotonic algorithm for scheduling periodic jobs with deferred deadlines. *Software Engineering*, 19(12):1171–1179, 1993.

[SLS95] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions On Computers*, 44(1):73–91, Jan 1995.

[SLS98] John A. Stankovic, Chenyang Lu, and Sang H. Son. The case for feedback control real-time scheduling. Technical Report CS-98-35, Department of Computer Science, University of Virginia, November 27 1998.

[SNF98] Kristian Sandström, Christer Norström, and Gerhard Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of the International Conferance on Real-Time Computing Systems and Applications*, pages 158–165, Oct 1998.

[SP97] Steven Sommer and John Potter. Admissibility tests for interrupted earliest deadline first scheduling with priority inheritance. Technical Report C/TR97-10, U Macquarie, Australia, Jun 1997.

[Spr90] B. Sprunt. *Aperiodic task scheduling for real-time systems.* PhD thesis, Carnegie Mellon University, Aug 1990.

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, INRIA Rocquencourt, jan 1996.

[SRG89] W. Schwabl, J. Reisinger, and G. Grunsteidl. A survey of mars. Technical report, Institut fur Technische Informatik, Technical University Wein, 1989.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept 1990.

[SRN+98] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, and Gary Wallace. The spring system: Integrated support for complex real-time systems. Technical Report CS-98-18, Department of Computer Science, University of Virginia, August 1 1998.

[SS93] Marco Spuri and John A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. Technical Report UM-CS-1993-019, U. Mass, 1993.

[SSDB94] J. Stankovic, M. Spuri, M. DiNatale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. Technical Report UM-CS-1994-089, U. Mass, 1994.

[SSNB94] J.A. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. Technical report, Scuola Superiore S.Anna, Pisa - Italy, June 1994.

[Sta88] John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, 1988.

[Sta93a] John A. Stankovic. Reflective real-time systems. Technical Report UM-CS-1993-056, University of Massachusetts, Amherst, Computer Science, June, 1993.

[Sta93b] Prof. John A. Stankovic. Major real-time challenges for mechatronic systems. In *to appear in Proceedings of International Workshop on Mecha-*

tronical Computer Systems for Perception and Action, June 1993, Massachussetts University, May 14 1993.

[Sta94] John A. Stankovic. Adjustable flow control filters and reflective memories as support for distributed real-time systems. Technical Report UM-CS-1994-034, University of Massachusetts, Amherst, Computer Science, April, 1994.

[Str95] H. Streich. Taskpair-scheduling: An approach for dynamic real-time systems. *Int. Journal of Mini & Microcomputers*, 17, No. 2:77–83, 1995.

[TBW92a] K. Tindell, A. Burns, and A.J. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real-Time Systems*, 4(2):145–165, Jun 1992.

[TBW92b] Ken Tindell, Alan Burns, and Andy J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 100–109, 1992.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[TDS+95] T. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L. Wu, and J. Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 164–173, may 1995.

[TH99] Henrik Thane and Hans Hansson. Handling interrupts in testing of distributed real-time systems. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.

[Tin92a] K. Tindell. An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS189, 1992.

[Tin92b] K. Tindell. Using offset information to analyse static priority pre-emptively scheduled task sets. Technical Report YCS182, Aug 1992.

[Tin93] K.W. Tindell. *Fixed Priority Scheduling of Hard Real-Time Systems*. PhD thesis, University of York, Dec 1993.

[Tin94] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS221, 1994.

[TLS95] T. Tia, J. Liu, and M. Shankar. Algorithms and optimality of scheduling aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*, 1995.

[TLSH94] T. Tia, W. Liu, J. Sun, and R. Ha. A linear-time optimal acceptance test for scheduling of hard real-time tasks, 1994.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.

[Vij98] Josh Haines Vijay. Development of application-level fault tolerance in a real-time benchmark. In *Proceedings of the IEEE Workshop On Embedded Fault-Tolerant Systems*, May 1998.

[VJP97] Brian VanVoorst, Rakesh Jha, and Luiz Pires. A real-time parallel benchmark suite. *SIAM's Parallel Processing for Scientific Computing*, 1997.

[WL99a] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-linux real-time kernel. In *IEEE Real-Time Systems Symposium*, pages 246–255, 1999.

[WL99b] Yu-Chung Wang and Kwei-Jay Lin. Providing real-time support in the linux kernel. In *IEEE Real-Time Technology and Applications Symposium*, 1999.

[WS99a] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.

[WS99b] Lonnie R. Welch and Behrooz A. Shirazi. A dynamic real-time benchmark for assessment of qos and resource management technology. In *Real-Time Technology and Applications Symposium*, Jun 1999.

[WS02] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold: An attractive technology?, 2002. Regehr's readings.

[Zub98] Khawar M. Zuberi. *Real-Time Operating System Services for Networked Embedded Systems*. PhD thesis, University of Michigan, 1998.

Liens référencés

[1] The rtai programming guide.
http://www.rtai.org
et http://www.aero.polimi.it/~rtai/documentation/index.html.

[2] Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski, and Gerhard Woeginger, A compendium of np optimization problems.
http://www.nada.kth.se/~viggo/wwwcompendium/.

[3] Office de la langue Française du Canada, Le grand dictionnaire terminologique.
http://www.granddictionnaire.com.

[4] Délégation générale à la langue française et aux langues de France, Base de données.
http://www.culture.fr/culture/dglf/terminologie/base-donnees.html.

[5] Glenn Reeves, What really happened on mars? Jan 1998.
http://catless.ncl.ac.uk/Risks/19.54.html#subj6.

[6] TRON Association Version-Up Working Group, Industrial real-time operating system nucleus (itron).
http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html.

Table des matières

Liste des tableaux

Table des figures

Index

T

U

V

W