HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

par

Isabelle Puaut

Supports d'exécution à temps de réponse contraint tolérants aux fautes

soutenue le 30 novembre 2001 devant le jury composé de

M.	Jean Camillerapp	Président
Mme	Yolande Berbers	Rapporteur
M.	Jean-Pierre Elloy	Rapporteur
M.	David Powell	Rapporteur
M.	Michel Banâtre	Examinateur
M.	André Seznec	Examinateur

Les plus courtes erreurs sont toujours les meilleures

Pierre Charron
Traité de la sagesse, 1601

Remerciements

En premier lieu, je remercie Yolande Berbers, David Powell, et Jean-Pierre Elloy, d'avoir accepté de rapporter sur mes travaux, malgré leur emploi du temps plus que chargé, ainsi qu'André Seznec pour sa participation au jury.

Je tiens à remercier Michel Banâtre, responsable scientifique du projet Solidor, pour la confiance qu'il m'a accordée tout au long de ces années, et qui m'a très vite permis de participer à l'encadrement des doctorants. Il a été à l'origine de mes travaux de recherche dans le domaine du temps-réel. Sans cette initiative, je n'aurais pas eu l'occasion de travailler sur ce thème passionnant.

De nombreuses personnes ont participé au travail relaté dans ce document, qui est avant tout un travail collectif. Les doctorants qui ont contribué à ce travail sont, (par ordre chronologique) Gilbert Cabillic, Pascal Chevochot, Antoine Colin et David Decotigny. Gilbert Cabillic a travaillé sur l'exécution d'applications parallèles en univers hétérogène. Pascal Chevochot, Antoine Colin et David Decotigny (la "HADES team") ont travaillé sur l'analyse et l'exécution d'applications temps-réel critiques. Leur participation active et leur volonté de travailler en commun m'a permis de bâtir un ensemble d'études et d'outils formant un tout cohérent. Sans leur participation, ce travail n'aurait certainement pas existé, et par conséquent, ce document d'habilitation est autant le mien que le leur. Je remercie tout particulièrement Pascal, pour les discussions (scientifiques ou non) animées qu'on a pu avoir dans le bureau et tout le temps passé en relectures d'articles et documents (dont celui-ci).

Le travail présenté dans ce document a été validé via la construction de logiciels prototypes, à la construction desquels de nombreuses personnes ont participé : ingénieurs experts (G. Cabillic, P. Chevochot), scientifiques du contingent (S. Benoist, D. Decotigny, S. Tudoret), stagiaires (G. Cadou, T. Colibert, J. Gloaguen, C. Ronsmans, P. Selo, M. Bellengé) et deux groupes d'étudiants de quatrième année INSA. Que chacun soit remercié de sa participation.

Je remercie également les membres passés et actuels du projet Solidor, avec qui j'ai eu l'occasion de travailler et d'avoir des discussions passionnantes, notamment Emmanuelle Anceaume, Valérie Issarny, Christine Morin et Gilles Muller, ainsi que l'ensemble des doctorants et ingénieurs experts passés et présents du projet pour la bonne ambiance qu'ils y ont fait régner.

Enfin, je tiens à remercier l'ensemble de mes collègues enseignants-chercheurs de l'INSA pour leur soutien tout au long de ces années. Je remercie tout particulièrement

Marie-Jo Pédrono et Ivan Leplumey qui ont eu l'extrême gentillesse d'avoir pris en charge mes enseignements pendant mes deux congés maternité et ma période de délégation auprès du CNRS, Mireille Ducassé, qui a supporté ma présence (et mon désordre) dans le bureau, et Jean Camillerapp, pour ses conseils, encouragements, et exercices semestriels de sport cérébral (il les reconnaîtra), auxquels j'ai pris beaucoup de plaisir.

Table des matières

Chapitre 1

Introduction

Ce document d'habilitation relate les activités de recherche que j'ai menées depuis 1993, année de soutenance de ma thèse, à l'IRISA Rennes, au sein du projet Solidor. L'objectif général de ces recherches est la construction de *supports d'exécution* pour applications *distribuées* (s'exécutant sur un ensemble de machines ne disposant pas d'une mémoire vive commune). Dans le document, le terme *support d'exécution* désigne l'ensemble des couches de logiciel nécessaires à l'exécution des applications, qu'elles s'exécutent en mode noyau ou en mode utilisateur, sous la forme de bibliothèques liées avec l'application à exécuter. Plus précisément, je me suis attachée à concevoir des supports d'exécution prenant en considération deux types d'exigences :

- *Exigences opérationnelles* via la fourniture de mécanismes d'exécution adaptés à la classe d'applications que l'on cherche à supporter. Je me suis intéressée plus spécifiquement à faire *cohabiter* de manière harmonieuse des méthodes pour assurer des *temps de réponse* adéquats pour les applications, et des méthodes pour *tolérer les fautes* physiques pouvant affecter un élément de l'architecture distribuée.
- *Exigences économiques.* Construire un support d'exécution à partir de rien est une tâche très consommatrice en temps. J'ai donc étudié de manière critique le bien fondé d'intégrer des composants généralistes existants (système d'exploitation notamment) pour construire des supports d'exécution répondant aux exigences des applications. Mes travaux dans ce cadre ont concerné le processus de *sélection* des composants à intégrer (évaluation des propriétés des composants) et des méthodes pour *compléter* leurs fonctionnalités pour qu'ils puissent être utilisés.

Les mécanismes à intégrer dans un support d'exécution distribué sont fortement dépendants du type d'applications à exécuter. Au long de mon parcours de recherche, je me suis penchée sur deux grandes classes d'applications : les applications parallèles de longue durée, et les applications temps-réel critiques. Les besoins de ces deux classes d'applications sont dégagés ci-dessous, avant d'introduire les travaux de recherche que j'ai menés pour fournir les mécanismes adaptés à leur exécution.

1.1 Domaines d'applications étudiés

1.1.1 Applications parallèles de longue durée

La première classe d'applications étudiée regroupe les applications parallèles de longue durée exécutées sur réseaux de stations de travail. Ces applications sont par exemple des applications de simulation numérique, utilisées dans de nombreux domaines (dynamique des fluides, électromagnétisme). De telles applications sont conçues sous la forme de processus s'exécutant en parallèle sur les machines de l'architecture distribuée (nous ne nous intéressons pas ici à l'extraction automatique de parallélisme dans des applications séquentielles). Le principal besoin pour ce type d'applications est de fournir des mécanismes d'exécution de processus, et de coopération entre processus qui soient *efficaces*, c'est-à-dire qui permettent de réduire le temps d'exécution de l'application sur l'architecture distribuée par rapport à son exécution sur une seule machine. De plus, étant donnée la classe d'architecture visée pour l'exécution des applications (architectures à mémoire distribuée), la probabilité de défaillance ou d'arrêt volontaire d'une des machines de l'architecture est non négligeable. Il est donc nécessaire que le support d'exécution fournisse des mécanismes de *tolérance aux fautes* pour tolérer l'arrêt (programmé ou accidentel) d'une machine.

1.1.2 Applications temps-réel critiques

La deuxième classe d'applications considérées regroupe les applications distribuées *critiques* pour lesquelles un fonctionnement non conforme aux spécifications de l'application peut occasionner des conséquences humaines, écologiques ou financières catastrophiques. On trouve de telles applications dans le domaine de la production d'énergie, du contrôle de production, des transports (avionique, automobile). Nous nous penchons plus particulièrement sur les applications ayant des contraintes de temps de réponse stricts (temps-réel *strict* ou *dur*), pour lesquelles le respect des contraintes de temps de réponse des applications est impératif. Étant données les conséquences catastrophiques en cas de non respect des spécifications, les applications temps-réel critiques doivent tolérer les fautes, qu'elles soient d'origine humaine ou physique (nous nous intéressons dans ce document exclusivement aux fautes physiques). De plus, les mécanismes de *tolérance aux fautes* doivent non seulement exister, mais également être compatibles avec les contraintes de temps-réel strict. En autres termes, les contraintes temporelles des applications doivent toujours être respectées, y compris en présence de fautes. Cet impératif de respect des contraintes temps-réel nécessite l'utilisation de techniques (analyse d'ordonnançabilité) permettant d'assurer *a priori* (avant exécution) le respect des contraintes temps-réel. La présence de contraintes de temps-réel strict influence de manière forte la conception d'un support d'exécution distribué, car toutes ses activités doivent impérativement avoir des temps de réponse bornés, de borne connue et compatible avec les échéances des applications, et ce même en présence de fautes.

1.2 Vue d'ensemble des travaux de recherche

Mes travaux de recherche ont porté sur la conception de supports d'exécution distribués en prenant en compte à la fois des exigences opérationnelles (en terme de temps de réponse et de tolérance aux fautes), et économiques.

Prise en compte d'exigences opérationnelles

Les exigences opérationnelles que nous avons prises en compte pour les deux domaines d'applications étudiés sont de tolérer les fautes des calculateurs, et d'assurer des temps de réponse adaptés pour les applications :

- *Tolérance aux fautes.* Pour les deux domaines d'application étudiés, nous avons proposé des méthodes permettant de tolérer les fautes d'origine physique de manière adaptée au domaine d'application. Pour les applications parallèles de longue durée, nous avons mis en place une méthode de recouvrement d'erreur par *reprise* (*backward error recovery*) car elle est économique en ressources en l'absence de défaillances ; des optimisations ont été mises en place de manière à minimiser le temps de sauvegarde de l'état des processus. En ce qui concerne les applications temps-réel critiques, l'accent a été mis sur *l'analyse d'ordonnançabilité* des méthodes de tolérance aux fautes plutôt que sur la proposition de nouvelles méthodes de tolérance aux fautes.
- *Temps de réponse*. Pour chacun des deux domaines d'applications considérés, nous avons proposé des méthodes donnant des garanties plus ou moins fortes de temps de réponse aux applications. Pour les applications parallèles de longue durée, la méthode proposée est une méthode d'*équilibrage de charge* qui permet de diminuer le temps de réponse des applications dans les situations de fluctuation de la charge des machines du système. Pour les applications temps-réel critiques, nous avons mis en place des méthodes d'*analyse statique de temps d'exécution* (permettant à partir d'un code source d'obtenir son pire temps d'exécution sur une architecture matérielle donnée) et d'*analyse d'ordonnançabilité* permettant de vérifier que le comportement temporel du système au sens large (support d'exécution et application) est correct.

Le point dur à aborder, qui constitue l'originalité de nos travaux, a été de faire *cohabiter* les deux types d'exigences. Notre démarche pour concilier tolérance aux fautes et temps de réponse est présentée de manière plus détaillée dans la suite du document.

Prise en compte d'exigences économiques : intégration de composants existants

Quel que soit le domaine d'application étudié, la construction de supports d'exécution distribués est une tâche très consommatrice en temps de développement et de maintenance. Pour limiter ces temps, nous avons cherché à *intégrer des composants existants* lors de la conception du support d'exécution. Les composants que nous

avons intégrés ("réutilisés") sont à la fois des composants matériels (calculateurs, réseaux) et des composants logiciels (système d'exploitation principalement) et sont des composants généralistes *sur étagères* (en anglais COTS pour Commercial-Off-The-Shelf).

Ainsi, pour les deux domaines d'application étudiés, nous avons développé le support d'exécution comme une couche logicielle intermédiaire (*intergiciel*, ou *middleware*) intercalée entre le système d'exploitation (composant logiciel COTS utilisé) et l'application.

Intégrer des composants existants nécessite de *caractériser* leur comportement de manière adéquate pour le domaine d'application visé. Par exemple, pour les applications temps-réel critiques, à la fois le comportement temporel des composants et leur comportement en présence de fautes est important. Nous avons par conséquent développé des méthodes et outils d'évaluation des comportements des composants (en terme de modes de défaillances et de temps d'exécution au pire cas en ce qui concerne les aspects temps-réel). Par ailleurs, il peut être nécessaire de *compléter* les fonctionnalités des composants intégrés pour les rendre adaptés à l'exécution des applications (par exemple, gérer l'hétérogénéité matérielle des calculateurs utilisés pour l'exécution d'applications parallèles de longue durée). Les méthodes et outils que nous avons conçus pour caractériser et compléter les composants COTS nous ont permis, sur l'exemple, d'évaluer de manière critique le bien fondé de l'intégration de composants COTS, et l'impact de cette intégration sur les performances et la fiabilité du support d'exécution résultant.

Expérimentations

Tous les travaux présentés dans ce document ont été intégrés dans des logiciels prototypes afin d'évaluer leurs performances : le support d'exécution pour applications parallèles Stardust [CP97], et le support d'exécution HADES [CPC+01], accompagné d'un environnement de développement, pour les applications temps-réel strict sûres de fonctionnement. Pour des raisons de place, les méthodes que nous avons proposées sont assez peu développées dans ce document. Toutefois, nous consacrons une place importante à l'évaluation expérimentale de leur performance, afin d'être apte à juger de leur adaptation au domaine d'application visé.

1.3 Organisation du document

Les applications parallèles de longue durée et les applications temps-réel critiques ont des besoins très différents en terme de temps de réponse et de tolérance aux fautes. Pour cette raison, nous avons choisi de présenter nos travaux en deux chapitres séparés, mais organisés de manière identique, correspondant aux deux domaines d'application visés.

Pour chacun des deux domaines d'application, nous détaillons tout d'abord quels sont les besoins des applications de ce domaine sur le support d'exécution, en terme

de temps de réponse et de tolérance aux fautes. Nous présentons alors la démarche que nous avons adoptée pour répondre de manière *conjointe* aux exigences de temps de réponse et de tolérance aux fautes. Nous détaillons ensuite, de manière séparée pour plus de lisibilité, les méthodes que nous avons proposées pour tolérer les fautes d'une part et pour garantir (de manière plus ou moins forte) les temps de réponse d'autre part. D'un point de vue méthode de construction du support d'exécution, nous étudions les conséquences de l'intégration de composants (logiciels et matériels) existants pour limiter les coûts de développement. Enfin, nous donnons une brève description des prototypes qui ont été construits pour valider nos travaux, et analysons leurs performances. Cette analyse de performances nous permet d'effectuer un bilan des travaux effectués dans chacun des deux cadres applicatifs.

Nous concluons ce document par un bilan des travaux effectués et les perspectives ouvertes par ces travaux.

Chapitre 2

Support d'exécution pour applications parallèles de longue durée

2.1 Domaine d'application

Nous nous intéressons dans ce chapitre à l'exécution d'applications numériques de longue durée sur réseaux de stations de travail hétérogènes. Ces applications, très gourmandes en temps de calcul et en mémoire, sont par exemple des applications de simulation numérique, utilisées dans de nombreux domaines tels que la mécanique des structures, la dynamique des fluides ou l'électromagnétisme.

Très longtemps, les applications parallèles numériques de longue durée ont été exécutées sur des calculateurs massivement parallèles (à mémoire partagée ou distribuée). Principalement pour des raisons de coût, l'utilisation de réseaux de stations de travail est rapidement apparue comme une solution attrayante complémentaire de celle des calculateurs massivement parallèles. En effet, des études telles que [CS93] ont montré que même pendant les horaires de travail, le taux d'inactivité des stations d'un réseau atteint en moyenne 90%. Dans ces conditions d'utilisation, la puissance cumulée de stations devient comparable à celle des calculateurs parallèles, et ce pour un coût comparativement moins élevé. Notons que les réseaux de station de travail peuvent se décliner en un ensemble d'architectures selon le type de réseau interconnectant les calculateurs : réseau à très haut débit et faible latence (on parle alors de *grappe*, en anglais CoW pour *Cluster of Workstations*) ; réseau de stations de travail banalisées (on parle alors simplement de réseau de stations, en anglais NoW pour *Network of Workstations*). Au niveau architecture, la différence principale des réseaux de stations de travail par rapport aux machines massivement parallèles est le réseau d'interconnexion entre calculateurs, de plus bas débit et de latence de transmission plus importante que dans les machines parallèles.

2.1.1 Expression de besoins

2.1.1.1 Diminuer les temps de réponse

Le besoin principal des applications parallèles de longue durée est de *diminuer leur temps d'exécution*. Ceci est généralement effectué en décomposant l'application en *processus*, qui sont placés sur les différents calculateurs de l'architecture.

La métrique couramment utilisée pour évaluer la diminution du temps de réponse d'une application est l'accélération (*speedup*) obtenue en exécutant l'application sur plusieurs processeurs au lieu d'un :

$$accélération(p) = \frac{T_1}{T_p}$$

avec T_p le temps d'exécution de l'application sur p processeurs et T_1 le temps d'exécution du meilleur algorithme séquentiel équivalent.

En complément du placement initial des processus sur les calculateurs, des méthodes de *migration de processus* à des fins d'équilibrage de charge peuvent être mises en place dans l'objectif de réduire les temps de réponse dans les situations de fluctuation de la charge des processeurs du système. Les méthodes de migration de processus sont particulièrement difficiles à mettre en place sur les réseaux de stations de travail, notamment à cause de l'hétérogénéité de l'architecture, tant au niveau matériel que logiciel.

2.1.1.2 Tolérer les arrêts de calculateurs

Un besoin également important pour l'exécution d'applications numériques de longue durée sur stations de travail est de tolérer les *arrêts de calculateurs*, qu'ils proviennent d'un arrêt de station pour des raisons de maintenance ou d'une défaillance matérielle de station. Nous nommons par la suite les mécanismes chargés de tolérer les arrêts de calculateurs mécanismes de *tolérance aux fautes* bien que les fautes d'origine matérielle ne constituent qu'une des sources possibles d'arrêts de calculateurs. La motivation de l'introduction de mécanismes de tolérance aux fautes est d'éviter de perdre des heures, voire de jours de calculs à cause d'un arrêt de calculateur.

Notons que ce besoin de tolérance aux fautes est un besoin annexe par rapport au besoin de diminution des temps de réponse. En aucun cas les mécanismes de tolérance aux fautes mis en place ne doivent dégrader les performances des applications quand ils ne sont pas utilisés, et ainsi remettre en cause l'accélération obtenue en exécutant l'application sur plusieurs calculateurs.

2.1.2 Démarche et travaux de recherche

Pour répondre aux besoins de tolérance aux fautes et de diminution des temps de réponse mentionnés plus haut, nous avons développé un support d'exécution pour applications numériques parallèles de longue durée, fondé sur le concept de mémoire

virtuelle partagée. Ce support d'exécution, dont le prototype est nommé Stardust, a été développé entre 1993 et 1996 et a fait l'objet du travail de thèse de Gilbert Cabillic.

Pour concilier accélération et tolérance aux fautes, notre démarche lors de la construction de ce support d'exécution a été de rechercher une méthode de recouvrement d'erreurs qui ait un coût en terme de temps d'exécution et de ressources consommées (mémoire, calcul) maîtrisable par le programmeur d'applications. Ceci nous a orientés vers une méthode de recouvrement d'erreurs par sauvegarde d'état restauration (checkpoint/restart), de type coordonné (coordination de tous les calculateurs pour mémoriser leur état en stockage stable). Ainsi, le coût lié au recouvrement d'erreur (sauvegarde d'état étant donné le type de méthode utilisée) apparaît uniquement lors de la sauvegarde d'état et peut être maîtrisé par le programmeur, par modification de la fréquence des mémorisations des états des processus. Par ailleurs, toujours dans un souci de limiter le coût des mécanismes de tolérance aux fautes, nous avons conçu un ensemble d'optimisations locales pour que la sauvegarde de l'état de chaque processus soit la plus rapide possible.

Ces travaux auraient été suffisants pour assurer une bonne accélération aux applications si les calculateurs servant à l'exécution avaient une charge constante et connue, par exemple si l'on réservait les processeurs d'une machine massivement parallèle pour l'exécution des applications. Or, nous nous intéressons aux réseaux de station de travail, dont la charge est difficile à anticiper et peut subir des fluctuations au cours du temps. Ces variations de charge nous ont conduits vers la définition d'un système de migration des processus de l'application, de manière à équilibrer la charge des calculateurs et ainsi continuer à assurer une bonne accélération malgré les variations de charge des machines. De manière à limiter le code à développer pour concilier accélération et tolérance aux fautes, les méthodes de tolérance aux fautes et d'équilibrage de charge reposent sur un mécanisme commun qui permet de sauvegarder de manière efficace l'état d'une application. Nous décrivons ce mécanisme et son utilisation pour la tolérance aux fautes dans le paragraphe 2.2, et son utilisation pour l'équilibrage de charge dans le paragraphe 2.3.

Afin de diminuer le coût de développement du support d'exécution, et de faciliter son portage dans différents environnements matériels et logiciels, nous avons utilisé autant que possible les composants logiciels existants dans ces environnements (systèmes d'exploitation, piles de protocoles). Le principal problème posé par cette intégration de composants existants est leur *hétérogénéité*, point que nous traitons dans le paragraphe 2.4.

Une brève description du prototype Stardust et de ses performances (§ 2.5) nous permet d'évaluer les méthodes que nous avons proposées et de dégager quelques perspectives dans le domaine (§ 2.6).

Nous donnons ci-après (paragraphe 2.1.3) une vue d'ensemble des fonctionnalités du support d'exécution, qui nous permettra de décrire dans plus de détails les résultats de nos travaux dans la suite du chapitre.

2.1.3 Principe de fonctionnement du support d'exécution

Les supports d'exécution pour applications parallèles peuvent être classés en deux catégories selon le mode de communication offert pour la communication entre processus. Certains supports d'exécution, tels que PVM [GBD+94] et MPI [GLS94] permettent aux processus de communiquer par *échange de messages*, alors que d'autre, tels que Ivy [Li86, LH89], Treadmarks [ACD+96] ou Mermaid [ZSLW92] permettent aux processus de communiquer par *mémoire virtuelle partagée* c'est-à-dire en offrant le partage de régions de mémoire virtuelle entre processus s'exécutant sur des calculateurs différents.

Le concept de mémoire virtuelle partagée (MVP, ou en anglais SVM pour *Shared Virtual Memory*), introduit par K. Li à la fin des années quatre-vingt [Li86, LH89], étend le concept de mémoire virtuelle aux architectures à mémoire distribuée. Une région de mémoire virtuelle (espace d'adressage entier dans les travaux initiaux de Li) est rendue accessible à plusieurs processus s'exécutant sur des calculateurs différents, les données (plus classiquement *pages*) migrant à la demande entre les mémoires locales de différents processeurs. La mémoire locale de chaque processeur agit comme une mémoire cache des régions accédées, permettant ainsi de réduire le nombre d'accès mémoire à distance.

Du fait de l'utilisation des mémoires locales comme mémoires cache, un même bloc mémoire doit être maintenu cohérent. Pour ce faire, différents modèles et protocoles de cohérence ont été définis (voir [AG96] pour une vue d'ensemble). Un modèle de cohérence est un contrat entre le programmeur d'application et le support d'exécution ; il définit un ensemble de règles à respecter par le programmeur d'application, et en échange assure des propriétés sur les données. Un protocole de cohérence est l'ensemble des mécanismes fournis par le support d'exécution permettant d'assurer ces propriétés.

Le support d'exécution au sein duquel nos travaux ont été menés, nommé Stardust [CP97] (voir paragraphe 2.5 pour une description du prototype) permet à des processus s'exécutant sur des calculateurs différents de partager des régions de mémoire virtuelle. Les régions sont découpées en *pages* qui constituent l'unité de transfert entre mémoires locales (voir figure 2.1). Le modèle de cohérence offert est le modèle de *cohérence atomique*, ou *forte* [CF78], établissant un ordre total entre les écritures et assurant que toute lecture retournera la dernière écriture (selon l'ordre total établi). Ce modèle est assuré par un protocole de cohérence à *invalidation sur écriture* : chaque page peut être dupliquée en mode lecture (page 1 sur la figure), mais ne peut posséder qu'un seul exemplaire en mode écriture (pages 2 et 3 sur la figure) ; lors d'une tentative d'écriture sur une page accessible en lecture seule, les copies en lecture sont invalidées (voir exemple sur la partie droite de la figure).

Par ailleurs, le support d'exécution fournit des moyens de synchroniser les processus entre eux. Le principal mécanisme de synchronisation fourni est la *barrière de synchronisation* (rendez-vous entre tous les processus de l'application). Le support d'exécution offre également à des fins de communication et de synchronisation des échanges de données point à point via des messages. Le code des applications est de

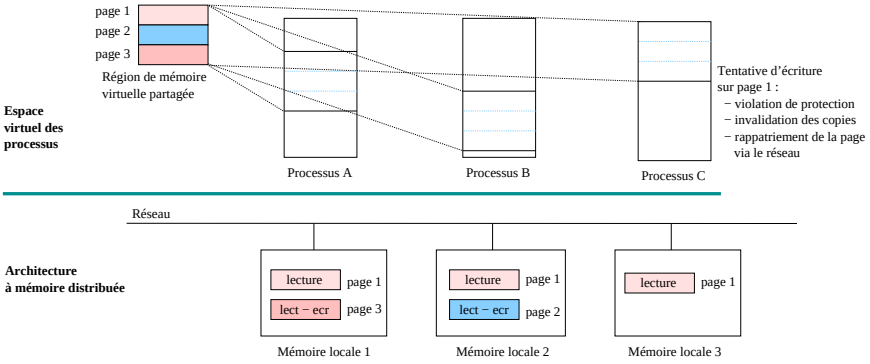


FIG. 2.1 – Support d'exécution à base de MVP (cohérence atomique, mise en œuvre par invalidation sur écriture)

type SPMD (Single Program Multiple Data) : tous les processus partagent le même code, avec toutefois la possibilité pour chaque processus de connaître son identité et ainsi effectuer des actions différentes des autres processus.

2.2 Méthode de tolérance aux fautes

Notre démarche pour concilier accélération et tolérance aux fautes dans le cadre des applications parallèles de longue durée est de définir une méthode de sauvegarde de l'état d'une application dont le coût est maîtrisable par le programmeur d'applications. Ce paragraphe, après une brève introduction des méthodes de recouvrements d'erreurs dans les systèmes à processus parallèles communiquants, présente la méthode de sauvegarde d'état que nous avons conçue puis optimisée. Cette méthode est utilisée de manière directe pour tolérer les arrêts de calculateurs. Elle sert également de base pour l'équilibrage de la charge des calculateurs (voir § 2.3).

2.2.1 Problématique

Notre objectif est de développer un support d'exécution qui soit à la fois *efficace* et qui tolère *les arrêts de machines*. Étant donnée la classe d'applications visée par le support d'exécution, le facteur efficacité domine sur le facteur fiabilité, et en aucun cas les mécanismes permettant de tolérer les arrêts de calculateurs ne doivent remettre en cause l'efficacité des applications. Cette contrainte nous a guidé dans le choix d'une méthode permettant de tolérer les arrêts de machine vers une méthode de recouvrement d'erreur par *reprise* [ABC+95] (*backward error recovery*), nommée également sauvegarde d'état/restauration (*checkpoint/restart*). Ce type de méthode consiste à sauvegarder l'état des processus de l'application sur un support insensible aux défaillances (mémoire stable [Lam81]), cet état étant utilisé pour redémarrer l'application suite à un arrêt de machine.

Les méthodes à base de *compensation*, comme par exemple la *redondance active* de processus ont été éliminées d'emblée. En effet, bien qu'en général les méthodes de redondance active ralentissent peu l'exécution des applications, chaque calcul est exécuté en plusieurs exemplaires sur des calculateurs différents. Cette consommation supplémentaire en ressources de calcul est à notre avis incompatible avec les exigences du domaine d'application étudié, pour lequel il est plus efficace d'utiliser des ressources de calcul pour accélérer l'application que pour mettre en place une méthode de redondance active.

Sauvegarder l'état d'une application composée de processus communicants est une tâche non triviale, tant au niveau *local* à chaque processus qu'au niveau *global* au système.

Au niveau local à chaque processus, pour pouvoir reprendre l'exécution d'un processus après l'avoir interrompu, il est nécessaire de mémoriser le contenu de son espace d'adressage, de sauvegarder son état d'avancement courant (région de pile et compteur de programme), ainsi que l'état courant du système d'exploitation en ce qui le concerne (fichiers ouverts, sémaphores acquis, etc.). La principale difficulté liée à la sauvegarde de l'état de chaque processus dans notre cadre de travail, outre d'assurer l'efficacité de la sauvegarde, est l'hétérogénéité des machines susceptibles d'accueillir l'exécution du processus, et ce tant au niveau matériel et logiciel (représentation des données et de l'état d'avancement de chaque processus – compteur de programme,

pile – différents selon les architectures). Ce point est abordé dans le paragraphe 2.4.2.

Au niveau global à l'application, la difficulté provient des *communications entre processus*. Considérons dans un premier temps le cas où les processus communiquent par échanges de messages. L'objectif des méthodes de sauvegarde d'état est que l'ensemble des processus redémarrent suite à un arrêt dans un *état global cohérent*. Cette notion, formalisée dans [CL85], définit un état global du système (état des processus et des canaux de communications) comme cohérent si tout message apparaissant comme reçu dans l'état du processus receveur apparaît comme ayant été envoyé dans l'état de l'émetteur (voir figure 2.2).

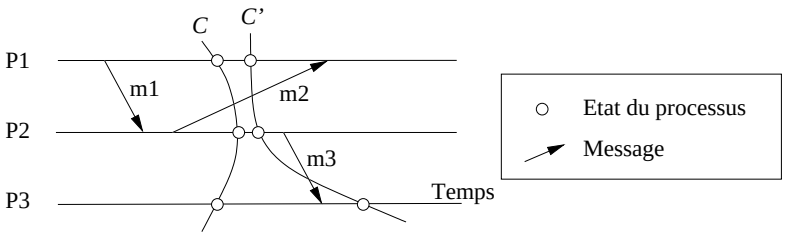


FIG. 2.2 – État global cohérent (C) et incohérent (C')

Différentes catégories de méthodes peuvent être utilisées pour sauvegarder l'état des processus de manière à redémarrer l'application à partir d'un état global cohérent (voir [EJW96] pour une description détaillée des méthodes). Dans les méthodes à sauvegarde *indépendante* des points de reprise (ou *independent checkpointing*), les processus ne se concertent pas pour mémoriser leur points de reprise respectifs, mais mémorisent les dépendances existant entre les points de reprise, pour, lors de la reprise, redémarrer à partir d'un état global cohérent. On parle aussi de méthodes *optimistes*. L'inconvénient principal de ce type de méthode est l'*effet domino* [Ran75], dans lequel les processus redémarrent à leur point de départ. Afin d'éliminer cet inconvénient, les méthodes à sauvegarde *coordonnée* des points de reprise (*coordinated checkpointing*) synchronisent les processus lors de leur sauvegarde de points de reprise, de manière à s'assurer dès la sauvegarde des points de reprise que l'état de redémarrage est cohérent. On parle aussi de méthodes *pessimistes*. Les méthodes les plus simples stoppent l'exécution des processus pendant la phase de sauvegarde d'état. Dans une troisième classe de méthode, dite *guidée par les communications* (*communication-induced*), le principe est de forcer la mémorisation d'un point de reprise lors de chaque communication entre processus. Enfin, les méthodes à base de *journalisation* (*log-based*) consistent à journaliser les événements (en particulier les envois de messages) et à les rejouer lors de la reprise après un arrêt.

Une problématique et des méthodes de sauvegarde d'état similaires se retrouvent dans les systèmes à base de MVP, qui nous intéressent tout spécialement ici, et dans

lesquels in-fine les échanges de données entre processus sont effectués par envois de messages. Toutefois, quelques différences entre les deux types de systèmes méritent d'être mentionnées (voir [MP97] pour une synthèse) :

- l'état global dans les systèmes à MVP doit être enrichi avec les données de gestion de la MVP (e.g. répertoires, qui sont les structures de données permettant de localiser les copies des pages) ;
- la cohérence de l'état global du système peut être définie à un niveau plus abstrait qu'au niveau des communications par messages servant à la mise en œuvre du modèle de cohérence, ce qui introduit moins de dépendances entre mémorisations de points de reprise qu'en travaillant au niveau des envois de messages nécessaires au maintien de la cohérence. Il est par exemple possible d'effectuer des actions destinées à maintenir la cohérence de l'état global uniquement lors des défauts de page plutôt que lors de tous les envois de messages.

Que les processus communiquent par messages ou par mémoire virtuelle partagée, le type de méthode de sauvegarde d'état utilisé (coordonnée, indépendante, guidée par les communications, à base de journalisation) a une implication sur les instants auxquels un surcoût en temps est introduit (lors de la sauvegarde d'état ou/et lors des communications). Les choix que nous avons effectués pour que le surcoût soit adapté aux applications parallèles de longue durée sont présentés ci-dessous.

2.2.2 Mémorisation de l'état d'une application

Comme nous l'avons mentionné dans le paragraphe précédent, les contraintes de performance associées au domaine d'applications visé nous ont orienté vers une méthode de recouvrement par *sauvegarde/restauration* de l'état des processus. De plus, nous estimons nécessaire, pour le concepteur d'applications parallèles, de garder la *maîtrise* du coût associé à la sauvegarde et aussi la restauration de l'état de l'application. Cette volonté nous a amené à éliminer les méthodes de sauvegarde guidées par les communications et à base de journalisation parce qu'elles effectuent des opérations - et donc introduisent un surcoût - lors de chaque communication entre processus. Les instants auxquels les processus communiquent dans les systèmes à MVP sont potentiellement nombreux et surtout difficiles à anticiper. Les méthodes à base de sauvegarde indépendante des points de reprise ont été également écartées, non seulement parce qu'elles nécessitent de mémoriser les dépendances entre points de reprise lors de chaque communication, mais également parce que le volume de calcul perdu lors d'une reprise peut être grand et non borné (au pire, à cause de l'effet domino, tous les processus sont redémarrés à leur début). Pour ces raisons, notre proposition s'appuie sur une sauvegarde *coordonnée* des points de reprise.

Plus précisément, nous avons proposé une méthode pragmatique pour la sauvegarde des points de reprise. Elle consiste à intégrer la sauvegarde des points de reprise dans les *barrières de synchronisation*, qui permettent d'établir un point de rendez-vous entre tous les processus de l'application (fonction *svm_Sync* dans l'interface d'accès). La fonction résultante, nommée *svm_SyncCheck* intègre à la fois une synchronisation par rendez-vous et la mémorisation de l'état de l'application.

De plus, de manière à éviter de sauvegarder le contenu de la pile d'exécution, ce qui serait une tâche difficile techniquement et consommatrice en temps d'exécution à cause du caractère hétérogène de l'environnement, nous imposons que les appels à *svm_SyncCheck* soient réalisés dans le corps de l'application (fonction *main*). Par ailleurs, les seules données sauvegardées sont le contenu des régions partagées.

Le déroulement (simplifié) de la sauvegarde d'un point de reprise pour un processus de l'application est le suivant :

1. Rendez-vous avec les autres processus de l'application au point de synchronisation.
2. Transmission de l'ensemble des pages dont le processus courant est *propriétaire* (à chaque page est associé un propriétaire unique) à un processus *serveur de capture* qui écrit les pages sur disque, et attente d'un acquittement de la part du serveur une fois l'écriture sur disque effectuée. Une procédure est mise en place pour assurer la validation *atomique* des pages sur le disque. Les pages sont sauvegardées dans un format indépendant machine (voir paragraphe 2.4 pour plus de détails).
3. Synchronisation avec les autres processus de l'application par rendez-vous, pour qu'aucun d'eux ne reprenne son exécution avant la fin de la sauvegarde de l'état de l'application.

À chaque point de reprise est associé un identificateur entier correspondant au numéro de la barrière de synchronisation à laquelle l'état de l'application a été sauvegardé. Cet identificateur est utilisé lors de la reprise après défaillance pour savoir à quel point de l'exécution le programme doit être redémarré.

Les intérêts de cette méthode sont les suivants :

- *Maîtrise de coûts.* Puisque l'état de l'application est sauvegardé lors des barrières de synchronisation, aucun message n'est en transit pendant la phase de sauvegarde d'état. Ainsi, aucune action particulière n'est à réaliser lors de l'échange de messages entre processus. Le surcoût en temps d'exécution dû au support des arrêts de machines se produit donc *exclusivement* lors de la sauvegarde et la restauration de l'état de l'application. Le coût lié à la gestion des arrêts de machines peut être maîtrisé en choisissant l'intervalle entre sauvegardes de point de reprise.
- *Sauvegarde/restauration d'état hétérogène.* Il est possible de sauvegarder l'état de l'application pour la redémarrer sur une architecture de type différent. En effet, toutes les informations sauvegardées le sont dans un format qui est indépendant machine : le contenu des régions partagées, le point d'avancement courant dans le programme (qui est tout simplement le numéro de barrière dans le programme). Cette possibilité de sauvegarde/restauration en milieu hétérogène est particulièrement intéressante sur réseaux de stations de travail, qui sont fortement hétérogènes au niveau matériel et logiciel. De plus, comme nous le voyons dans le paragraphe 2.3, ce mécanisme de sauvegarde/restauration d'état peut être utilisé pour mettre en place un mécanisme d'équilibrage de charge en milieu hétérogène.

2.2.3 Optimisation de la mémorisation des points de reprise

Afin de diminuer le temps de sauvegarde des points de reprise, nous avons mis en place un ensemble d'optimisations : sauvegardes *incrémentales* et *asynchrones* des points de reprise, et mémorisation des points de reprise en *mémoire vive* [KCG+95]. Les deux premières optimisations, développées dans [CMP95], ont été intégrées au support d'exécution Stardust ; une analyse de leur performance est donnée dans le paragraphe 2.5.2.

2.2.3.1 Sauvegarde incrémentale des points de reprise

Dans la méthode de sauvegarde présentée dans le paragraphe précédent, toutes les pages des régions partagées sont sauvegardées sur disque, même si elles n'ont pas été modifiées depuis le dernier point de reprise. La sauvegarde *incrémentale* consiste à ne sauvegarder que les pages ayant été modifiées depuis la dernière sauvegarde d'état. Toutefois, la mise en place d'un tel mécanisme n'est pas directe car les systèmes d'exploitation utilisés dans les expérimentations ne permettent pas de connaître le bit de modification (*dirty bit*) de chaque page mis à jour par le processeur. Deux réalisations de la sauvegarde incrémentale ont donc été effectuées :

- *Sauvegarde incrémentale exacte*, dans laquelle le bit de modification d'une page est obtenu (de manière exacte) en exploitant les capacités du noyau à détecter les violations de protection sur les pages (toute page est initialement protégée contre l'écriture pour détecter ses modifications).
- *Sauvegarde incrémentale approchée*, dans laquelle les pages modifiées depuis le dernier point de reprise sont identifiées de manière approchée (cette méthode identifie un sur-ensemble des pages effectivement modifiées). Contrairement à la sauvegarde incrémentale exacte, aucun support du noyau n'est nécessaire pour identifier les pages modifiées.

2.2.3.2 Sauvegarde asynchrone des points de reprise

Dans la méthode de sauvegarde présentée dans le paragraphe précédent, un processus n'est redémarré qu'après avoir reçu un acquittement de la mémorisation du point de reprise sur disque. Nous avons mis en place une sauvegarde asynchrone, dans laquelle le processus qui sauvegarde son état reprend son exécution sans attendre d'acquittement. Deux variantes ont été mises en place (voir figure 2.3):

- *Sauvegarde asynchrone simple* (repère a. sur la figure), pour laquelle le contenu des pages est transféré dans un message (pouvant être envoyé en plusieurs fragments selon le protocole de communications utilisé). L'application continue à s'exécuter pendant l'acheminement du message vers le serveur de capture et pendant l'écriture sur disque.
- *Sauvegarde asynchrone avec recopie sur écriture* (repère b. sur la figure), pour laquelle le contenu des pages n'est pas transféré immédiatement. Un système de transmission des messages utilisant le mécanisme de recopie sur écriture (*copy-on-write*) fourni par le système d'exploitation est utilisé pour effectuer

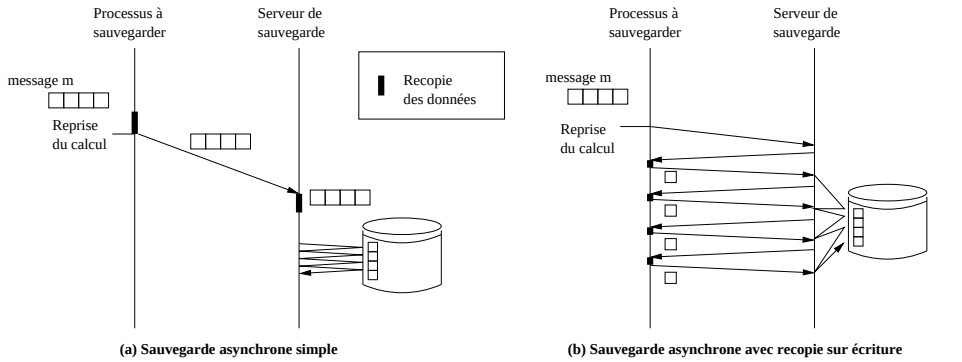


FIG. 2.3 – Optimisations de la sauvegarde de points de reprise

en parallèle la recopie du contenu des pages et l'exécution du processus. Lors de la transmission du message, les pages devant être transmises dans le message sont protégées contre l'écriture, jusqu'à ce que le récepteur du message tente de les lire. Si l'émetteur du message tente de modifier une page, celle-ci est dupliquée avant d'autoriser l'émetteur à poursuivre son exécution.

2.2.3.3 Sauvegarde des points de reprise en mémoire vive

En complément des méthodes de sauvegarde incrémentale et asynchrone des points de reprise, nous avons participé à l'élaboration d'une méthode d'optimisation consistant à sauvegarder les points de reprise dans la *mémoire vive* d'une autre machine indépendante vis à vis des défaillances. Les temps de sauvegarde des points de reprise sont alors moins importants qu'avec une sauvegarde sur disque étant donnés d'une part la latence plus faible des accès mémoire par rapport aux disques et d'autre part la réduction du goulot d'étranglement causé par le réseau lors des accès simultanés au disque par les calculateurs. En revanche, par construction, cette méthode ne tolère simultanément qu'un arrêt de machine. Une description de cette méthode, ainsi qu'une analyse détaillée de ses performances pourront être trouvés dans [KCG+95].

2.3 Méthode de diminution des temps de réponse

Ce paragraphe est consacré à la description de la méthode d'équilibrage de charge que nous avons mise en place dans le support d'exécution Stardust afin de réduire le temps d'exécution des applications parallèles. Cette méthode s'appuie sur la sauvegarde de l'état des applications ayant été présentée dans le paragraphe précédent. Après avoir présenté les principales composantes des méthode d'équilibrage de charge, nous développons notre proposition, ses performances étant étudiées par la suite (§ 2.5.3).

2.3.1 Problématique

À un instant donné, les charges des calculateurs servant à l'exécution des applications parallèles peuvent être très différentes. De plus, comme les machines, notamment les stations de travail, ne sont pas dédiées à l'exécution des applications parallèles, on peut observer des fluctuations de la charge des machines. Ces deux aspects nous ont amenés à mettre en place dans notre support d'exécution une méthode d'*équilibrage de charge* (*load balancing*), permettant de déplacer (*migrer*) les processus d'une machine chargée à une machine moins chargée, et ainsi diminuer le temps d'exécution de l'application par rapport à une exécution sans migration.

Étant donné que les stations de travail sont utilisées à la fois par un usager privilégié (*propriétaire* de la station de travail) et pour exécuter des applications parallèles, nous imposons comme contrainte que le propriétaire de la station puisse contrôler l'accueil d'applications parallèles, comme par exemple en interdire l'accueil pendant ses heures de présence.

Par ailleurs, notre volonté est de disposer pour l'exécution des applications parallèles de toute la puissance de calcul disponible dans le parc de machines, quel que soit leur type (simple station de travail ou machine massivement parallèle). Ceci pose le problème de prise en compte de l'hétérogénéité des machines. En particulier, il est souhaitable de pouvoir déplacer des processus d'une machine à une autre d'un type différent (migration *hétérogène*). De plus, pour tenir compte des fluctuations de charge des machines, il est souhaitable de pouvoir changer dynamiquement le nombre de processus de l'application parallèle, de manière par exemple à augmenter son nombre de processus quand un ensemble de machines peu chargées se libère.

La méthode d'équilibrage de charge que nous avons conçue repose sur le mécanisme de sauvegarde de l'état des applications présenté dans le paragraphe 2.2. Ce même mécanisme est utilisé pour équilibrer les charges des calculateurs et tolérer les arrêts de machines.

Les méthodes d'équilibrage de charge sont classiquement décrites en terme de trois composants [BSS91, Zho88] :

- La *politique d'information* définit la nature et la quantité d'informations nécessaires à l'algorithme d'équilibrage de charge, ainsi que la manière dont cette information est disséminée sur les différentes machines. La politique d'information repose sur l'existence d'un indicateur de charge qui symbolise la charge de

chaque machine.
- La *politique de transfert* définit les conditions locales qui rendent un processus éligible pour un placement initial sur une machine ou une migration (déplacement d'une machine à une autre).
- La *politique de localisation* est responsable du choix de la machine cible pour un placement initial ou une migration.

Les paragraphes qui suivent développent notre proposition au travers de ces trois composants. Notre méthode d'équilibrage de charge se distingue des méthodes existantes parce qu'elle permet la migration d'applications en *milieu hétérogène*, et ce sans modification ni du compilateur, ni du système d'exploitation, et parce qu'elle permet de changer dynamiquement la structure de l'application (son nombre de processus). Plus de détails sur cette méthode d'équilibrage de charge peuvent être trouvés dans les documents [CP96b, CP97].

2.3.2 Politique d'information

La politique d'information que nous avons mise en place repose sur la définition d'un indicateur de charge commun à tous les types de systèmes (concrètement, une valeur entière), la mise à jour des informations de charge étant effectuée périodiquement.

De manière à limiter le nombre de messages échangés pour disséminer la charge de chacune des machines, la propagation des indicateurs de charge tire parti de la structure hiérarchique du support d'exécution, composée d'un niveau de logiciel propre à chaque type d'architecture et d'un niveau global (voir § 2.4.2). Un *gestionnaire de charge* par type d'architecture est chargé de récolter les charges des machines de cette architecture, et de les traduire en indicateur de charge commun. Un *coordinateur* est désigné parmi les gestionnaires de charge pour centraliser les indicateurs de charge de toutes les machines à partir des indicateurs de charge communiqués par les gestionnaires de charge. Le coordinateur est le seul à posséder les indicateurs de charge de l'ensemble des machines.

2.3.3 Politique de transfert

Le rôle de la politique de transfert est d'identifier les processus cibles d'un placement initial ou d'une migration. La politique de transfert peut raisonner soit individuellement au niveau de chaque processus, soit globalement, en prenant une décision commune pour tous les processus de l'application. La première approche, adoptée notamment dans le système Mosix [BGW93], déplace chaque processus indépendamment des autres. Comme les processus communiquent entre eux, il se pose le problème de reconfigurer le système de communication de manière à rendre transparente la migration d'un processus aux processus communiquant avec lui. Ce problème est résolu dans Mosix par la conservation sur la machine source d'une migration d'un *processus squelette*, dont le rôle est de rediriger les communications vers la machine cible de la migration. L'intérêt d'un tel mécanisme est de rendre transparente la migration

d'un processus aux processus avec lesquels il communique. Mais cette méthode ne tolère pas l'arrêt des machines accueillant un processus squelette. Pour cette raison, nous avons choisi d'appliquer la politique de transfert à l'ensemble des processus de l'application, qui sont placés et migrés conjointement, en utilisant le mécanisme de sauvegarde d'état présenté dans le paragraphe 2.2.

La politique de transfert repose sur la définition de deux seuils pour l'indicateur de charge de chaque machine :

- un *seuil haut* au dessus duquel la machine est considérée surchargée.
- un *seuil bas* au dessous duquel la machine est considérée sous-chargée (un indicateur de charge au dessous du seuil bas signifie que la machine accepte de nouvelles applications).

Cette politique à double seuil permet aux utilisateurs privilégiés de stations de travail de contrôler l'utilisation de leur station par des applications parallèles. Par exemple, un seuil bas très faible interdit une telle utilisation.

Afin de limiter le coût de la politique de transfert, celle-ci est activée uniquement à certains instants clé de l'exécution des programmes : (i) lors du démarrage et de la terminaison des programmes (ii) lors du passage au dessus du seuil haut, ou au dessous du seuil bas d'une des stations, ce passage étant testé à chaque remise à jour des indicateurs de charge. De plus, pour gérer le partage des machines entre applications en cas de surcharge globale du système, les applications sont classées par priorités assignées statiquement. Lors de l'activation de la politique de transfert, les applications sont examinées par ordre de priorité, de manière à ce qu'en cas de surcharge globale au système les applications les moins prioritaires soient suspendues temporairement.

2.3.4 Politique de localisation

La politique de localisation a pour rôle de choisir les machines sur lesquelles ces processus doivent être effectivement placés. Placer un ensemble de processus sur une architecture à mémoire distribuée dans le cadre général (temps de calcul et de communication arbitraires) est un problème NP-complet [PY90, CP91]. Il est donc irréaliste de dérouler un algorithme optimal de placement en-ligne. Ainsi, de nombreuses études ont été entreprises pour trouver des solutions approchées en des temps raisonnables (voir [Fol93] pour un état de l'art).

Le mécanisme de sauvegarde d'état que nous utilisons pour déplacer une application (voir § 2.2), par construction, permet de changer dynamiquement le nombre de processus de l'application. Lors de la migration d'une application, il est donc nécessaire de sélectionner le nombre de machines à utiliser, et ce nombre étant sélectionné, les identités des machines, ce qui rend encore plus complexe l'algorithmique de la politique de localisation. Pour ces raisons, nous avons opté pour un support du programmeur d'applications. Ce dernier fournit les topologies d'application souhaitées pour l'exécution de son application par ordre d'efficacité décroissante. Une *topologie* définit le nombre de processus de l'application, et pour chaque processus le ou les types de machines pouvant accueillir l'exécution du processus. Par exemple, un

utilisateur peut établir les priorités suivantes : (i) 32 processus placés sur les processeurs d'une machine massivement parallèle ; (ii) 16 processus s'exécutant sur ce même type d'architecture ; (iii) 16 processus s'exécutant pour moitié sur architecture massivement parallèle et sur station de travail. Ainsi, la politique de localisation peut être déroulée en temps polynomial. Lorsqu'une application est sélectionnée comme cible d'une migration, ses topologies sont examinées tour à tour jusqu'à ce que l'une d'entre elles soit sélectionnée pour l'exécution d'après les indicateurs de charge des machines la composant.

2.3.5 Autres travaux concernant l'équilibrage de charge

Outre les travaux décrits dans ce paragraphe, ayant été intégré dans le support d'exécution Stardust, nous avons participé à la mise en place d'une méthode d'équilibrage de charge dans le cadre d'un support d'exécution distribué pour langage concurrent à objets [BBI+94, BBI+95b, BBI+95a]. Les points forts de cette méthode se situent dans ses politiques d'information et de localisation. Sa politique d'information optimise la charge du réseau en n'envoyant à partir d'un calculateur C des messages de charge qu'à un sous-ensemble des calculateurs, ceux qui sont susceptibles d'envoyer des calculs sur C. L'originalité de la politique de localisation est de prendre en compte les particularités du cadre objet pour décider d'où placer ou déplacer un objet. Au lieu de prendre en compte uniquement la charge des calculateurs comme dans les méthodes traditionnelles, le placement est sélectionné d'après la charge, mais aussi les communications entre objets [BBI+95a], de manière à regrouper les objets communicants entre eux.

2.4 Intégration de composants existants

De manière à limiter le coût d'achat, de construction et de maintenance de notre support d'exécution, nous nous sommes efforcés d'intégrer dans le support d'exécution des composants existants, à la fois au niveau *matériel* (calculateurs et réseaux) et *logiciel* (systèmes d'exploitation, protocoles de communication, compilateurs). Nous énumérons ci-dessous les principaux problèmes posés par cette volonté d'intégration de composants existants, et nos contributions pour les résoudre.

2.4.1 Problématique

Le principal problème issu de l'intégration de composants existants, qu'ils soient matériels ou logiciels, est leur *hétérogénéité*.

Au niveau matériel, notre souci est d'utiliser le plus grand ensemble possible de machines (machines massivement parallèles et stations de travail généralistes) de manière à augmenter la puissance de calcul globale disponible. Ceci pose le problème de gestion de l'hétérogénéité des machines au niveau matériel :

- jeux d'instructions différents pour les différentes architectures selon le processeur utilisé ;
- représentations des données différentes selon l'architecture (taille et ordre des octets en mémoire, représentation des flottants) ;
- tailles de pages différentes utilisées pour la gestion de mémoire virtuelle ;
- réseaux d'interconnexion entre processeurs différents selon les architectures.

Ce problème d'hétérogénéité se pose également au niveau logiciel. En effet, les machines exécutent potentiellement des systèmes d'exploitation différents et disposent de protocoles d'accès différents pour accéder au réseau. Bien que des protocoles standard (e.g. TCP/IP) existent pour communiquer entre les différentes architectures, des protocoles propriétaires, souvent plus efficaces, peuvent également exister. De tels protocoles existent par exemple dans les machines parallèles où des bibliothèques de communication propriétaires sont mises en place pour exploiter au mieux des réseaux de communication à très haut débit.

Par ailleurs, notons que nous avons sélectionné une méthode de recouvrement d'erreurs par sauvegarde d'état/restauration pour supporter les arrêts de calculateurs (cf § 2.2). L'utilisation de ce type de méthode requiert le respect de l'hypothèse de *silence sur défaillances* [PBS+88] pour chacun des calculateurs : soit il produit des résultats corrects dans le domaine temporel et valué, soit il s'arrête de produire des résultats. Étant donné que les conséquences du non respect de cette hypothèse dans le cadre d'applications parallèles de longue durée ne sont pas catastrophiques, nous n'avons pas dans ce cadre applicatif, cherché à évaluer la couverture de cette hypothèse. En revanche, nous avons mis en place des méthodes et outils permettant d'évaluer la couverture du silence sur défaillances dans le cadre des applications temps-réel critiques (voir chapitre 3).

2.4.2 Contributions : prise en compte de l'hétérogénéité

Nous détaillons ici les moyens que nous avons mis en œuvre pour gérer l'hétérogénéité des composants au niveau logiciel et matériel (voir paragraphe 2.5 pour une évaluation de performances).

2.4.2.1 Gestion de l'hétérogénéité au niveau logiciel

Le support d'exécution est fourni sous la forme d'une bibliothèque liée avec le code de chaque processus de l'application, et s'exécutant en *mode utilisateur*. Pour assurer les meilleures performances possibles pour le support d'exécution, nous l'avons structuré de manière *hiérarchique* :

- Le niveau bas de la hiérarchie (*intra-architecture*) comprend pour chaque type d'architecture un support d'exécution *optimisé pour ce type d'architecture*. Par exemple, sur la machine parallèle Paragon utilisée dans le prototype, nous avons utilisé la bibliothèque de communications optimisée NX pour les transferts de pages entre nœuds de la machine. Tant qu'une page réside sur un nœud de ce type d'architecture, les transferts se font en utilisant les moyens les plus efficaces offerts par ce type d'architecture.
- Le niveau haut de la hiérarchie (*inter-architecture*) définit les coopérations entre les différentes architectures nécessaires à la mise en place du support d'exécution global. À ce niveau, nous avons défini un protocole de cohérence permettant de transférer les pages mémoire entre deux architectures de type différents. Ce protocole repose sur une notion de page *hétérogène*, unité de transfert entre architectures de types différents, et utilise un protocole de communication standardisé.

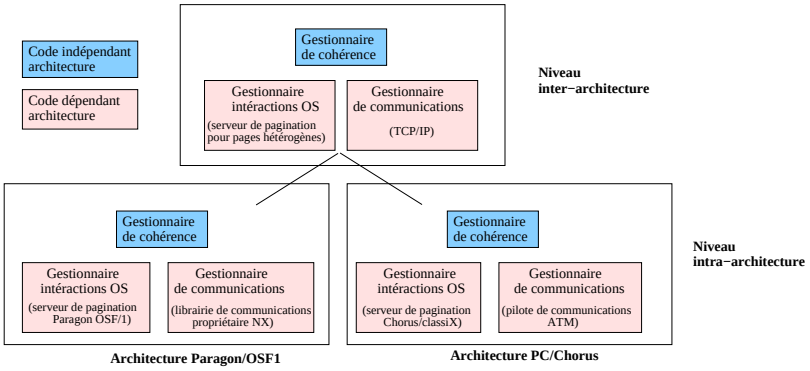


FIG. 2.4 – Structuration hiérarchique du support d'exécution

Cette décomposition hiérarchique est illustrée sur la figure 2.4 en prenant comme exemple les deux architectures cibles de Stardust : la machine parallèle Paragon et un réseau de PCs.

Pour chaque type d'architecture, ainsi qu'au niveau global à l'architecture, le support d'exécution est divisé en trois modules : le gestionnaire de cohérence, le gestionnaire de communications et le gestionnaire des interactions avec le système d'exploitation. Seuls les deux derniers modules mentionnés sont dépendants de l'architecture cible, ce qui facilite la portabilité du support d'exécution sur d'autres architectures.

2.4.2.2 Gestion de l'hétérogénéité au niveau matériel

Gérer l'hétérogénéité au niveau du matériel nécessite de connaître les types des données manipulées, afin de pouvoir les transformer lors du transfert d'une page d'un type d'architecture à un autre (par exemple, inverser l'ordre des octets dans les entiers lors du passage d'une architecture de type *big-endian* à une architecture de type *little-endian*). Ne voulant pas modifier le compilateur, notre solution [CP96a, CP97] consiste à associer un type aux données contenues dans une région partagée par l'intermédiaire d'une fonction de bibliothèque. Le typage est exprimé grâce à un langage dont la grammaire (sous forme BNF) est donnée ci-dessous.

Type	:	'{' (TypeBase | Type)+ '}'	
TypeBase	:	'C'	(caractère - char)
		'I'	(entier court - short)
		'L'	(entier long - long)
		'F'	(flottant simple précision - float)
		'D'	(flottant double précision - double)

Un type est de la forme *(TypeString,* N_1*,* N_2*, ... ,* N_n*)*, avec *TypeString* conforme à la grammaire donnée ci-dessus. N_i est le nombre d'éléments de la i^{eme} sous-séquence (identifiée par "{") de la chaîne *TypeString*. Par exemple, le type *("{C{D}}",10,20)* correspond à un tableau de 10 structures, chacune d'elles étant composée d'un caractère et de 20 valeurs flottantes double précision.

Ce typage des valeurs permet de transformer les données dans un format indépendant architecture quand une donnée est transférée d'une architecture à une autre. Le format de représentation utilisé est le format XDR (eXternal Data Representation) de Sun [mic90].

Gérer l'hétérogénéité matérielle nécessite également de supporter des tailles de pages différentes. Au sein d'une architecture donnée, grâce à la structuration hiérarchique de notre support d'exécution, c'est la taille de page associée à cette architecture qui est utilisée. Pour les transferts de pages inter-architectures, nous définissons la notion de *page hétérogène*, qui pour un ensemble d'architectures donné est la taille de page servant aux transferts entre les différentes architectures utilisées. La taille d'une page hétérogène est un multiple des tailles de pages utilisées dans les différentes

architectures ; elle est également un multiple de l'*élément de base*, qui est l'unité minimale de conversion des données due à la méthode de typage des données utilisée. Plus de détails sur la taille des pages hétérogènes peuvent être trouvés dans [Cab96].

Notons que de par la structure hiérarchique du support d'exécution, la transformation des données contenues dans les pages, qui est une opération consommatrice en temps d'exécution, n'est effectuée que quand cela est strictement nécessaire. Cette structure est l'élément clé permettant d'obtenir des accélérations importantes en milieu hétérogène.

2.5 Expérimentation : l'environnement Stardust

Les techniques de sauvegarde d'état et d'équilibrage de charge présentées dans les paragraphes 2.2 et 2.3 ont été implantées dans le support d'exécution Stardust [CP97, Cab96]. Nous présentons dans les paragraphes suivants sa structure interne et analysons ses performances. De plus amples détails sur les performances de Stardust pourront être trouvées dans [CP97, Cab96].

2.5.1 Architecture matérielle et logicielle de Stardust

Le support d'exécution Stardust s'exécute dans l'environnement matériel schématisé sur la figure 2.5, composé d'une machine massivement parallèle Intel Paragon et d'un réseau de stations de travail PC Pentium.

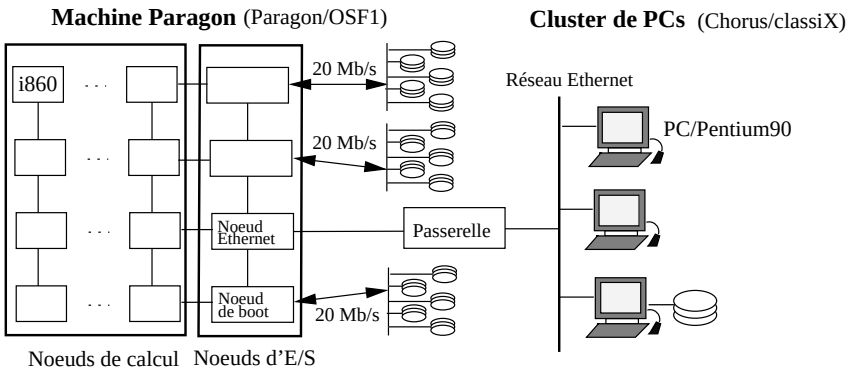


FIG. 2.5 – Architecture matérielle de Stardust

Pour les deux types d'architecture, un système d'exploitation de type *micro-noyau* a été utilisé (Paragon/OSF1 sur Paragon et Chorus/ClassiX sur les PCs).

Stardust apparaît comme une bibliothèque liée avec le code de chaque processus, et composée de trois modules (voir figure 2.4) :

- Un module de gestion de la cohérence est chargé du maintien de la cohérence forte sur les pages partagées, grâce à un protocole à invalidation sur écriture. Le code de ce module est le même quel que soit l'architecture sur laquelle s'exécute le processus (nœud Paragon ou machine PC).
- Un module de gestion des communications.
- Un module de gestion des interactions avec le système d'exploitation, chargé de détecter les défauts de page et les violations de protection sur les pages.

Le code des deux derniers modules est différent selon l'architecture cible pour le processus (nœud Paragon ou machine PC). Au niveau de la génération des exécu-

tables, un exécutable est généré par type d'architecture sur laquelle l'application est susceptible de s'exécuter.

À cette bibliothèque liée aux programmes utilisateurs s'ajoute un serveur de sauvegarde. Un tel serveur est chargé de la sauvegarde des points de reprise sur disque. Il en existe un exemplaire par type d'architecture.

2.5.2 Performances de la méthode de tolérance aux fautes

Nous examinons tout d'abord le temps de sauvegarde des points de reprise en comparant les différentes mises en œuvre de la sauvegarde d'état ayant été présentées dans le paragraphe 2.2.3. Le temps de sauvegarde a été mesuré en exécutant un ensemble d'applications [CMP95]. Nous présentons ici les résultats sur d'eux d'entre elles : Matmult, qui est une application de calcul matriciel (matrices carrées flottantes 512x512 - temps de calcul de 20 mn) et Mp3d, qui est est une application du benchmark Splash [SWG91] (temps de calcul de 25 mn) résolvant un problème de simulation de fluides. Pour les deux applications, un point de reprise est mémorisé toutes les 7 mn.

Le tableau 2.1 montre les performances de la mémorisation des points de reprise pour les différentes optimisations. Les temps sont donnés en pourcentage du temps d'exécution total des programmes.

Programme	**Base (synchrone)**	**Synchrone incrémental**		**Asynchrone incrémental**	
		exact	**approché**	**simple**	**copy-on-write**
	(%)	(%)	(%)	(%)	(%)
Mp3d	8.14	8.71	7.33	0.21	0.18
Matmult	11.04	6.52	6.42	0.11	0.04

TAB. 2.1 – Performances de la sauvegarde des points de reprise

Les chiffres figurant dans le tableau montrent l'intérêt des deux classes d'optimisations (sauvegarde incrémentale et asynchrone). La sauvegarde incrémentale est d'autant plus utile que l'application a un faible taux de modification des données, comme c'est le cas par exemple pour l'application de calcul matriciel.

La figure 2.6 montre, sur la même application de calcul matriciel que plus haut, le coût de la mémorisation des points de reprise (en terme de pourcentage du temps d'exécution) en fonction du nombre de points de reprise mémorisés pendant la durée de l'application. Ces mesures ont été effectuées sur la machine Paragon, avec une sauvegarde incrémentale approchée des points de reprise.

Quand un seul point de reprise est sauvegardé, toutes les pages des matrices sont sauvegardées, amenant à un coût de sauvegarde de 0.18% du temps d'exécution total. Quand plusieurs points de reprise sont sauvés, hormis pour la première sauvegarde de l'état de l'application, seules les pages modifiées sont sauvegardées. On observe alors que le coût de sauvegarde des points de reprise est quasiment linéaire par rapport au

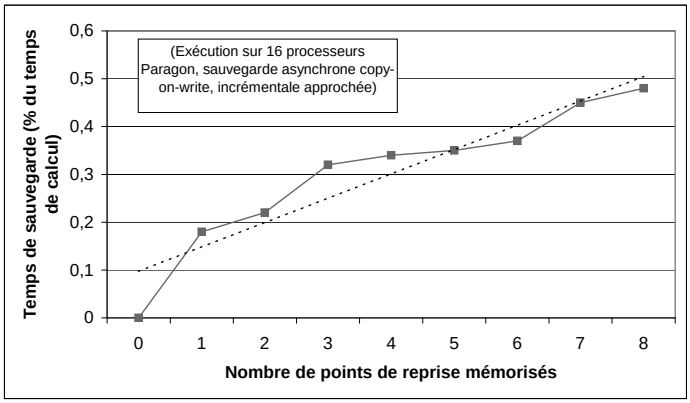


FIG. 2.6 – Coût de sauvegarde des points de reprise

nombre de points de reprise sauvegardés. De cette manière, l'utilisateur peut prédire pour chaque application le coût de mémorisation des points de reprise et choisir en conséquence la fréquence de leur mémorisation.

2.5.3 Performances de la méthode d'équilibrage de charge

Le tableau 2.2 montre sur deux applications (calcul matriciel et lancer de rayons) les gains pouvant être obtenus par la migration de l'application. Ces résultats ont été obtenus en déplaçant l'application pendant son exécution, la charge extérieure à l'application étant nulle. Pour la première ligne, on déplace l'application de 4 processeurs Paragon vers 4 processeurs PC (plus rapides) au milieu de son exécution (migration hétérogène). Pour la deuxième ligne, on étend le nombre de processeurs deux fois sur architecture PC (2, 4 et 8 processeurs), les déplacements de l'application étant équi-répartis sur la durée de l'exécution. Pour la dernière ligne, on passe d'une exécution sur 2 processeurs PCs à 8 processeurs PCs puis 64 processeurs (56 Paragon et 8 PCs).

	Sans Migration	**Migration**	**Gain** (%)
MatMultHetero4_4	65187 ms	61410 ms	5.8
MatMultHomo2_4_8	69640 ms	46910 ms	32.6
PovrayHetero2_8_64	3945 s	1125 s	71.5

TAB. 2.2 – Gains en performance dus à la technique d'équilibrage de charge

Les chiffres montrent que même quand une migration en milieu hétérogène est effectuée, le coût de la sauvegarde d'état (important en cadre hétérogène, étant don-

nés le volume de données à transférer et convertir – de l'ordre de la centaine de millisecondes) est masqué par les gains de performances obtenus.

La figure 2.7 donne le temps d'exécution d'une application de lancer de rayons du domaine public, nommée Povray [YW94] en utilisant Stardust, mais sans déplacer l'application au cours de son exécution. Sur la figure, la courbe notée *hétérogène* représente une configuration dans laquelle 8 processus s'exécutent sur PCs, les autres s'exécutant sur Paragon.

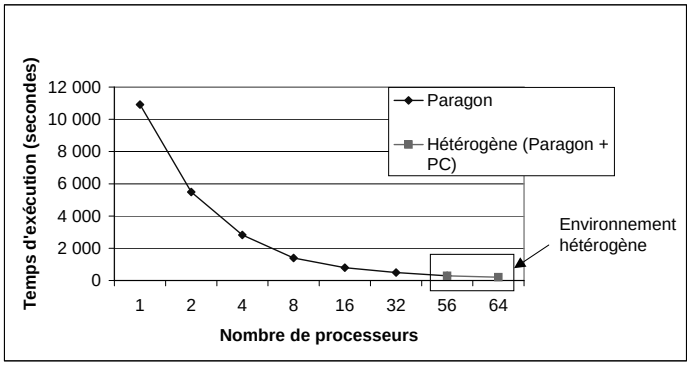


FIG. 2.7 – Temps d'exécution de Povray en univers hétérogène

Cette figure montre pour cette application qu'une bonne accélération peut être obtenue en milieu hétérogène (on n'observe pas de rupture significative dans la courbe lorsque l'on passe du milieu homogène au milieu hétérogène, à partir de 56 processeurs). Notons toutefois que cette application est bien adaptée à Stardust (et en général aux systèmes à base de MVP) car il y a peu de partage de données entre les processus, ce qui la rend particulièrement extensible à un grand nombre de calculateurs.

2.6 Discussion

Nous avons présenté dans ce chapitre nos travaux pour supporter l'exécution d'applications parallèles de longue durée sur réseaux de stations de travail. Nous avons ainsi conçu et réalisé un support d'exécution à base de mémoire virtuelle partagée qui permet d'allier bonnes accélérations pour les applications parallèles et tolérance aux fautes. Pour tolérer les fautes tout en maîtrisant l'impact de la tolérance aux fautes sur l'accélération, nous avons conçu un mécanisme de sauvegarde d'état de coût faible et surtout maîtrisable par le programmeur d'applications. Ce mécanisme de sauvegarde d'état est utilisé pour obtenir une bonne accélération des applications en présence de fluctuations des charges de travail des calculateurs accueillant les applications. De plus, le support d'exécution est conçu en intégrant le plus possible de composants existants (systèmes d'exploitation, compilateurs, piles de protocole), ce qui permet au support d'exécution d'être développé et porté facilement sur une nouvelle architecture, mais en revanche nécessite de gérer le caractère hétérogène des composants. Un prototype de support d'exécution, nommé Stardust, a été développé. Une analyse de ses performances sur plusieurs applications montre que pour des applications exhibant un taux de partage de données modéré, de bonnes accélérations peuvent être obtenues malgré le coût des transferts de données entre architectures hétérogènes.

Nos travaux se distinguent des travaux apparentés du domaine de plusieurs points de vue. Une première originalité est d'avoir fait cohabiter dans un support d'exécution à base de mémoire virtuelle partagée tolérance aux fautes et répartition de charge, ce qui rend notre support d'exécution plus riche en fonctionnalités que des supports d'exécution apparentés. Une autre originalité est la gestion de l'hétérogénéité. Si la gestion de l'hétérogénéité au niveau matériel est maintenant bien maîtrisée dans les environnements à base de communications par messages (e.g. PVM [GBD+94]), il n'en est pas de même dans les environnements à base de mémoire virtuelle partagée. À notre connaissance, seul Mermaid [ZSLW92] peut s'exécuter en univers hétérogène, mais dans un cadre limité (données de base de la même taille sur toutes les architectures, intervention dans le processus de compilation des applications).

Quelques travaux nous semblent toutefois nécessaires pour améliorer encore l'accélération lors de l'exécution d'applications parallèles en tolérant les fautes. Tout d'abord, il nous semble nécessaire de choisir un modèle de cohérence plus faible que le modèle de cohérence forte que nous avons sélectionné. En effet, avec un tel modèle de cohérence, et les protocoles à invalidation sur écritures qui y sont généralement associés, les performances des applications se détériorent lorsque de nombreux processus se partagent une même donnée de petite taille. De nombreuses études concernant les modèles et protocoles de cohérence ont été menées en parallèle à notre travail dans le domaine des modèles et protocoles de cohérence tout au long des années 1990 (par exemple [GLL+90, KCZ92, Gha95, ISL98]). La plupart de ces modèles imposent au programmeur des règles (par exemple, synchronisation imposée sur les variables partagées ou annotations des variables selon leur mode d'accès) et profitent de ces informations pour diminuer le nombre des échanges de messages entre

calculateurs. Cette perspective est plus une perspective d'amélioration du prototype Stardust qu'une réelle perspective de recherche, de nombreuses recherches sur le sujet ayant été menées dans les dix dernières années. Il s'agit uniquement dans ce cadre de sélectionner le modèle de cohérence qui permet d'établir le meilleur compromis entre gains en accélération et contraintes additionnelles imposées au programmeur d'applications.

Des travaux au niveau du système d'exploitation nous semblent également nécessaires pour pouvoir mettre en œuvre le plus efficacement possible des supports d'exécution pour applications parallèles, et plus particulièrement ceux fondés sur les mémoires virtuelles partagées. D'une part, les systèmes d'exploitation sur lesquels nous nous sommes appuyés n'offrent pas suffisamment de support pour accéder à leur état interne (e.g. fichiers ouverts, état des objets de synchronisation tels que les sémaphores, pages modifiées). Cette propriété rend difficile la sauvegarde de l'état des processus, ce qui nous a amenés dans le conception de notre support d'exécution à porter des restrictions sur les appels systèmes utilisables par le programmeur d'applications (par exemple ne pas utiliser directement les objets de synchronisation fournis par le système d'exploitation mais utiliser les outils de synchronisation Stardust). Des outils d'introspection au niveau du système d'exploitation, par exemple par utilisation de la réflexivité, permettraient de sauvegarder l'état des processus sans restreindre le spectre des appels systèmes utilisables par le programmeur.

D'autre part, les fonctionnalités de nombreux systèmes d'exploitation sont figées et ne permettent pas l'ajout de code additionnel pour les besoins particuliers d'une classe d'applications. Par exemple, le développement de mémoires virtuelles partagées, bien que possible, n'est réalisable pour de nombreux systèmes qu'en mode utilisateur, ce qui limite les gains en performances. Des travaux dans le domaine des systèmes d'exploitation permettant d'étendre (statiquement ou dynamiquement) leur fonctionnalités, ou simplement d'exécuter des processus en mode noyau, permettraient d'implanter des systèmes de mémoire virtuelle partagée de manière plus efficace, et ce sans avoir besoin de développer un système d'exploitation spécifique, ce qui ne nous semble pas souhaitable si l'on utilise des réseaux de stations de travail pour l'exécution des applications parallèles. Des travaux sur les noyaux spécialisables ont déjà été menés à ce jour (e.g. Spin [BSP+95] et Exokernel [EKO95]), mais n'ont pas encore eu de réel impact sur la conception des noyaux commerciaux.

Par ailleurs, nous avons constaté lors de l'évaluation de performances du prototype Stardust de grosses variations d'accélérations selon les applications. En particulier, des facteurs influençant fortement les performances sont les types d'accès aux données (taux de partage des données) et de l'organisation des données en mémoire (placement des données dans les pages dans les MVP paginées). Ainsi, il est délicat de prédire les gains en performances pouvant être obtenus. Ces facteurs étant relativement difficiles à anticiper, il nous semble nécessaire de développer des outils permettant d'analyser (en cours d'exécution ou de manière post-mortem) le comportement des applications et de résoudre leurs problèmes de performances. Dans le même ordre d'idée, lorsqu'on développe une application parallèle, choisir entre communication par messages et

communication par MVP ne semble toujours pas actuellement une tâche triviale. Des méthodes et outils permettant de guider le concepteur d'applications dans ce choix nous semblent nécessaires.

Notons que les méthodes présentées dans ce chapitre ne s'appuient pas sur des connaissances *a priori* des applications et de son environnement d'exécution, que ce soit au niveau du volume et des accès aux données qu'au niveau de la charge des calculateurs accueillant l'application. Par conséquent, bien que les méthodes que nous avons présentées visent et réussissent en *moyenne* à améliorer les temps d'exécution des applications tout en supportant les fautes, elles ne donnent pas de *garanties* sur les temps d'exécution. Si de telles garanties sont nécessaires de par le cadre applicatif (systèmes temps-réel strict), des méthodes et outils différents doivent être mis en place pour non seulement fournir le support pour *exécuter* les applications en présence de fautes, mais également *vérifier* que les temps d'exécution seront garantis. Nous nous intéressons à ces méthodes dans le chapitre qui suit.

Chapitre 3

Support d'exécution pour applications distribuées temps-réel critiques

3.1 Domaine d'application

Nous nous intéressons dans ce chapitre à l'exécution d'applications qui sont *critiques* (pour lesquelles le non respect des spécifications peut avoir des conséquences catastrophiques) et *temps-réel strict* (pour lesquelles le respect de contraintes temporelles fait partie des spécifications de l'application). On trouve de telles applications, par exemple, dans les domaines de la production d'énergie, de l'avionique, et maintenant dans l'automobile où l'informatique n'est plus réservée exclusivement à des fonctions de confort. Nous précisons dans le paragraphe 3.1.1 les besoins de cette classe d'application, puis donnons dans le paragraphe 3.1.2 nos travaux de recherche pour supporter l'exécution de telles applications, ces derniers étant développés dans la suite du chapitre.

Notons que nous nous intéressons à l'aspect *temps-réel* du système indépendamment du fait qu'il soit *embarqué*, ce qui apporterait une dimension supplémentaire au niveau des ressources disponibles (mémoire, énergie) qui sont le plus souvent limitées dans les systèmes embarqués. Nous revenons sur ce point en conclusion du document.

3.1.1 Expression des besoins

3.1.1.1 Garantie des temps de réponse (temps-réel strict)

De manière générale, on qualifie de *temps-réel* tout système dont le caractère correct ne dépend pas uniquement de la valeur du résultat produit par le système, mais également de l'instant auquel il est produit [SR88]. Les contraintes temporelles dans les systèmes temps-réel sont le plus souvent matérialisées par la notion d'*échéance*,

qui est la date au plus tard à laquelle un calcul doit être terminé. Les systèmes temps-réel sont usuellement séparés en deux classes selon les conséquences que peut avoir le non respect d'une échéance sur le système :

- Systèmes *temps-réel souple* (ou *mou*, en anglais *soft*). Pour ces systèmes, respecter les échéances est important, mais le dépassement d'échéances est tolérable si les échéances sont dépassées de manière occasionnelle.
- Systèmes *temps-réel strict* (ou *dur*, en anglais *hard*). Pour ces systèmes, le non respect d'une échéance constitue une défaillance de l'application. Dans le cadre d'un système critique, ceci signifie que le non respect d'une échéance peut avoir des conséquences catastrophiques, telles que la mise en danger de vies humaines ou des pertes financières importantes. Sauf mention contraire, nous nous plaçons par la suite dans le cadre des systèmes temps-réel strict.

Étant données les conséquences du non respect d'une échéance dans les systèmes temps-réel strict, il est nécessaire (ou tout du moins fortement recommandé) pour de tels systèmes de pouvoir *vérifier* avant leur exécution que toutes les échéances seront toujours respectées. Cette vérification est le rôle de l'*analyse d'ordonnançabilité* et requiert une connaissance du *pire* comportement temporel du système. En particulier, il est nécessaire de connaître les pires instants d'arrivée des tâches (*loi d'arrivée* des tâches, qui peut être périodique, sporadique ou apériodique) et les temps d'exécution des tâches (prises isolément) dans le pire des cas (en anglais *WCET*, pour *Worst-Case Execution Times*). De très nombreux travaux en analyse d'ordonnançabilité (théorie de l'ordonnancement) ont été menés depuis maintenant plus d'une trentaine d'année [SSNB95, But97]. Ils diffèrent notamment de par le *modèle de tâches* considéré (loi d'arrivée des tâches, modes de communication entre tâches), la *métrique* à minimiser (nombre d'échéances non respectées, retard cumulé), l'*instant* auquel est testé la faisabilité du système (hors-ligne, en-ligne) ou encore le *type d'informations généré* pour choisir en-ligne l'ordre d'exécution des tâches (plans, priorités).

Notons que vérifier qu'un système (au sens large) vérifie toutes ses contraintes temporelles nécessite de connaître non seulement le pire comportement temporel de l'application, mais également celui du support d'exécution et du matériel.

3.1.1.2 Tolérance aux fautes

Étant données les conséquences catastrophiques en cas de non respect des spécifications, les applications temps-réel critiques doivent tolérer les fautes, notamment les fautes d'origine physique. De plus, de par le contexte temps-réel strict, les mécanismes de tolérance aux fautes doivent également être compatibles avec les contraintes de temps-réel strict. En autres termes, les contraintes temporelles des applications doivent toujours être respectées, y compris en présence de fautes. La présence de contraintes de temps-réel strict influence de manière forte la conception d'un support d'exécution distribué, car toutes ses activités doivent impérativement avoir des temps de réponse bornés, de borne connue et compatible avec les échéances des applications, et ce même en présence de fautes.

3.1.2 Démarche et travaux de recherche

Dans un contexte temps-réel critique, concilier temps-réel et tolérance aux fautes nécessite l'introduction de méthodes de tolérance aux fautes (détection d'erreurs, recouvrement d'erreurs, traitement de fautes) *compatibles* avec la présence d'échéances strictes.

Cette compatibilité signifie d'une part que les méthodes utilisées pour la tolérance aux fautes (notamment le recouvrement d'erreurs) doivent introduire un coût en temps d'exécution suffisamment faible pour que les échéances des tâches puissent être respectées. Par exemple, les méthodes de sauvegarde/restauration d'état dans une mémoire stable mise en œuvre à partir de disques ne sont pas compatibles avec des échéances courtes à cause des latences disque. Ceci nous a amenés à sélectionner des méthodes de recouvrement d'erreurs à base de *duplication de tâches*.

D'autre part, quel que soit l'ordre de grandeur du coût des méthodes de tolérance aux fautes, il est nécessaire dans un contexte temps-réel strict de *vérifier* que toutes les échéances seront respectées lors de l'exécution. Ceci nous à amenés à introduire des méthodes faisant en sorte que les mécanismes de tolérance aux fautes soient naturellement pris en compte dans l'analyse d'ordonnançabilité du système. Ainsi, nos travaux de recherche, initialement concentrés sur les supports d'exécution, de par la prise en compte de contraintes de temps-réel strict, se sont élargis aux environnements de programmation et plus précisément à la vérification de contraintes temporelles. De cette manière, nos travaux de recherche ont non seulement porté sur la définition de mécanismes à inclure dans le support d'exécution pour que ce dernier tolère les fautes (détection d'erreurs, communications fiables et gestion de groupes) mais aussi sur l'analyse d'ordonnançabilité du système incluant des mécanismes de tolérance aux fautes. Ces travaux sur la tolérance aux fautes en contexte temps-réel strict ont notamment fait l'objet de la thèse de Pascal Chevochot [Che99]. Ils sont développés dans le paragraphe 3.2. Soulignons ici que nos travaux dans le domaine de la tolérance aux fautes ont été conçus pour des applications devant fournir un service de *qualité non dégradée* en présence de fautes, permettant comme nous le verrons par la suite d'automatiser le processus de fiabilisation des applications.

L'utilisation de méthodes d'analyse d'ordonnançabilité, que ce soit pour des systèmes tolérants aux fautes ou non, requiert la connaissance des temps d'exécutions au pire cas de programmes (en langue anglaise, *WCET*, pour *Worst-Case Execution Times*). Alors que l'analyse d'ordonnançabilité est un des champs de recherche traditionnel dans le domaine des systèmes temps-réel, l'obtention des temps d'exécution au pire cas n'a attiré l'attention de la communauté de la recherche en temps-réel qu'il y a environ dix ans, et constitue à l'heure actuelle un thème de recherche très actif. Aussi, nous avons mené une étude consistant à utiliser une méthode d'*analyse statique de programmes* pour obtenir leurs pires temps d'exécution sur une architecture donnée. Une méthode d'analyse a été définie, et une modélisation de l'architecture cible a été intégrée à la méthode d'analyse de manière à éviter un pessimisme excessif. La méthode d'analyse a été expérimentée sur différents logiciels dont le code d'un noyau temps-réel. Cette étude, ayant fait l'objet de la thèse d'Antoine Colin [Col01],

est décrite dans le paragraphe 3.3.

Une de nos préoccupations pour la construction d'un support d'exécution pour applications temps-réel critiques a été de diminuer les coûts de développement par intégration de composants logiciels existants (noyau de système temps-réel notamment). Cette volonté nous a amené à étudier l'impact de cette intégration de composants existants sur la construction de supports d'exécution pour applications temps-réel critiques. En particulier, nous avons mis en place des méthodes permettant de caractériser des composants existants, tant d'un point de vue comportement en présence de fautes que d'un point de vue comportement temporel. De plus, le fait d'intégrer des composants existants peut impliquer une connaissance imparfaite de leurs caractéristiques temporelles (loi d'arrivée et temps d'exécution au pire cas des tâches notamment). Ceci nécessite de concevoir des algorithmes d'ordonnancement qui permettent de s'adapter à des caractéristiques temporelles incertaines. La conception de tels algorithmes fait actuellement l'objet de la thèse de David Decotigny. Les travaux que nous menons dans cet objectif d'intégration de composants existants sont décrits dans le paragraphe 3.4.

Tous les travaux de recherche décrits dans ce chapitre ont été validés par la construction de logiciels. Ainsi, nous avons construit un environnement intégré permettant de développer, analyser le comportement temporel, fiabiliser et exécuter des applications temps-réel critiques. Une analyse des performances de chaque composant de cet environnement, en particulier son outil d'analyse de temps d'exécution au pire cas et son support d'exécution, a été effectuée. Nous présentons cet environnement et l'analyse de ses performances dans le paragraphe 3.5, avant de faire un bilan de nos travaux dans le domaine des systèmes temps-réel critiques.

3.2 Méthode de tolérance aux fautes

Ce paragraphe est consacré à la description de mécanismes de tolérance aux fautes physiques (temporaires ou permanentes), dans un cadre temps-réel strict. Contrairement aux mécanismes ayant été présentés dans le chapitre 2, une contrainte forte ici est de devoir, à cause des contraintes de temps-réel strict, être capable de vérifier que le comportement temporel du système est correct. Contrairement aux applications sans contrainte de temps-réel strict, pour lesquelles les mécanismes de tolérance aux fautes se doivent d'être le plus efficaces possibles, sans besoin de borner leur pire temps d'exécution, ici il est nécessaire de connaître de manière sûre (jamais trop optimiste) le comportement temporel des mécanismes de tolérance aux fautes, de manière à les intégrer dans l'analyse d'ordonnançabilité du système.

Les mécanismes de tolérance aux fautes, notamment les mécanismes de détection et de recouvrement d'erreurs peuvent être implantés soit par *matériel*, soit par *logiciel*. À titre d'exemple, l'architecture FTMP [HSL78] utilise une redondance triple pour les modules (processeurs, mémoire, périphérique), ainsi que du matériel spécifique pour contrôler l'état de chaque module, ainsi qu'effectuer pour chaque module un vote sur les valeurs redondantes qui lui sont transmises. Ici, et contrairement aux approches utilisant du matériel dédié pour la tolérance aux fautes, les mécanismes de tolérance aux fautes que nous introduisons (détection, recouvrement d'erreurs) sont exclusivement implantés *par logiciel*.

De manière à pouvoir vérifier le comportement temporel des tâches du système, et ce même en présence de fautes, il est nécessaire d'utiliser des *modèles* du comportement des tâches et du support d'exécution. En particulier, il est nécessaire de connaître statiquement la structure interne des tâches, comme par exemple d'identifier statiquement les points de synchronisation et de communications. Les modèles de tâches et de fautes que nous avons utilisés sont présentés dans le paragraphe 3.2.1.

À partir de la structure des tâches, notre approche pour qu'elles tolèrent les fautes est d'introduire les mécanismes de tolérance aux fautes (recouvrement d'erreurs notamment) pendant la phase de conception de l'application, par *transformation* de la structure des tâches. Par exemple, pour mettre en place une méthode de recouvrement d'erreurs par *redondance active*, la structure de l'application est modifiée de manière à dupliquer le code de la tâches sur différents calculateurs et insérer des voteurs pour décider du résultat du calcul dupliqué. L'introduction des mécanismes de tolérance aux fautes dans la structure des tâches est décrite dans le paragraphe 3.2.2.

Enfin, nous nous intéressons dans le paragraphe 3.2.3 à analyser l'ordonnançabilité du système. Cette analyse considère à la fois les tâches applicatives et celles implantant le support d'exécution, et ce en présence de fautes.

Notre contribution ici ne réside pas dans l'introduction de nouveaux mécanismes de tolérance aux fautes, pour lesquels des solutions classiques sont réutilisées. C'est la méthode d'introduction de ces mécanismes, qui permet leur intégration aisée dans des analyses d'ordonnançabilité, qui en constitue l'originalité.

3.2.1 Modèles et hypothèses

3.2.1.1 Modèle de tâches

Le modèle de tâches adopté indique comment sont structurées les tâches, les synchronisations possibles entre elles, ainsi qu'un ensemble de caractéristiques qui doivent être connues hors-ligne afin de vérifier leur ordonnançabilité. Les points essentiels de ce modèle sont les suivants :

- *Structure des tâches.* Chaque tâche est modélisée en utilisant un *graphe orienté acyclique* (DAG). Un nœud correspond à un calcul sans synchronisation interne, et avec un temps d'exécution au pire cas (WCET) connu. Un arc correspond à une contrainte de précédence entre deux nœuds, le nœud destination ne pouvant être exécuté qu'après le nœud source. Une contrainte de précédence peut relier deux nœuds placés sur des calculateurs distincts (auquel cas la tâche est distribuée). La transmission de données (d'un message) peut être associée à une contrainte de précédence. Enfin, le placement de chaque nœud sur un calculateur est identifié statiquement. Un exemple de tâche est donné sur la partie gauche de la figure 3.1, page 49.
- *Attributs de tâches.* Un ensemble d'*attributs* concernant les tâches doivent être connus hors-ligne pour dérouler l'analyse d'ordonnançabilité :
  - des *attributs temporels* : échéance de terminaison, loi d'arrivée (périodique, sporadique, apériodique), temps d'exécution au pire cas (de tels temps sont typiquement fournis par une méthode d'analyse de WCET telle que celle qui est décrite dans le paragraphe 3.3) ;
  - des *attributs de synchronisation* qui permettent, entre autres, de définir des contraintes d'exclusion entre nœuds. Pour cela chaque nœud indique les ressources nécessaires à son calcul, et pour chacune, s'il souhaite l'utiliser en *mode exclusif* (pour la modifier) ou en *mode partagé* (uniquement pour la consulter). Chaque *ressource* peut aussi bien modéliser un dispositif matériel (capteur, actionneur) qu'une entité logicielle composée de données et de code pour agir sur ces données. Les données contenues dans les ressources sont les *seules* à persister après l'exécution d'une tâche.

Ce modèle de tâche à base de graphes orientés sans cycles est un modèle classique dans le domaine de l'analyse d'ordonnançabilité temps-réel, que les tâches soient centralisées ou distribuées [Mok83].

3.2.1.2 Modèle d'exécution en présence de fautes

Les fautes considérées dans notre travail sont des fautes d'origine physique, que celles-ci soient temporaires ou permanentes. Pour pouvoir raisonner sur le comportement temporel du système en présence de fautes, et en particulier pour analyser l'ordonnançabilité du système en présence de fautes, le comportement du support d'exécution est caractérisé par un ensemble de *propriétés*. Ces propriétés permettent de vérifier que le comportement du support d'exécution est correct, aussi bien d'un point de vue fonctionnel que temporel, et ce indépendamment de son implantation.

Bien entendu, il est alors nécessaire que le support d'exécution intègre les services permettant d'assurer ces propriétés. Nous laissons ce point en suspens pour l'instant, il est traité dans le paragraphe 3.4.1.

Nous supposons qu'en présence de fautes, le support d'exécution de chaque calculateur vérifie un ensemble de propriétés (détaillées dans [Che99]), qui peuvent être résumées informellement comme suit :

- Le support d'exécution s'exécutant sur chaque calculateur est à *silence sur défaillances* [PBS+88] : soit il produit des résultats corrects dans le domaine temporel et valué, soit il s'arrête définitivement de produire des résultats ;
- Le support d'exécution offre des communications multipoints fiables en temps borné (multicast fiable). Plus précisément, le support d'exécution, lors de l'envoi de messages, vérifie les propriétés de *ponctualité* (*timeliness* – diffusion d'un message en temps borné), la *validité* (*validity* – si un message est émis de C_1 à C_2, et C_1 et C_2 ne s'arrêtent pas pendant le déroulement du protocole, alors le message est nécessairement livré à C_2), la livraison selon l'ordre causal, et l'*accord* (*agreement* – si un message est adressé à deux destinataires non arrêtés C_1 et C_2, si C_1 le livre et que C_2 ne s'arrête pas pendant le déroulement du protocole, alors C_2 le livre également).
- Le support d'exécution d'un calculateur permet de détecter l'arrêt d'un autre calculateur en temps borné. Plus précisément, les propriétés suivantes sont respectées : *ponctualité* (*timeliness* – la détection de l'arrêt ou du démarrage d'un calculateur est effectuée en temps borné) ; *vivacité* (*vivacity* – l'arrêt ou le démarrage d'un calculateur est nécessairement détecté par les autres) et *intégrité* (*integrity* – pas de fausse détection de l'arrêt d'un calculateur).
- Les horloges locales de calculateurs sont synchronisées. L'écart maximal entre deux horloges locales quelconques est borné et connu et la vitesse de progression de chaque horloge est contenue dans un intervalle borné (propriétés d'*accord* – *agreement* et d'*exactitude* – *accuracy*).

Concernant les éléments matériels, les *actionneurs*, moyens de commande de l'environnement extérieur, sont supposés fiables. Les *capteurs*, permettant de surveiller l'environnement extérieur, peuvent être sujets à des défaillances par valeurs, mais pas à des défaillances temporelles.

Si l'on considère l'exécution des tâches structurées selon le modèle donné ci-dessus (§ 3.2.1.1), ces propriétés signifient que le support d'exécution est apte à valider *en temps borné* une contrainte de précédence entre nœuds du graphe, et ce que le calculateur source s'arrête ou non. Nous entendons par *valider* le fait que le nœud cible de la contrainte de précédence puisse commencer à s'exécuter, que le calculateur source soit opérationnel (on parle alors de validation correcte) ou arrêté (validation incorrecte).

3.2.2 Mécanismes de tolérance aux fautes

Nous proposons d'intégrer les différentes étapes de la tolérance aux fautes (détection d'erreurs, recouvrement d'erreurs) lors de la phase de conception des tâches du

système, par *transformation de la structure* des tâches le composant [Che99, CP00a]. Par souci de concision, nous nous concentrons ici sur la phase de recouvrement d'erreurs, pour laquelle nous donnons le principe de transformation des tâches pour l'introduction de recouvrement d'erreurs et une illustration sur une technique de recouvrement d'erreurs, la redondance active. Pour les autres étapes de la tolérance aux fautes, on se reportera à [Che99].

3.2.2.1 Principe général

L'introduction de mécanismes de recouvrement d'erreurs est réalisée par transformation du graphe définissant la structure des tâches, ainsi que ses attributs, et notamment ses attributs temporels. Pour récupérer une erreur, la structure d'une tâche est transformée de manière à :

- Insérer de la *redondance* au niveau des données et/ou des exécutions de tâches. Par exemple, dans le cas de la *redondance active* [CPR+92], on effectue une redondance d'une tâche ou d'une portion de tâche (exécution et données manipulées).
- Insérer des *blocs de base* de tolérance aux fautes, qui assurent que l'exécution en présence de fautes est équivalente à une exécution de la même tâche en absence de fautes. Des exemples de blocs de base sont par exemple : un *voteur*, établissant un consensus sur le résultat de l'exécution de tâches dupliquées (utilisé par les méthodes de redondance active), ou encore un bloc de *transfert* de l'état d'une ressource d'un calculateur à un autre (utilisé par les méthodes de redondance passive).

Ce principe de transformation des tâches a été utilisé pour intégrer dans la structure des tâches différentes techniques de recouvrement d'erreurs fondées sur la redondance de tâches : redondance *active*, *passive* et *semi-active*, ainsi qu'une technique de détection d'erreurs : la *redondance temporelle* [RSL95]. La redondance active, ou redondance n-modulaire [CPR+92, Sch90] consiste à exécuter en parallèle les mêmes calculs sur des calculateurs distincts et à voter sur les résultats des calculs dupliqués. La redondance semi-active [BHV+90] est similaire à la redondance active, excepté que les décisions communes à toutes les copies d'un calcul sont prises par un des calculateurs (appelé primaire), tandis qu'avec la redondance active, elles sont prises par un vote sur les résultats des calculs dupliqués. La redondance passive [Pow91] consiste à exécuter un calcul sur un calculateur appelé primaire, tandis que les autres mémorisent régulièrement son état d'exécution servant à redémarrer en cas d'arrêt du primaire.

Soulignons que les différentes méthodes de recouvrement d'erreurs que nous avons mises en place reposent toutes sur la duplication des *tâches* sur plusieurs calculateurs indépendants vis à vis des défaillances, plutôt que par la sauvegarde de leur état dans une *mémoire stable*, comme nous avons choisi de le faire dans le chapitre 2 pour les applications parallèles de longue durée. Ce choix a été motivé par le fait que les mémoires stables sont généralement mises en œuvre à l'aide de *disques*, ce qui rend la latence d'accès à la mémoire stable importante. Utiliser de telles mémoires stables

pour le recouvrement d'erreurs rendrait le comportement temporel des applications incompatible avec des échéances courtes, d'où notre choix d'opérer par duplication de tâches[1].

Remarquons également que la méthode que nous proposons pour l'introduction de mécanismes de recouvrement d'erreurs nécessite la présence d'une architecture redondante, composée de multiples calculateurs, capteurs et actionneurs. Cette redondance est naturellement présente dans le type d'architecture visé par nos travaux (architectures distribuées, composées d'un ensemble de calculateurs en réseau).

L'intérêt consistant à opérer par transformation de la structure de tâches est double. D'une part, il permet d'être sélectif, dans le sens où il est possible de demander uniquement la duplication des parties jugées critiques (i.e. devant tolérer les fautes). D'autre part, les mécanismes de tolérance aux fautes (copies des états et des exécutions) sont banalisées vis à vis de l'analyse d'ordonnançabilité. Ainsi, la présence de mécanismes de tolérance aux fautes est rendue quasiment transparente à l'outil d'analyse d'ordonnançabilité, comme nous le montrons dans le paragraphe 3.2.3.

3.2.2.2 Exemple de transformation : redondance active

La figure 3.1 montre sur un exemple de tâche comment est transformée la structure d'une tâche pour y intégrer des mécanismes de recouvrement d'erreurs, ici par une méthode de redondance active. La tâche est composée de quatre nœuds : le nœud *nr* lit une donnée par l'intermédiaire d'un capteur, le nœud *nw* transmet le résultat de la tâche à un actionneur, le nœud *nc* effectue des traitements critiques sur les données obtenues de *nr* et le nœud *nnc* effectue des traitements non critiques sur ces mêmes données.

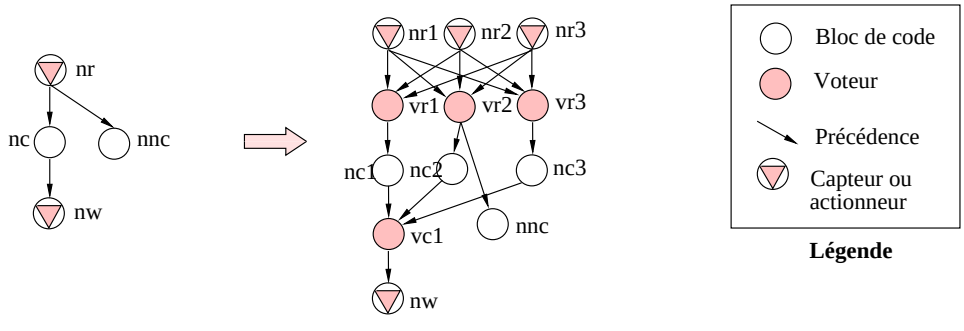


FIG. 3.1 – Exemple d'application de la redondance active

[1] Notons qu'il existe également des dispositifs de mémoire stable *rapides*, mis en œuvre à partir de bancs de mémoire vive (par exemple [BBM88]). Nous ne nous sommes pas appuyés sur ce type de dispositif pour des raisons économiques, nous poussant à utiliser des composants matériels COTS.

Sur l'exemple, les nœuds nr et nc sont dupliqués en utilisant la redondance active avec un degré de redondance de 3 sur les calculateurs C_1, C_2 et C_3. En ce qui concerne le nœud de calcul nc, les copies de ce nœud ($nr1$, $nr2$ et $nr3$ sur la figure), ainsi que les ressources utilisées par ces copies, sont créées automatiquement. Pour le nœud capteur nr, le concepteur du système doit s'assurer de la présence de plusieurs capteurs (nœuds $nr1$, $nr2$ et $nr3$ sur la figure) et en informer l'outil en charge de la transformation des tâches.

Afin d'établir un consensus sur le résultat de calculs dupliqués, l'outil en charge de la transformation des tâches insère des blocs de base de tolérance aux fautes, nommés *voteurs*. Un voteur (par exemple $vr1$, $vr2$, $vr3$ ou $vc1$ sur la figure) calcule, à partir d'un ensemble de valeurs provenant de calculs dupliqués, une valeur unique. Un voteur correspond à la fonction de convergence d'un protocole de consensus. Puisque les calculateurs vérifient l'hypothèse de silence sur défaillance, un voteur très simple peut être utilisé dans le cas où uniquement des calculs sont dupliqués. Ce voteur ($vc1$ sur la figure), nommé *voteur par défaut* retourne n'importe quelle valeur reçue en entrée. Quand des capteurs, qui sont sujets à des défaillances par valeurs, sont dupliqués, le voteur par défaut ne peut plus être utilisé. Le concepteur du système peut alors choisir une fonction de vote parmi une bibliothèques de voteurs (par exemple fonction de sélection de la valeur médiane ou de la moyenne).

3.2.2.3 Déterminisme

Le *déterminisme* est un objectif clé pour tous les systèmes fondés sur la redondance de tâches. L'objectif du déterminisme des copies d'un calcul est de s'assurer que les copies non défaillantes restent toujours convergentes.

Dans notre contexte, deux principales sources d'indéterminisme sont possibles. La première, existant uniquement dans le cadre de la redondance active, est due aux copies de capteurs (nœuds $nr1$ à $nr3$ sur la figure) qui ne fournissent pas nécessairement les mêmes valeurs selon leur emplacement, même si elles sont lues au même instant. Cette source d'indéterminisme est réglée par l'utilisation d'une fonction de convergence adaptée dans les voteurs (choix par le concepteur de l'application d'une fonction de vote de la bibliothèque).

La seconde source d'indéterminisme (commune à toutes les techniques de redondance) est l'exécution dans un ordre différent, selon les calculateurs, des copies de plusieurs tâches/nœuds. À titre d'illustration, avec la redondance active, si a_1 et b_1 (copies des nœuds a et b sur le calculateur C_1) ne s'exécutent pas dans le même ordre que a_2 et b_2 (copies des nœuds a et b sur le calculateur C_2), et si a et b modifient une même ressource, alors l'état de cette ressource diverge selon les calculateurs. Un tel scénario peut se produire par exemple si les exécutions de a et b sont lancées par des événements arrivant dans un ordre différent sur les deux calculateurs (messages, interruptions matérielles).

Pour éviter cette deuxième source d'indéterminisme, une solution est de choisir en ligne (grâce à l'utilisation d'un protocole de synchronisation distribué) un ordre d'exécution identique pour les copies de nœuds effectuant des opérations conflic-

tuelles. Toutefois cet ordre peut être contraire à celui choisi hors-ligne par le test d'ordonnançabilité, et donc mener à des dépassements d'échéances non prévus. C'est pourquoi, nous avons choisi de laisser à l'analyse d'ordonnançabilité la responsabilité de calculer un ordre d'exécution qui ne soit pas source d'indéterminisme. Pour cela, l'outil en charge de la duplication des tâches génère, en plus du graphe des tâches dupliquées, des contraintes d'ordre qui doivent être respectées par tout algorithme d'ordonnancement en charge des tâches dupliquées. Ainsi, un algorithme d'ordonnancement peut être indépendant (excepté concernant ces contraintes d'ordre) de la technique de redondance utilisée pour les tâches (voir § 3.2.3.2).

3.2.3 Analyse d'ordonnançabilité

Nous nous consacrons ici à la vérification du comportement temporel du système en présence de défaillances matérielles. Notre contribution dans ce cadre est de prendre en compte *tous* les coûts logiciels, et en particulier ceux provenant du support d'exécution, qui utilise des tâches pour assurer les propriétés d'exécution mentionnées dans le paragraphe 3.2.1.2.

Nous décomposons l'analyse d'ordonnançabilité en deux sous-problèmes : l'analyse des temps de réponse au pire cas du support d'exécution (§ 3.2.3.1) et l'intégration de ces coûts dans l'analyse d'ordonnançabilité du système global, en considérant à la fois l'application et le support d'exécution (§ 3.2.3.2).

3.2.3.1 Analyse au pire cas des coûts système

Les propriétés d'exécution brièvement mentionnées dans le paragraphe 3.2.1.2 ne sont pas dans le cadre général directement assurées par le matériel, et doivent alors être assurées par logiciel. Ainsi, le support d'exécution doit intégrer un ensemble de protocoles (diffusion multipoint, synchronisation d'horloges) pour assurer ces propriétés. Nous avons conçu et implanté un tel support d'exécution, nommé par la suite HADES pour *Highly Available Distributed Embedded System*. Nous nous concentrons ici sur les points permettant d'analyser le temps de réponse au pire cas des tâches le constituant. Une description plus complète du support d'exécution est donnée dans le paragraphe 3.5.

De manière à ne pas imposer d'analyse d'ordonnancement particulière aux tâches applicatives, notre environnement d'exécution intègre un système d'ordonnancement à *deux niveaux* :

- les tâches du support d'exécution (synchronisation d'horloges, diffusion multipoint) sont gérées par un ordonnanceur par défaut, qui est *préemptif à priorités fixes* ;
- les tâches de l'application sont gérées par un ordonnanceur différent de l'ordonnanceur par défaut, qui peut être changé pour les besoins d'une classe d'application particulière. Concrètement, l'ordonnanceur associé aux tâches applicatives apparaît sous la forme d'un *service d'ordonnancement*. Un tel service gère l'ordre d'exécution d'un ensemble de tâches, associées à une ou plusieurs

priorités, moins importantes que celles du support d'exécution. Ce principe de conception en deux niveaux est emprunté aux micro-noyaux de première génération, dans lesquels une partie des fonctionnalités pouvait être déportée sous la forme de serveurs (par exemple, serveurs de pagination) permettant de spécialiser le système aux besoins d'un domaine d'application particulier.

Connaître le comportement temporel du support d'exécution nécessite de connaître les *temps de réponse (au pire cas)* de toutes les tâches le constituant, c'est-à-dire l'intervalle (au pire) entre l'arrivée de la tâche dans le système et la fin de son exécution, et ce malgré les préemptions par des tâches plus prioritaires. Afin de déterminer les temps de réponse des tâches de notre support d'exécution, nous avons adapté les travaux de Tindell et Clark [TBW94], consacrés à l'analyse de temps de réponse de tâches dans les systèmes à priorités fixes, aux spécificités de notre support d'exécution. Nous donnons ci-dessous les grandes lignes du travail d'analyse qui a été effectué, plus de détails pouvant êtres trouvés dans le document [CP00a].

Notons bien ici la différence entre *temps d'exécution au pire cas* et *temps de réponse au pire cas*. La première notion se réfère au temps d'exécution d'une tâche considérée de manière *isolée*. La deuxième notion, pouvant être utilisée pour vérifier que toutes les échéances seront respectées, représente le pire délai entre arrivée de la tâche et fin de son exécution, en tenant compte *des autres tâches* présentes dans le système. Le temps de réponse au pire cas d'une tâche est nécessairement supérieur à son temps d'exécution au pire cas, car il comptabilise le temps pendant lequel la tâche ne s'exécute pas (n'est pas encore démarrée, est en attente ou encore interrompue par une tâche plus prioritaire).

Analyse de temps de réponse de Tindell et Clark. Dans les travaux de Tindell et Clark, une tâche est une séquence infinie d'exécutions d'un programme séquentiel. À chaque tâche i est associée une priorité pr_i, et on note $hp(i)$ l'ensemble des tâches plus prioritaires que i. Les priorités sont uniques, et à tout instant, c'est la tâche la plus prioritaire qui s'exécute (ordonnancement préemptif à priorité fixe). À chacune de ses arrivées, une tâche i s'exécute au plus C_i unités de temps (C_i est le temps d'exécution au pire cas de la tâche considérée de manière isolée). Pour l'analyse de temps de réponse, C_i est un paramètre d'entrée connu pour chaque tâche i ; déterminer les C_i nécessite l'utilisation de méthodes et outils, comme ceux que nous décrivons dans le paragraphe 3.3. Les tâches sont sporadiques (l'intervalle moyen entre deux arrivées successives d'une tâche i est supérieur ou égal à P_i). Chaque tâche i peut se synchroniser avec d'autres (par exemple en utilisant des sémaphores) et être bloquée par des tâches moins prioritaires au plus pendant B_i unités de temps. On peut établir une telle borne sur les temps de blocage en utilisant un protocole à héritage de priorité lors de l'acquisition des sémaphores, comme par exemple celui décrit dans [SRL90]. Enfin, il peut exister un délai entre l'arrivée d'une tâche (date à laquelle une tâche est présente dans le système d'un point de vue logique) et sa libération (date à laquelle la tâche est insérée dans les files d'exécution) ; ce délai peut se produire par exemple quand l'ordonnanceur remplit ses files d'exécution

périodiquement. Un tel délai est nommé *gigue* et est noté J_i.

Appelons période i-busy (voir figure 3.2) la durée pendant laquelle le processeur est continuellement occupé par des tâches de priorité $pr(i)$ ou supérieure.

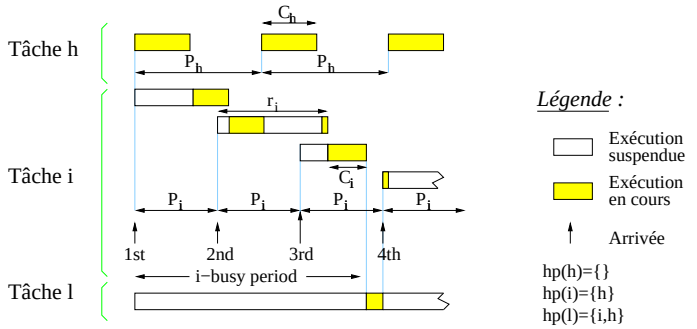


FIG. 3.2 – Période i-busy

En supposant que pour toute tâche j on connaisse P_j, B_j, C_j et J_j, le temps de réponse d'une tâche i, nommé r_i peut être calculé de manière itérative grâce aux formules suivantes :

$$r_i = J_i + \underset{q=0,1,2,\ldots}{Max} \{w_i(q) - q.P_i\} \tag{3.1}$$

$$w_i(q) = (q+1)C_i + B_i + \sum_{\forall j \in hp(i)} \lceil \frac{J_j + w_i(q)}{P_j} \rceil . C_j \tag{3.2}$$

$w_i(q)$ est la longueur d'une période i-busy débutant par la libération de i et contenant $q+1$ exécutions de i. Dans la formule 3.2, $\lceil \frac{J_j+w_i(q)}{P_j} \rceil$ est le nombre de libérations de la tâche j pendant $w_i(q)$. Dans la formule 3.1, $w_i(q) - qP_i$ est le temps entre la $q+1$ème libération de la tâche i et sa terminaison. Dans la figure 3.2, la seconde exécution de i est la plus retardée, r_i est alors égal au temps de réponse de cette deuxième exécution. Il est montré dans [TBW94] que ces équations produisent toujours un résultat quand le taux d'utilisation du processeur est inférieur à 100%. De plus, dans l'équation 3.1, la séquence de valeurs pour q est finie, et l'itération peut stopper quand $w_i(q) \leq (q+1)P_i$.

Adaptation de l'analyse de temps de réponse. Le support d'exécution a été conçu en utilisant (autant que possible) des tâches périodiques [CP00a]. Par exemple, le réseau a été géré sans utiliser de demandes d'interruption de la carte réseau, mais plutôt par échantillonnage périodique du tampon de la carte réseau. Malgré

cette règle de conception, les temps de réponse des tâches du support d'exécution ne peuvent pas être directement obtenus par l'analyse de temps de réponse introduite ci-dessus, et ceci pour plusieurs raisons :

- Les tâches du support d'exécution n'ont pas toutes des priorités différentes.
- Les tâches du support d'exécution s'appellent entre elles, de manière synchrone (bloquante) ou asynchrone (non bloquante). Par exemple, périodiquement, une tâche déclenchée par l'interruption du sablier (timer) met à jour l'horloge locale et appelle toutes les tâches périodiques prêtes à s'exécuter à cet instant. Cette notion d'appel n'apparaît pas dans le modèle de tâches des travaux de Tindell et Clark.
- Certains traitements de bas niveau du support d'exécution sont non interruptibles, comme par exemple le code de prise en compte des interruptions.
- Enfin, certaines tâches du support d'exécution ne sont pas sporadiques (par exemple, les tâches appelées à chaque livraison de message).

Le premier point a nécessité une très légère modification de l'équation 3.2. Les autres points ont influencé l'évaluation des paramètres d'entrée C_i, B_i, P_i et J_i de l'analyse de temps de réponse, et ce de la manière suivante :

- *Évaluation de* C_i. Pour chaque tâche exécutée via un appel synchrone, son temps d'exécution au pire cas C_i est incorporé dans celui de la tâche appelante ;
- *Évaluation de* B_i. Des inversions de priorité peuvent se produire à cause de la présence d'appels synchrones et de traitements interruptibles. Toutefois, elles sont bornées et il est possible d'identifier la borne sur le temps de blocage (voir [CP00a]).
- *Évaluation de* P_i. Certaines tâches du support d'exécution ne sont pas sporadiques. Toutefois, pour toutes ces tâches, il a été possible, via un artifice, de transformer chaque tâche non sporadique par n tâches sporadiques, avec n le nombre de calculateurs dans le système, et ce en ayant connaissance des lois d'arrivée des tâches applicatives.
- *Évaluation de* J_i. La gigue lors de l'arrivée d'une tâche a été évaluée dans notre contexte. Elle est égale (i) dans le cas d'une tâche lancée par interruption, à la latence maximale de traitement d'une interruption ; (ii) dans le cas d'une tâche lancée par un appel asynchrone, au temps de réponse de la tâche appelante.

Modulo ces modifications, l'analyse de temps de réponse de Tindell et Clark a pu être appliquée sur chaque calculateur, permettant pour chaque tâche i du support d'exécution de borner son temps de réponse r_i.

De manière similaire, ce type d'analyse a été appliqué pour déterminer le temps maximum d'attente des messages dans les files d'attente du réseau. Il est alors possible d'obtenir le temps de transmission d'un message de bout en bout (coûts logiciels inclus). In fine, cette analyse conduit à l'estimation (i) du temps maximum nécessaire au lancement d'une tâche distribuée δ_{start} ; (ii) du temps maximum nécessaire à la validation d'une contrainte de précédence distante δ_{vd}. Plus de détails sur les étapes permettant d'évaluer ces paramètres peuvent être trouvés dans [CP00a].

À partir des caractéristiques temporelles de tâches du support d'exécution, il est possible de calculer l'impact du support d'exécution sur le temps d'exécution des tâches de l'application. La formule ci-dessous donne le temps d'exécution d'une activité utilisateur e (qui par construction est moins prioritaire que les tâches du support d'exécution), coûts du support d'exécution inclus.

$$\overline{C}_e = C_e + C_{valid} + \sum_{\forall j} \lceil \frac{J_j + \overline{C}_e}{P_j} \rceil . C_j + \sum_{1 \leq x \leq Nb_{int}} \lceil \frac{\overline{C}_e}{P_{int_x}} \rceil . wcet_{int} \qquad (3.3)$$

Dans cette formule, C_e est le temps d'exécution d'un nœud dans le graphe décrivant le programme utilisateur, coûts système exclus, et $\overline{C}_e$ est l'équivalent coûts systèmes compris. C_{valid} représente le coût de la tâche exécutée à chaque validation de contrainte de précédence. La première somme représente le temps d'interruption du calcul e par des tâches du support d'exécution. La deuxième somme représente l'équivalent pour des traitements d'interruptions (P_{int_x} est la pseudo-période d'arrivée de l'interruption de niveau x et $wcet_{int}$ le coût de traitement associé).

Intégration transparente des coûts du support d'exécution dans un test d'ordonnançabilité. À ce point, si l'on considère un ensemble de tâches utilisateur décrites sous la forme de graphes (voir § 3.2.1) il est possible de faire abstraction de la structure interne du support d'exécution. Décider de l'ordonnançabilité du système global revient alors à résoudre un problème d'ordonnançabilité d'un graphe avec les contraintes suivantes :

- Exclusion entre nœuds du graphe.
- Précédences entre nœuds. Un nœud peut débuter son exécution au plus tôt δ unités de temps après la fin du nœud le précédent dans le graphe (δ prend les valeurs δ_{vd} ou δ_{vl} selon que les nœuds source et destination sont plaçés sur le même calculateur ou non).

Dans le graphe, le temps d'exécution au pire cas de chaque nœud e (C_e) est remplacé par son analogue intégrant les activités asynchrones du support d'exécution ($\overline{C}_e$, calculé dans l'équation 3.3).

3.2.3.2 Analyse d'ordonnançabilité globale (application plus support d'exécution)

Afin de montrer qu'il est possible d'analyser l'ordonnançabilité d'un système en présence de fautes et en prenant en compte les coûts du support d'exécution, nous avons développé un test d'ordonnançabilité, par extension de l'analyse d'ordonnançabilité de Xu et Parnas, 1990 [XP90]. Nous décrivons ici l'analyse d'ordonnançabilité résultante, cette dernière étant également développée dans [CP99]. Notons que l'intérêt principal de notre démarche pour intégrer des mécanismes de recouvrement d'erreurs est de pouvoir analyser l'ordonnançabilité du système de manière *indépendante* de la méthode de recouvrement d'erreurs utilisée. Par exemple, il est possible d'ana-

lyser l'ordonnançabilité d'un système dans lequel les tâches utilisent des méthodes de recouvrement d'erreurs différentes (par exemple redondance active, passive, voire pas de méthode de recouvrement d'erreurs).

Principe de l'algorithme. L'analyse d'ordonnançabilité que nous avons mise en place est de type *hors-ligne* (l'ordonnançabilité du système est vérifiée avant exécution), *statique* (l'ordre d'exécution des tâches est déterminé avant exécution), *préemptif* (une tâche peut être interrompue par une autre tâche) et *distribué* (deux nœuds d'une même tâche peuvent être placés sur des calculateurs distincts).

Le modèle de tâches utilisé par l'algorithme est le modèle de tâches présenté dans le paragraphe 3.2.1, réduit à l'utilisation de tâches périodiques. L'algorithme opère sur un intervalle de temps de taille bornée T égale au PPCM (Plus Petit Commun Multiple) entre les périodes de toutes les tâches du système. Chaque tâche de période P s'exécute donc $\frac{T}{P}$ fois dans cet intervalle. Nous nommons par la suite *segment* chaque exécution d'un nœud d'une tâche pendant cet intervalle (à chaque nœud d'une tâche de période P est associé $\frac{T}{P}$ segments).

Dans l'algorithme, l'ordonnançabilité du système est testée de manière conjointe avec la construction d'un plan définissant l'ordre dans lequel seront exécutés les segments. L'algorithme procède en deux temps :

- Calcul d'une solution au problème initial. Dans cette solution, les contraintes d'exclusion et de précédence sont vérifiées, mais il est possible que les échéances de certaines tâches soient dépassées.
- Parcours de l'arbre de recherche des solutions selon une stratégie de *branch-and-bound*. À chaque nœud de l'arbre de recherche est associé un problème d'ordonnançement à résoudre et un plan d'exécution P. Un ensemble de problèmes *dérivés* est créé à partir de ce problème en ajoutant des contraintes (préemption, précédence), dont le but est de diminuer le retard du segment ayant le plus grand retard dans P. L'arbre de recherche est parcouru jusqu'à ce qu'une solution dans laquelle toutes les échéances des tâches sont respectées (il est également possible de dérouler intégralement l'arbre de recherche pour trouver la meilleure solution possible).

Nous utilisons les notations suivantes dans la description de l'algorithme :

- *Dates d'arrivée, temps d'exécution, placement et échéance d'un segment.* On note *A(s)* la date d'arrivée d'un segment s, et *C(s)* son temps d'exécution (valeur $\overline{C}_s$ calculée dans le paragraphe 3.2.3.1). Le calculateur sur lequel le segment est placé est nommé *calc(s)* et son échéance (relative au début de l'intervalle de temps considéré – PPCM des périodes) est notée *D(s)*.
- *Précédence entre nœuds.* $a\ PC\ b$ indique que le segment b ne peut commencer à s'exécuter que δ_{vd} (resp. δ_{vl}) unités de temps après que le segment a ait terminé son exécution. La constante δ_{vd} (resp. δ_{vl}) correspond au temps nécessaire au support d'exécution pour valider une contrainte de précédence à distance (resp. en local). L'évaluation de ces deux constantes a été présentée dans le paragraphe 3.2.3.1.

- *Exclusion entre segments.* A *EX* b avec A un ensemble de segments indique que l'exécution du segment b est exclusive avec une exécution quelconque d'un élément de l'ensemble A.
- *Préemption entre segments.* a *PR* b indique que le segment b n'est pas exécuté si le segment a est prêt à s'exécuter (on dit que l'exécution de a préempte celle de b).
- *Contrainte d'exécution selon un ordre série.* Pour assurer le déterminisme des calculs dupliqués (voir § 3.2.2), nous avons choisi d'imposer que les copies de certains segments s'exécutent dans le même ordre sur plusieurs calculateurs. Cette contrainte est matérialisée par la relation IO (*Identical Order*), ces contraintes étant générées par l'outil en charge de la duplication des tâches. Soient a_x et b_x deux segments d'un même exemplaire de la tâche x, et soient a_y et b_y deux segments d'un même exemplaire de la tâche y. De plus, supposons que $calc(a_x) = calc(a_y)$ et que $calc(b_x) = calc(b_y)$. Notons $begin(s)$ la date de début d'exécution du segment s. $(a_x, a_y) IO (b_x, b_y)$ ssi

$$begin(a_x) < begin(a_y) \wedge begin(b_x) < begin(b_y)$$

ou

$$begin(a_x) > begin(a_y) \wedge begin(b_x) > begin(b_y)$$

Calcul d'une solution initiale. Soit le problème initial, défini comme :
- un ensemble de segments, caractérisés par leur placement (calc), leur temps d'exécution (C), leur date d'arrivée (A) et leur échéance (D)
- un ensemble de contraintes d'exécution associées à ces segments : exclusions (EX), précédences (PC), préemptions (PR), contraintes d'ordre d'exécution (IO).

La solution à ce problème, nommée *solution initiale* est un plan construit selon la stratégie EDF (*Earliest Deadline First*). Plus précisément, sur chaque calculateur p à une date t, on évalue l'ensemble $S(p,t)$ des segments éligibles et non préemptés par d'autres segments éligibles.

$$\begin{aligned} S(p,t) = \{ s \mid calc(s) = p \ \wedge \ Eligible(s,t) \ \wedge \\ \nexists b \mid calc(b) = p \ \wedge \ (Eligible(b,t) \wedge b \ PR \ s) \} \end{aligned}$$

Pour toute date t, on choisit d'exécuter sur le calculateur p le segment de $S(p,t)$ ayant la plus petite échéance (*EDF - Earliest Deadline First*). Dans la définition de $S(p,t)$, $Eligible(s,t)$ signifie que toutes les contraintes (exclusion, précédences, préemptions, ordre) sont respectées pour que s soit exécutable.

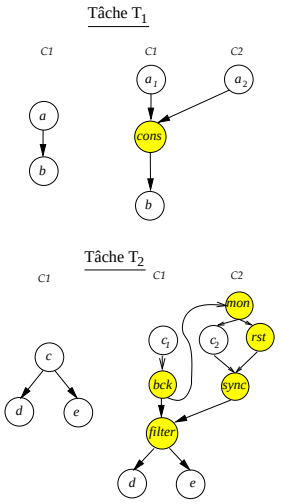
Recherche d'une solution satisfaisant toutes les échéances. La solution initiale telle qu'elle a été calculée dans le paragraphe précédent respecte les contraintes de précédence, d'exclusion, de préemption et d'ordre identique. Toutefois, dans le

plan qui est construit, toutes les échéances ne sont pas nécessairement respectées. À chaque nœud N de l'arbre de recherche est associé un problème d'ordonnancement à résoudre O_N (ensemble de segments et contraintes associées) et un plan d'exécution par calculateur p, noté P_{Np}. Un ensemble de problèmes *dérivés* est créé à partir de ce problème. Soit l le segment le plus en retard dans le plan P_{Np}. De nouveaux problèmes sont dérivés en ajoutant des contraintes au problème O_N :

- des contraintes de précédence $l\ PC\ s$,
- des contraintes de préemption $l\ PR\ s$.

Ces contraintes sont ajoutées de telle sorte qu'une solution à un problème dérivé soit toujours une solution au problème initial. Pour chaque nœud de l'arbre de recherche, on évalue une borne inférieure b pour le retard de ses nœuds fils (voir [CP99] pour plus de détails). On calcule un plan, selon la stratégie EDF présentée dans le paragraphe précédent pour le nœud ayant la valeur de b la plus petite. L'arbre de recherche est ainsi parcouru jusqu'à ce qu'une solution dans laquelle toutes les échéances des tâches sont respectées soit trouvée.

Exemple d'application de l'algorithme. La figure 3.3 illustre l'application de l'analyse d'ordonnançabilité présentée ci-dessus sur un petit jeu de tâches. La partie à l'extrême gauche de la figure montre les graphes de deux tâches périodiques (de périodes identiques) T_1 et T_2, avant introduction de recouvrement d'erreur.



Seg.	A	C	D	
a_1	0	10	40	$\{a_1\}$ EX c_1
a_2	0	10	40	$\{a_1\}$ EX bck
$cons$	0	1	66	$\{c_1, bck\}$ EX a_1
b	38	9	66	$\{a_2\}$ EX c_2
c_1	0	10	30	$\{c_2\}$ EX a_2
bck	0	2	69	$\{a_2\}$ EX rst
mon	0	1	69	$\{rst\}$ EX a_2
rst	0	2	69	$\{c_2\}$ EX rst
c_2	0	10	30	$\{rst\}$ EX c_2
$sync$	0	1	69	$\{d\}$ EX e
$filter$	0	1	69	$\{e\}$ EX d
d	45	9	69	(a_1, a_2) IO (bck, rst)
e	53	9	63	

Remarque : Les attributs temporels des blocs de base de tolérance aux fautes ($cons$, bck, ...) sont calculés par l'outil en charge de la duplication des tâches, et en fonction des attributs temporels des autres nœuds, excepté l'attribut C.

FIG. 3.3 – Exemple de problème initial

La partie gauche de la figure montre la structure de ces deux tâches après introduction de mécanismes de recouvrement d'erreur. Les blocs apparaissant en grisé sur la figure correspondent aux blocs de base de tolérance aux fautes. Concernant la tâche T_1, le segment a est dupliqué en utilisant la redondance active, sur les calculateurs 1

et 2. Le bloc *cons* est le voteur permettant d'établir un vote sur le résultat du calcul dupliqué. En ce qui concerne T_2, le segment c est dupliqué en utilisant la redondance passive (c_1 est la copie primaire, c_2 la copie secondaire exécutée uniquement en cas de défaillance de c_1). Le bloc noté *mon* détecte si la copie primaire est défaillante ou pas : si c'est le cas, la copie de sauvegarde c_2 est exécutée ; dans le cas contraire, on exécute le bloc *rst* qui met à jour l'état des ressources utilisées par le calcul.

Le reste de la figure (parties centrale et droite) montre le problème initial à résoudre (caractéristiques temporelles des segments et contraintes associées). Les contraintes d'exclusion proviennent du fait que a et c modifient une même ressource. Sur cet exemple, nous supposons que $\delta_{vd} = 2$ et que $\delta_{vl} = 0$.

La figure 3.4 déroule l'analyse d'ordonnançabilité pour les tâches T_1 et T_2. La partie haute de la figure présente la solution initiale, dans laquelle le segment e ne respecte pas son échéance. Deux problèmes dérivés sont construits, et leurs solutions respectives Sol_1 et Sol_2 sont données en bas de la figure. Sol_1 est obtenu en ajoutant une contrainte de précédence $e\ PC\ d$, tandis que Sol_2 est obtenu en ajoutant une contrainte de préemption $e\ PR\ b$. Dans la solution Sol_1, d ne respecte pas son échéance, mais en revanche dans Sol_2 tous les segments respectent la leur. La solution Sol_2 est donc retenue comme solution.

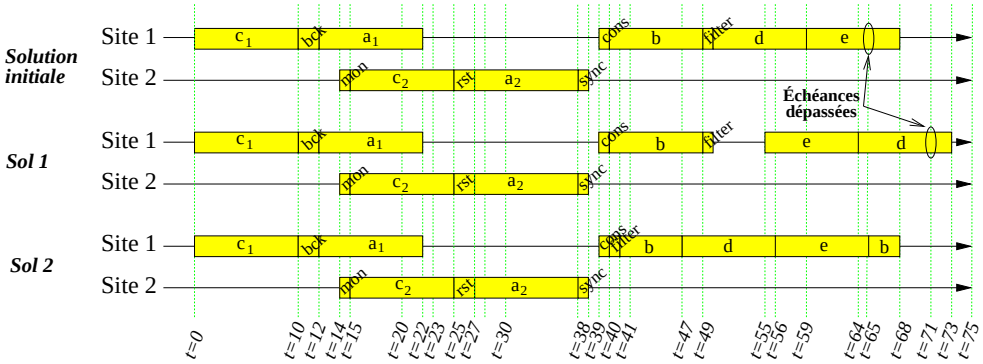


FIG. 3.4 – Exemple d'analyse d'ordonnançabilité tolérante aux fautes

3.3 Méthode de garantie des temps de réponse

3.3.1 Problématique

Pour tout système temps-réel strict, qu'il soit ou non tolérant aux fautes, la connaissance du temps d'exécution au pire cas des tâches est nécessaire à l'analyse d'ordonnançabilité du système. Par exemple, dans l'analyse de temps de réponse de Tindell et Clark, que nous avons introduite dans le paragraphe 3.2.3.1, page 52, le temps d'exécution au pire cas de chaque tâche i, noté C_i est nécessaire pour connaitre son pire temps de réponse r_i et pouvoir ainsi vérifier que toutes les échéances seront respectées.

Nous nous intéressons ici à obtenir les temps d'exécution au pire cas par *analyse statique du code source* des tâches, ce que nous nommons par la suite *analyse statique de WCET* [PB00]. Le rôle de l'analyse statique de WCET est, par analyse du code source d'un programme, de calculer une borne supérieure de son temps d'exécution sur une architecture matérielle donnée. Deux points sont importants à souligner dans une telle définition. D'une part, le temps évalué par une telle méthode est le temps que le processeur va utiliser pour *exécuter* le programme, en supposant une exécution *ininterrompue* du programme. En particulier, les calculs des temps d'attente dus à l'utilisation d'ordonnancement préemptifs, de routines de synchronisation bloquantes ou d'interruptions matérielles ne relèvent pas de l'analyse de WCET, et sont du ressort de l'analyse d'ordonnançabilité. D'autre part, le calcul de WCET dépend de l'architecture matérielle utilisée, et par conséquent un outil d'analyse de WCET doit intégrer des informations sur les caractéristiques temporelle l'architecture cible.

Comme le WCET calculé par analyse statique est utilisé pour vérifier que le comportement temporel du système est correct, le temps d'exécution calculé doit être *sûr*, c'est-à-dire qu'il ne doit jamais être sous-estimé. Mais le WCET calculé ne doit pas pour autant être une approximation trop pessimiste du temps d'exécution effectif du programme, car une surestimation trop importante du WCET d'un programme entraîne une surestimation des ressources matérielles jugées nécessaires à son exécution et donc une sous-utilisation de ces ressources à l'exécution.

Deux sous-problèmes doivent être traités pour obtenir le WCET d'un programme à partir de son code source :

- *Caractérisation des chemins d'exécution (analyse de haut niveau)* Il s'agit ici d'identifier les différents chemins d'exécution faisables (et infaisables) dans un programme à partir de son code source (c'est pourquoi on parle d'analyse de *haut niveau*).
  Le résultat de cette caractérisation est un ensemble d'informations sur les chemins d'exécution, identifiant par exemple les fonctions exécutées, le nombre d'itération des boucles, les dépendances entre constructions conditionnelles, etc. Ces informations peuvent être obtenues soit automatiquement par analyse du code source [EG97, HAM+99, EES+99], soit par l'intermédiaire d'annotations [Par93, FMW97, LM95].
- *Obtention du temps d'exécution au niveau matériel (analyse de bas niveau)* Il

s'agit ici d'obtenir la durée d'exécution de chaque instruction ou séquence d'instructions sur une architecture matérielle donnée. Cette analyse opère au niveau du code objet des programmes (analyse de *bas niveau*), mais peut également requérir des informations provenant du code source pour augmenter la précision de l'analyse.

La complexité de l'analyse de bas niveau dépend des caractéristiques du matériel considéré. Par exemple, si l'on considère une architecture dotée d'un cache d'instructions, la durée d'exécution de l'instruction dépend de l'absence ou la présence de l'instruction dans le cache, qui dépend du contexte d'exécution de l'instruction (contenu du cache au moment où l'instruction est exécutée). De nombreux travaux récents ont pris en compte la structure interne des processeurs (micro-architecture) pour déterminer de manière sûre mais le moins pessimiste possible les temps d'exécutions des programmes. Les principaux éléments architecturaux ayant été étudiés à ce jour sont les *caches d'instructions* [Mue00, LMW96, FMW97, HAM+99, SA00], *caches de données* [KMH96], les *pipelines* [EE99, HAM+99, SA00] et les *prédicteurs de branchement* [CP00b].

Les méthodes d'analyse statique de WCET peuvent être séparées en trois catégories : les méthodes à base de *chemins*, d'*arbres* et d'*énumération implicite des chemins*. Dans les méthodes à base de chemins [HAM+99, SA00], le WCET d'un programme est généré en calculant les temps d'exécution le long des différents chemins d'exécution possibles dans le programme. L'aspect distinguant cette classe de méthode des deux autres est que les chemins d'exécution y sont *explicitement* représentés. Dans les méthodes à base d'arbres [PS91, LBJ+94, CP00b], le WCET d'un programme est calculé par un parcours hiérarchique de bas en haut de l'arbre syntaxique du programme, le WCET d'un nœud de l'arbre étant utilisé pour calculer le WCET de son père. Enfin, dans les méthodes à base d'énumération implicite des chemins [LM95, PS97, OS97], les chemins d'exécution dans les programmes, ainsi que les temps d'exécution sont exprimés sous la forme d'un ensemble d'équations algébriques et/ou logiques. Le WCET du programme est alors calculé par résolution du système de contraintes généré. Il n'existe pas à ce jour (à notre connaissance) de comparaisons entre ces trois types de méthodes, que l'on prenne comme critère de comparaison la précision de l'analyse ou son coût en terme de temps d'analyse.

Nous avons proposé une méthode d'analyse à base d'arbres, qui permet d'obtenir des WCETs sûrs, sans être pour autant trop pessimistes. Notre contribution réside dans l'introduction de deux types de méthodes permettant de réduire le pessimisme de l'analyse : au niveau haut, un système d'annotations adapté aux boucles non rectangulaires (§ 3.3.3), et au niveau bas une modélisation de la micro-architecture (cache d'instructions, pipeline, prédiction de branchement - § 3.3.4). Un prototype d'outil d'analyse de WCET est présenté plus loin dans le document (§ 3.5.1), et ses performances analysées.

3.3.2 Principe général de l'analyse

Notre méthode d'analyse s'applique à du code source programmé dans le langage C [2]. Certaines restrictions au langage source, classiques dans le domaine de l'analyse statique de WCET, sont nécessaires à l'identification de tous les chemins d'exécution :

- *Interdiction des appels dynamiques de fonctions.* L'utilisation d'appels *dynamiques* de fonctions (appels réalisés via des pointeurs sur fonctions) est interdite ;
- *Interdiction des graphes de flot de contrôle non structurés.* Les branchements autres que ceux introduits par les structures de contrôle du langage que sont les boucles et conditionnelles sont interdits. Ceci interdit par exemple l'inclusion de code source assembleur contenant des instructions de branchement, ou encore la commande *goto* du langage C.
- *Établissement de bornes sur le nombre d'itérations des boucles.* Le nombre maximum d'itérations des boucles, doit être borné. L'établissement de telles bornes repose sur l'introduction d'annotations symboliques dans le programme source (voir § 3.3.3). Il en est de même pour le niveau maximum de récursivité, qui doit également être borné.

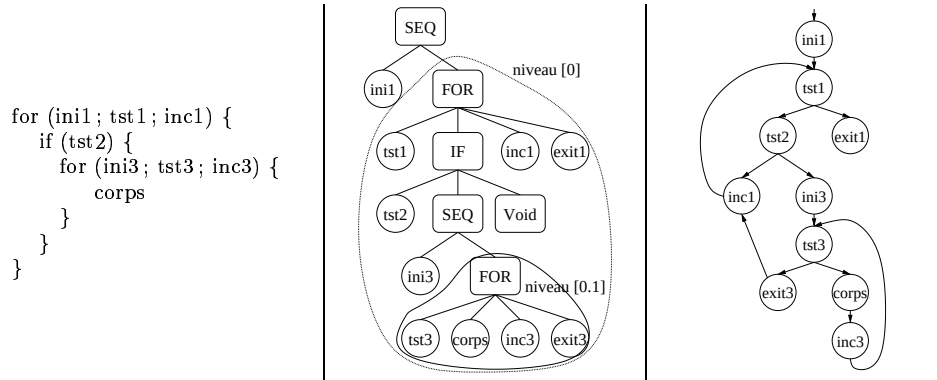
Pour être analysé, le code (assembleur) du programme est découpé en blocs nommés *blocs de base*. Un bloc de base est une suite d'instructions possédant un unique point d'entrée et un unique point de sortie, et pour lequel une instruction de transfert de contrôle (branchement, instruction de gestion des procédures) ne peut être présente qu'à la fin du bloc de base. La méthode d'analyse repose sur les structures de données schématisées sur la figure 3.5.

L'arbre syntaxique (voir figure 3.5.b) est un arbre représentant la structure syntaxique du programme. Les nœuds sont de quatre types : séquence (SEQ), conditionnelle (IF), boucle (FOR) et feuille, correspondant aux blocs de base. À chaque boucle dans l'arbre syntaxique est associé un *niveau* ([0], [1] pour les boucles de plus haut niveau, [0.0], [0.1] pour les boucles emboîtées dans la boucle [0], etc). Nous notons $\succ$ la relation d'ordre partiel définissant les inclusions (strictes) entre niveaux de boucles (par exemple, $[0] \succ [0.1] \succ [0.1.5]$). On note [] le niveau correspondant au corps d'un programme et $niveau_{\perp}(B)$ le plus petit niveau (au sens de $\succ$) dans lequel un bloc de base B est contenu.

Le graphe de flot de contrôle (voir figure 3.5.c) décrit, pour chaque fonction du programme, son flot de contrôle interne. Les nœuds du graphe sont les blocs de base et les arcs les séquencements possibles entre ces blocs de base. Notons qu'il y a une correspondance un à un entre les nœuds du graphe de flot de contrôle et les feuilles de l'arbre syntaxique.

En supposant une architecture matérielle pour laquelle le WCET d'une instruction est connu et indépendant des autres instructions, le WCET d'un programme peut être évalué simplement (voir tableau 3.1, donnant le système d'équations proposé

[2] La méthode d'analyse statique de WCET décrite dans ce paragraphe est facilement transposable à d'autres langages impératifs. Comme le logiciel implantant cette méthode d'analyse prend en entrée du code C, tous les exemples sont exprimés dans ce langage.



Code source (C) (a) — Arbre syntaxique (b) — Graphe de flot de contrôle (c)

FIG. 3.5 – Exemple de code source et ses représentations

initialement par Puschner et Koza dans [PK89]).

Construction S		
SEQ :	$S_1; \ldots; S_n$	$WCET(S) = \sum_{i=1}^{n} WCET(S_i)$
IF :	if (tst) S_1else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
FOR :	for $(;tst;inc)$ S_1	$WCET(S) = maxiter_S \times (WCET(Test) + WCET(S_1) + WCET(Incr)) + WCET(Test) + WCET(Exit)$
feuille	(bloc de base)	$WCET(S) = \sum_{i=1}^{n} WCET(I_i)$

La constante $maxiter_S$ dénote le nombre maximum d'itérations de la boucle S.
$Test$, $Incr$ et $Exit$ sont les blocs de base résultats de la compilation d'une boucle.
I_i dénote la i^{ieme} instruction du bloc de base S.

TAB. 3.1 – Calcul du WCET sur une architecture matérielle simple [PK89]

Le WCET d'un bloc de base (dernière ligne du tableau) est simplement la somme des WCETs des instructions le composant. On peut alors calculer le WCET d'un programme par un parcours hiérarchique de son arbre syntaxique (trois premières lignes du tableau). Par exemple, le WCET d'une construction conditionnelle (ligne 2 du tableau, nœud IF) est égal au WCET du test de la conditionnelle auquel on ajoute le maximum des WCETs des deux branches de la conditionnelle.

3.3.3 Réduction du pessimisme par annotations de boucles

Nous présentons ici une méthode d'annotation des boucles [CP00b] qui permet de déterminer le nombre maximum d'itérations des boucles (*maxiter*) de manière moins pessimiste que les systèmes d'annotations usuels, exprimant ce nombre maximum d'itérations par une simple constante.

Au lieu d'exprimer *maxiter* par une constante, nous proposons de l'exprimer par l'intermédiaire d'une *expression symbolique*. Ce mode d'expression permet notamment d'exprimer que *maxiter* dépend de la progression des compteurs des boucles englobantes (boucles *non-rectangulaires*).

Plus précisément, chaque boucle est annotée par un couple d'expressions mathématiques [*maxiter*, *compteur*] en lieu et place d'une valeur constante utilisée dans les méthodes traditionnelles :

- *maxiter* est une expression donnant le nombre maximum d'itérations de la boucle. Cette expression peut dépendre de plusieurs expressions *compteur* (voir ci-dessous) dans le cas d'une boucle non rectangulaire, ou encore être une simple constante.
- *compteur* est une expression définissant comment évolue le compteur de la boucle courante. *compteur* est exprimé en utilisant une variable muette λ (qui progresse de 0 à $maxiter - 1$), et le cas échéant les expressions *compteur* des boucles englobantes.

L'exemple ci-dessous illustre cette méthode d'annotations. Le programme considéré est un extrait du code de calcul de transformée de Fourier rapide sur 2048 échantillons complexes (NbSamples = 2048). Dans l'exemple, pour une boucle i on nomme les expressions de *maxiter*, *compteur* et λ pour cette boucle M_i, C_i et λ_i.

```
void fft (int NbSamples; complex tab[]) {
   ...
   Bsize = 1;
   for (i=2 ; i<= NbSamples ; i=i*2)      [M_i = 11, C_i = 2^(λ_i+1)] {
      ...
      for (j=0 ; j < NbSamples ; j=j+i)   [M_j = 2048/C_i, C_j = λ_j * C_i] {
         ...
         for (k=0; k==Bsize ; k++)        [M_k = C_i/2, C_k = λ_k] {
            ...
         }
         Bsize = i;
      }
   }
}
```

FIG. 3.6 – Exemple d'annotations symboliques

Le nombre maximum d'itérations pour la boucle i est exprimé par une constante,

alors que celui des boucles j et k dépendent de la progression du compteur de la boucle i. L'expression $C_i = 2^{\lambda_i+1}$ dans l'annotation de la boucle i indique que l'indice de boucle de la boucle i prend comme valeurs des puissances de deux.

Pour rendre la désignation des compteurs de progression de boucles indépendants des noms de variables de ces compteurs dans le code source, nous utilisons un nommage *relatif* des compteurs de progression de boucles (pile de compteurs). Ce nommage relatif permet de référencer les compteurs de progression des boucles englobantes sans référence explicite à des éléments dépendant du code source.

Avec le système d'annotations proposé, le nombre maximum d'itérations d'une boucle dépend uniquement des compteurs de progression des boucles englobantes. De plus, de par l'introduction de l'expression *compteur*, toutes les progressions des compteurs de boucles sont connues statiquement. Il est donc possible de connaître statiquement le nombre maximum d'itération de toutes les boucles en les exprimant sous la forme d'expressions symboliques composées de sommes finies (voir [CP00b] pour plus de détails). Ces expressions peuvent alors être simplifiées et évaluées en utilisant des outils comme Maple ou Mathématica (dans le logiciel prototype décrit dans le paragraphe 3.5.1.2, nous utilisons Maple).

Le tableau 3.2 montre sur l'exemple de programme de la figure 3.6 que l'évaluation du nombre maximum d'itérations des trois boucles de cet exemple est plus précise en utilisant la méthode d'annotations proposée qu'en utilisant des constantes.

Boucle	**Annotations**		**Nb. max. itérations**	
	Cst	Expr. symb.	Cst	Expr. symb.
i	11	$[M_i = 11, C_i = 2^{\lambda_i+1}]$	11	11
j	1024	$[M_j = \frac{2048}{C_i}, C_j = \lambda_j * C_i]$	11×1024 $= 11264$	$\sum_{\lambda_i=0}^{(11-1)} (\frac{2048}{2^{\lambda_i+1}})$ $= 1791$
k	1024	$[M_k = \frac{C_i}{2}, C_k = \lambda_k]$	$11 \times 1024 \times 1024$ $= 11534336$	$\sum_{\lambda_i=0}^{(11-1)} (\frac{2^{\lambda_i+1}}{2} \times \frac{2048}{2^{\lambda_i+1}})$ $= 11264$ (Gain de 99.9%)

TAB. 3.2 – Nombre maximum d'itérations estimé

Notons que l'introduction du système d'annotations proposé complexifie sensiblement le système d'équations donné dans le tableau 3.1 (voir [CP00b] pour le système d'équations complet).

Remarquons également que l'utilisation de tout système d'annotations, bien qu'améliorant la précision de l'analyse, est sujet aux erreurs quand les annotations sont disposées manuellement dans le code source. Des méthodes permettant de vérifier en-ligne la validité des annotations (e.g. [PK89]) peuvent être utilisées pour remédier à ce problème.

3.3.4 Réduction du pessimisme par analyse au niveau architecture

Le calcul de WCET tel qu'il a été présenté dans le paragraphe 3.3.2 fonctionne sous deux hypothèses. La première est que le temps d'exécution d'une instruction est supposé constant. La deuxième hypothèse est que le WCET d'une séquence d'instructions est égal à la somme des WCETs des instructions prises séparément. Si l'on considère les processeurs modernes qui intègrent des éléments architecturaux tels que les caches, les pipelines ou encore la prédiction de branchement, dont le but conjoint est d'améliorer les performances du processeur, les hypothèses précédentes ne sont plus vérifiées. Il est toujours possible de contourner ce problème en ignorant ces éléments architecturaux. Par exemple, il est possible d'ignorer la présence du cache d'instructions en supposant que tous les accès aboutissent à des *échecs* dans le cache ; la présence des pipelines peut être ignorée en supposant qu'il n'y a pas de parallélisme entre l'exécution des instructions (simple exécution en séquence). Bien qu'une telle méthode soit *sûre*, le WCET calculé dans ces conditions est une approximation très pessimiste du temps d'exécution pire cas (voir § 3.5.3 pour des éléments quantitatifs à ce sujet).

Notre proposition consiste à intégrer à la méthode de calcul de WCET une modélisation des éléments architecturaux améliorant les performances du processeur (cache d'instruction, pipeline, prédiction de branchement). L'objectif visé est de *réduire le pessimisme* de l'analyse sans compromettre la *sûreté* de l'analyse (le WCET calculé pour un programme ne doit en aucun cas être inférieur au temps d'une quelconque de ses exécutions). Notre contribution dans le domaine de l'analyse statique de WCET de bas niveau est double : d'une part, nous avons proposé une méthode permettant de prendre en compte les prédicteurs de branchements [CP00b], méthode inexistante auparavant ; d'autre part, nous avons intégré la prise en compte du cache d'instructions, du pipeline et de la prédiction de branchement dans un cadre commun [CP01c] (les trois techniques étant le plus souvent considérées de manière isolée).

Remarquons que la prise en compte des trois éléments architecturaux que sont les caches, les pipelines et les prédicteurs de branchement n'est pas triviale car ces éléments introduisent une dépendance au contexte. Par exemple, la durée d'exécution d'une instruction dans le pipeline dépend des instructions précédentes, les durées des accès mémoire en lecture et écriture dépendent de l'état des caches et donc de l'historique de l'exécution (instructions et données accédées). Ceci nécessite une modification des méthodes de calcul du WCET des instructions et donc des blocs de base.

Nous présentons dans les paragraphes qui suivent nos propositions pour prendre en compte les mécanismes de cache d'instructions (§ 3.3.4.1), de prédiction de branchement (§ 3.3.4.3) et de pipeline (§ 3.3.4.2). Nous terminons en montrant comment il est possible d'intégrer *simultanément* la prise en compte de ces trois éléments architecturaux dans un même outil d'analyse (§ 3.3.4.4).

3.3.4.1 Prise en compte du cache d'instructions

Problématique. Les évolutions techniques récentes ont eu pour effet une augmentation considérable de la différence entre les vitesses des processeurs et celles des mémoires. La mémoire est devenue un élément limitant les performances de l'architecture. Or, les mémoires rapides sont d'un coût trop élevé pour que l'on puisse en disposer en quantité importante. Une solution à ce problème est l'installation d'une *hiérarchie de caches* organisée en plusieurs niveaux, le niveau le plus bas étant le plus rapide mais de taille réduite. Quand un bloc de donnée auquel on veut accéder en est absent (*défaut de cache*), il y est recopié depuis le niveau supérieur. Grâce à ce mécanisme on dispose d'une quantité importante de mémoire presque aussi rapide que le cache à un coût presque aussi faible que celui de la mémoire centrale.

L'usage d'une mémoire cache introduit de l'indéterminisme dans l'architecture, car en ajoutant cet intermédiaire entre la mémoire centrale et le processeur, la durée des accès mémoire n'est plus constante. En effet, deux durées correspondant à deux cas d'accès au cache sont à considérer. Une première durée correspond à l'accès à un bloc mémoire qui est dans le cache (*succès* ou *hit*). Une deuxième durée correspond au cas où le bloc mémoire référencé est absent du cache (*échec* ou *miss*), il est alors recopié depuis le niveau supérieur. La durée d'un accès mémoire dans le cas d'un échec est évidemment plus importante que celle d'un succès.

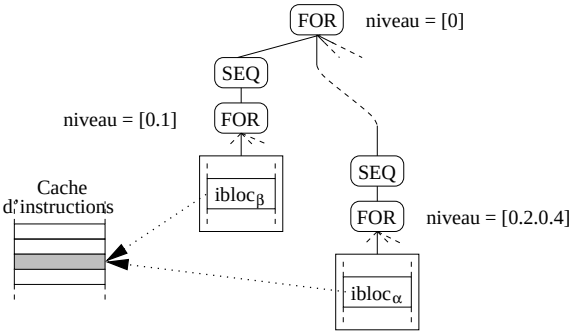
Dans le contexte de l'analyse statique de WCET, il faut pouvoir déterminer, de manière *prudente* (en ne déclarant jamais comme *succès* un accès mémoire pouvant entraîner un *échec*), le résultat (*succès/échec*) de chaque accès au cache d'instructions. Ceci nécessite de connaître statiquement le comportement du cache lors de l'exécution d'un programme. Une telle méthode est brièvement présentée ci-dessous.

Solution. Notre méthode intègre l'effet du cache grâce au classement des instructions d'un programme en fonction de leur comportement (au pire cas) par rapport au cache. Elle est inspirée du travail de F. Mueller, nommé par son auteur *simulation statique de caches* [Mue94, Mue00]. Brièvement, la méthode opère en deux étapes :

- **a. Calcul d'états abstraits de cache (ACS).**
  Cette étape consiste à calculer l'ensemble des contenus du cache d'instructions possibles avant et après l'exécution de chaque bloc de base. Ces contenus sont nommés ACS pour *Abstract Cache States*. Cette phase opère par résolution d'un système récursif d'équations opérant sur le graphe de contrôle de flot du programme à analyser (voir [Col01] pour le détail des équations). À la fin de cette première phase, on connaît pour chaque bloc de base et chaque ligne de cache l'ensemble des instructions pouvant y figurer[3]. Il est ainsi possible de connaître pour chaque instruction l'ensemble des autres instructions avec lesquelles elle est en conflit.

[3] En réalité, cette étape est appliquée au niveau de *blocs de code*, c'est-à-dire l'ensemble des instructions correspondant à une ligne de cache plutôt qu'au niveau des instructions, et ceci de manière à supporter les jeux d'instructions de taille variable. Ce détail est omis ici pour des raisons de clarté.

– **b. Classification des instructions.**
Connaissant le contenu des ACS, il est possible de savoir si une instruction est en conflit avec une autre, donc de pouvoir classer le comportement (au pire cas) de cette instruction vis-à-vis du cache. Une méthode simple consiste alors à classer une instruction *échec* lorsqu'elle est en conflit avec une autre instruction, et *succès* dans le cas contraire.
Ce mode de classification des instructions est plus précis que de considérer que tous les accès mémoire sont des échecs, mais peut être affiné en connaissant la structure syntaxique des programmes. Prenons par exemple l'instruction α schématisée ci-dessous.



Cette instruction est en conflit avec l'instruction β pour l'accès à la ligne de cache schématisée en grisé sur la figure. Toutefois, supposer qu'α provoquera un échec dans le cache à chacune de ses exécutions est trop pessimiste. En effet, d'après les niveaux de boucles auxquelles appartiennent les instructions α et β, α ne provoquera un échec dans le cache qu'à chaque itération de la boucle de niveau [0] et non pas à chaque exécution de α.
Par conséquent, au lieu d'associer une simple indication *succès/échec* à chaque instruction, nous lui associons une information qui dépend du niveau de boucle de/des instructions avec la/lesquelles l'instruction est en conflit. À chaque instruction est associée un *niveau d'échec-icache*, signifiant qu'un échec dans le cache d'instruction se produira uniquement à chaque itération correspondant à ce niveau. Un niveau d'échec-icache égal au niveau courant de l'instruction ($niveau_\perp$) signifie que l'instruction provoquera systématiquement un échec dans le cache d'instructions. Le niveau d'échec-icache correspondant à chaque instruction est calculé à partir des états abstraits de cache. L'utilisation de cette information dans le calcul de WCET final est présentée dans le paragraphe 3.3.4.4.

3.3.4.2 Prise en compte du pipeline

Problématique. L'exécution en pipeline est une technique fondamentale utilisée pour réaliser des processeurs rapides, permettant le recouvrement de l'exécution de plusieurs instructions. Afin d'expliciter brièvement le principe du pipeline, on rappelle que le cycle d'exécution d'une instruction se décompose en plusieurs étapes, par

exemple : lecture d'instruction, décodage d'instruction, exécution, accès mémoire, écriture du résultat. Le but du pipeline est d'introduire du recouvrement entre l'exécution de plusieurs instructions. Pour ce faire, le pipeline comporte plusieurs étages qui lui permettent de traiter en parallèle des étapes différentes des instructions (*cf.* figure 3.7, qui illustre les différents étages du pipeline entier d'un processeur Pentium).

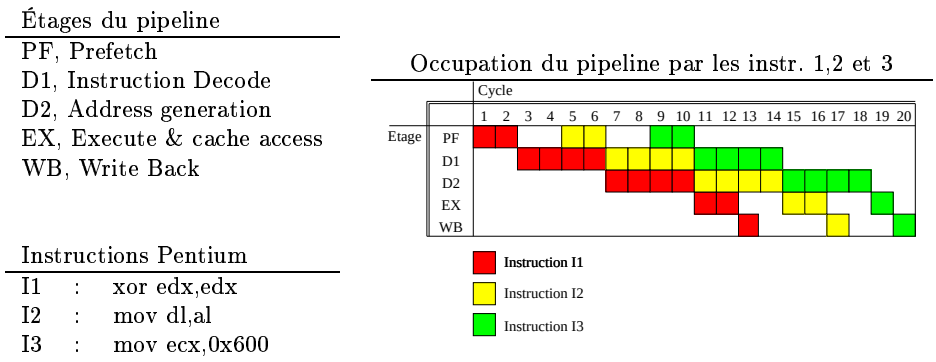


FIG. 3.7 – Occupation du pipeline entier du processeur Pentium

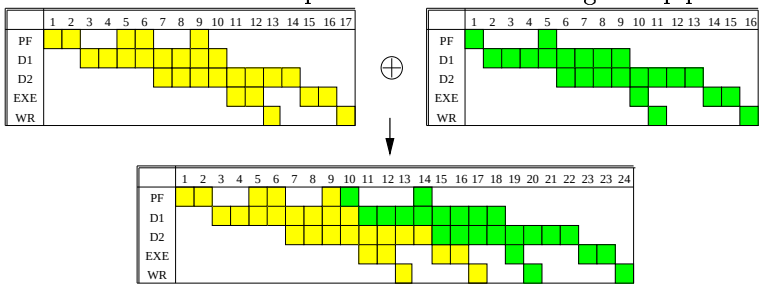
Les exécutions des instructions se chevauchent et le temps d'exécution de deux instructions consécutives dans le pipeline n'est pas la somme de leurs temps d'exécution unitaires. Il est toutefois possible de prendre en compte ce chevauchement des instructions dans le pipeline si l'on connaît la décomposition de chaque instruction dans les différents étages du pipeline. Une telle méthode est présentée ci-dessous.

Solution. Notre méthode pour prendre en compte le mécanisme de pipeline est classique dans le domaine de l'analyse statique de WCET [BJ95, ZBN93]. Elle est fondée sur une *simulation statique de pipeline*, c'est-à-dire une simulation du remplissage du pipeline avant exécution du programme. La simulation opère à deux niveaux : en interne à un bloc de base (i.e. sans possibilité de branchement) et à la frontière entre deux blocs de base (prise en compte des branchements).

- **Simulation en interne à un bloc de base.**
  Le calcul du WCET d'un bloc de base est réalisé par simulation de l'occupation du pipeline par les instructions du blocs de base. Il faut pour cela connaître pour chaque instruction d'un bloc de base son temps d'exécution au pire cas pour chaque étage du pipeline. Notons que cela nécessite de connaître le comportement de l'instruction vis à vis du cache d'instructions (succès/échec). On réalise pour chaque bloc de base une simulation d'occupation des étages du pipeline. Pour avoir le temps d'exécution au pire cas d'un bloc de base isolé, il suffit alors

de calculer la longueur des occupations de pipelines (20 dans l'exemple de la figure 3.7).

– **Frontières entre blocs de base.**
Après avoir analysé l'effet du pipeline en interne à chaque bloc de base, on dispose de descripteurs d'occupation du pipeline pendant la durée nécessaire à leur exécution. Ces descripteurs indiquent pour chaque cycle et chaque étage du pipeline s'il est occupé ou non. Pour connaître le temps d'exécution total de deux blocs de base s'exécutant l'un après l'autre, une solution sûre mais pessimiste consiste à ajouter les temps d'exécution de chacun d'eux. Pour limiter ce pessimisme, au lieu de procéder de cette manière, nous concaténons avec recouvrement les descripteurs d'occupation de pipelines pour tous les couples de blocs de base consécutifs. Les descripteurs de pipelines des deux blocs de base consécutifs sont ajustés (voir figure ci-dessous) et se recouvrent tout en respectant les contraintes d'occupation des différents étages de pipelines.



Notons que cette composition entre descripteurs de pipelines n'est effectuée que lors d'une prédiction de branchement correcte (*cf.* § 3.3.4.3). Dans le cas contraire, on se trouve dans le cas où le pipeline est inefficace et doit être vidé pour reprendre l'exécution à partir de l'instruction cible du saut. Il est alors nécessaire de prendre en compte le délai additionnel occasionné par l'erreur de prédiction de branchement (*cf.* § 3.3.4.3)

3.3.4.3 Prise en compte de la prédiction de branchement

Problématique. De nombreux processeurs comportent une logique de prédiction de branchement leur permettant d'éviter les cycles d'attente dans le pipeline si le comportement du branchement lors de son exécution est correctement prédit. Lorsqu'une opération de branchement est correctement prédite, il n'y a pas de pénalité sur les performances (on évite les cycles d'attente). Mais lorsque la prédiction de branchement n'est pas correcte, cela provoque une attente (par exemple 3 ou 4 cycles sur le processeur Pentium), le temps de "remplir" le pipeline à partir de la nouvelle adresse d'exécution.

Une approche possible pour calculer le WCET de programmes sur des architectures dotées de prédiction de branchement est de supposer que toutes les prédictions sont incorrectes. Toutefois, cette approche est bien évidemment pessimiste. Nous présentons ci-dessous une méthode sûre permettant de déterminer statiquement les

résultats de la prédiction de branchement, et donc de limiter ce pessimisme.

Solution. Nous présentons brièvement ici une technique permettant de prendre en compte de manière précise et sûre le mécanisme de prédiction de branchement dans le calcul de WCET (pour des détails sur cette technique, le lecteur se référera à l'article [CP00b]). À notre connaissance, cette technique est la première technique permettant de prendre en compte le mécanisme de prédiction de branchement dans une méthode d'analyse statique de WCET.

Les méthodes de prédiction de branchement pouvant différer de manière significative d'un processeur à un autre, nous nous concentrons dans la suite de ce paragraphe sur les processeurs utilisant un tampon de cibles de branchement (BTB - Branch Target Buffer) comme par exemple le processeur Pentium utilisé dans nos expérimentations. Pour le Pentium, le BTB est une mémoire cache dont chaque entrée contient, entre autres :

- des bits d'historique, mémorisant si le branchement a été pris ou non lors de ses précédentes exécutions,
- l'adresse de l'instruction de branchement (adresse source),
- l'adresse cible du branchement.

Lors de la prédiction du résultat d'une instruction de transfert de contrôle, celle-ci peut-être présente ou non dans le BTB, résultant dans deux modes de prédiction :

- *Prédiction par défaut.* Si l'instruction considérée n'est pas dans le BTB et donc qu'aucune information sur l'historique de l'instruction n'est disponible, une prédiction par défaut est effectuée (dans le cas Pentium, le branchement est prédit non-pris).
- *Prédiction selon l'historique.* Quand l'instruction est présente dans le BTB une prédiction fondée sur l'historique de branchement de l'instruction est effectuée. En particulier, si les bits d'historique indiquent que le branchement a été pris lors des dernières exécutions, le BTB fournit l'adresse cible du branchement au pré-chargeur.

Comme la méthode de prise en compte du cache d'instructions, la méthode de prise en compte du mécanisme de prédiction de branchement procède en deux étapes :

- **a. Calcul d'états abstraits de BTB.**
  Cette étape consiste à calculer l'ensemble des contenus du tampon de cibles de prédiction de branchement (BTB) possibles avant et après l'exécution de chaque bloc de base. Comme dans le cas du cache d'instructions, cette phase résout un système récursif d'équations par itérations de point fixe opérant sur le graphe de contrôle de flot du programme à analyser. À la fin de cette première phase, on connaît pour chaque bloc de base et chaque ligne du BTB l'ensemble des instructions pouvant y figurer.
- **b. Classification des instructions de transfert de contrôle.**
  Contrairement au cas du cache d'instructions, les contenus possibles du BTB ne donnent pas une information directement exploitable pour classer les instructions de branchement. En effet, savoir si une instruction est ou non dans le

BTB donne seulement le *mode de prédiction utilisé* (par défaut ou selon l'historique) et pas le *résultat de la prédiction* (pris ou non pris). Notre proposition est de prendre en compte le *rôle* de chaque instruction de transfert de contrôle (test de conditionnelle, test de boucle, sortie de boucle, etc.), qui est connu à partir de la structure syntaxique du programme, pour connaître le résultat de sa prédiction. Ainsi, par exemple, une instruction de test de sortie d'une boucle, dans le cas où son adresse est dans le BTB, est toujours prédite *prise* sauf lors de la dernière itération de la boucle.
Les contenus possibles du BTB ainsi que les informations sur le rôle des instructions de transfert de contrôle permettent d'effectuer une classification de ces instructions pour chacun des choix possibles à l'issue de l'instruction (continuer en séquence ou branchement). Comme dans le cas du cache d'instructions, cette classification dépend du niveau d'emboîtement des boucles. Ainsi, pour chaque bloc de base et chaque possibilité d'exécution (*seq* pour une exécution en séquence ou *jmp* pour une rupture de séquence) on associe un *niveau d'échec-prédiction*. Ce niveau indique qu'une prédiction erronée se produira à chaque itération de niveau inférieur ou égal à ce niveau.

3.3.4.4 Intégration

Les méthodes de prise en compte des caches d'instruction et des pipelines sont maintenant bien maîtrisées dans le domaine de l'analyse statique de WCET. En revanche, les méthodes existantes sont généralement conçues indépendamment les unes des autres et ne sont pas intégrées dans une même méthode de calcul. Afin de résoudre ce problème d'intégration, nous avons proposé un formalisme commun permettant d'exprimer les résultats des analyses de cache d'instructions, pipelines et prédiction de branchement, et modifié les équations de Puschner et Koza (tableau 3.1) pour intégrer ce formalisme et ainsi prendre en compte de manière *simultanée* la présence de ces trois éléments architecturaux [CP01c].

Le principe général utilisé pour intégrer les résultats de la prise en compte de la micro-architecture est le suivant. Pour un bloc de base B donné, la simulation statique de pipeline (§ 3.3.4.2) est utilisée pour calculer le WCET de B pour un ensemble de niveaux de boucles : niveau de boucle associé à B ($niveau_\perp$), niveaux d'échec-icache de toutes les instructions de B, niveau d'échec-prédiction de B. Si l'on classe ces niveaux par ordre décroissant (selon $\succ$), on obtient alors une liste de WCETs W_i ($1 \leq i \leq q$) classée par niveau de boucle croissant.

Les équations de base du tableau 3.1 peuvent alors être modifiées. Au lieu de représenter le WCET d'un bloc de base B par un simple scalaire comme dans les équations d'origine, le WCET d'un bloc de base est maintenant représenté par une séquence de couples $\{< W_1, niveau_\perp >, < \delta_1, n_1 >, ..., < \delta_q, n_q >\}$, avec $\delta_i = W_{i+1} - W_i$. De même, on augmente chaque équation de calcul du WCET par le niveau de boucle de la construction dont on évalue le WCET (on évalue le WCET d'une fonction f en évaluant $WCET(f, [])$). Les équations modifiées de calcul de WCET sont données dans le tableau 3.3.

Construction S	
SEQ : $S_1; \ldots; S_n$	$WCET(S,L) = WCET(S_1,L_1) \oplus \ldots \oplus WCET(S_n,L_n)$ $WCET^{seq}(S,L) = WCET(S_1,L_1) \oplus \ldots \oplus WCET(S_{n-1},L_{n-1})$ $\oplus WCET^{seq}(S_n,L_n)$ $WCET^{jmp}(S,L) = WCET(S_1,L_1) \oplus \ldots \oplus WCET(S_{n-1},L_{n-1})$ $\oplus WCET^{jmp}(S_n,L_n)$
IF : if (tst) S_1 else S_2	$WCET(S,L) = max($ $WCET^{seq}(Test,L) \oplus WCET(S_1,L)$, $WCET^{jmp}(Test,L) \oplus WCET(S_2,L)$ $)$
FOR : for $(;tst;inc)$ S_1	$WCET(S,L) = maxiter_S \overset{L}{\otimes} \big(WCET^{jmp}(Test,L) \oplus WCET(S_1)$ $\oplus WCET(Incr,L)\big)$ $\oplus WCET^{seq}(Test,L) \oplus WCET(Exit,L)$
feuille (bloc de base BB)	$WCET(BB,L) = WCET\_BB^{?}$ $WCET^{seq}(BB,L) = WCET\_BB^{seq}$ $WCET^{jmp}(BB,L) = WCET\_BB^{jmp}$

$WCET\_BB^{?}$ est le WCET d'un bloc de base ayant une seule branche sortante.
$WCET^{seq}$ et $WCET^{jmp}$ correspondent aux deux possibilités d'exécution d'un bloc de base.
$maxiter_S$ est le nombre maximum d'itérations de la boucle S.

TAB. 3.3 – Équations de calcul de WCET intégrant cache, pipeline et BTB

Dans le tableau, les opérateurs $\oplus$ et $\otimes$ sont des adaptations des opérateurs entiers $+$ et $\times$ à la nouvelle représentation des WCETs ($\oplus$ réalise une union, puis ajoute les valeurs à niveaux d'emboitement égaux ; $\otimes^L$ applique un facteur multiplicatif à tous les éléments de niveau n inférieur ou égal à L). Des explications plus approfondies sur ces équations sont données dans le document [CP01c].

3.4 Intégration de composants existants

Pour des raisons de coût, à la fois du point de vue du prix d'achat et du coût de maintenance, de nombreux industriels se tournent vers l'utilisation de composants (matériels ou logiciels) *sur étagères* pour supporter des applications ayant des contraintes de tolérance aux fautes et de temps-réel. Un composant est dit *sur étagères* (en anglais COTS, pour Commercial-Off-The-Shelf) s'il est diffusé en grande quantité (et donc généralement à faible coût) et est utilisé sans modification. La large diffusion des composants *sur étagères* fait que ce sont le plus souvent des composants généralistes, non dédiés à des contraintes spécifiques d'un domaine d'application particulier (tolérance aux fautes, temps-réel). Nous examinons l'impact qu'a l'utilisation de composants COTS sous l'angle de la tolérance aux fautes (§ 3.4.1) et du temps-réel (§ 3.4.2).

3.4.1 Composants COTS et tolérance aux fautes

Les composants COTS, tant au niveau matériel qu'au niveau logiciel sont des composants à large diffusion, et ne sont pas en général conçus pour être intégrés dans des systèmes tolérants aux fautes. Les intégrer dans des systèmes tolérants aux fautes nécessite d'une part de *caractériser* le comportement de ces composants en présence de fautes, et d'autre part de *complémenter* les fonctions de ces composants pour qu'ils vérifient des propriétés de haut niveau en présence de fautes. Nous présentons nos travaux dans ces deux directions ci-dessous.

3.4.1.1 Évaluation du mode de défaillance

Nos travaux concernant l'évaluation du mode de défaillance de composants COTS ont été effectués dans le cadre de la conception du support d'exécution HADES (voir § 3.5 pour une description du prototype). La méthode que nous avons développée pour l'évaluation est une méthode *d'injection de fautes*, consistant à introduire intentionnellement des fautes dans le logiciel à évaluer dans l'objectif d'examiner son comportement en présence de fautes. Bien que ces travaux aient été effectués dans le cadre des systèmes temps-réel, la méthode développée est adaptable à d'autres types de logiciels. En particulier, la méthode pourrait être utilisée pour évaluer le mode de défaillance des calculateurs utilisés pour l'exécution d'applications parallèles que nous avons considérées dans le chapitre 2.

La méthode d'injection de fautes que nous avons conçue est une injection de fautes *par logiciel*. L'objectif est de simuler l'occurrence de fautes physiques. Dans ce travail, nous simulons les fautes physiques transitoires, qui sont les fautes physiques les plus courantes dans les systèmes informatiques [Lal85]. L'injection est effectuée par logiciel, en introduisant la manifestation des fautes (les *erreurs*) dans la mémoire du support d'exécution. L'injection est déclenchée à un instant aléatoire, et l'emplacement mémoire modifié est lui aussi sélectionné de manière aléatoire (dans le code, les données ou la pile du support d'exécution). La corruption mémoire consiste à inver-

ser un bit d'un octet. Ce modèle de fautes est en accord avec la plupart des travaux utilisant l'injection de fautes par logiciel [BCSS90, KKT95, CMS98, SMFA99]. Une bonne discussion sur la représentativité de ce mode d'injection vis-à-vis des fautes physiques transitoires peut être trouvée dans l'article [CMS98].

Un outil d'injection de fautes a été développé, en limitant au maximum l'effet intrusif sur le système analysé. Quelques détails sur son implantation sont donnés dans le paragraphe 3.5.2.

3.4.1.2 Maintien de propriétés d'exécution de haut niveau

Afin de pouvoir raisonner sur le comportement des applications temps-réel strict en présence de fautes, et notamment d'analyser leur ordonnançabilité, il est nécessaire que le support d'exécution vérifie des propriétés, qui définissent son comportement en présence de fautes. Ainsi, dans le cadre de la conception du support d'exécution d'HADES, nous avons identifié dans le paragraphe 3.2.1.2 un ensemble de propriétés (silence sur défaillances des calculateurs, transmission de messages fiable en temps borné, détection de l'arrêt et le redémarrage des calculateurs en temps borné, synchronisation des horloges) et nous avons montré qu'avec ces propriétés il est possible d'analyser l'ordonnançabilité du système, même en présence de fautes (voir § 3.2.3).

Nous nous intéressons ici à assurer ces propriétés en utilisant exclusivement du matériel (processeur, réseau) et un système d'exploitation COTS. Ces propriétés sont assurées par deux types de mécanismes. D'une part, des mécanismes de détection d'erreurs réalisent des auto-tests sur les données et le flot de contrôle du support d'exécution de manière à assurer l'hypothèse de silence sur défaillances. D'autre part, trois protocoles, décrits brièvement ci-dessous et détaillés dans [ACCP99] assurent les autres propriétés en supposant que l'hypothèse de silence sur défaillances est vérifiée. Ces protocoles sont un protocole de diffusion multipoint, un protocole de gestion de groupes et un protocole de synchronisation des horloges. Il supposent que l'environnement est *synchrone* (en particulier, le délai de transmission d'un message sur le réseau est borné et sa borne est connue), qu'au pire le réseau peut perdre successivement ω messages dans une fenêtre de temps de taille bornée, et que le temps d'accès au médium de communication au niveau physique s'effectue en temps borné. L'intérêt de ces protocoles ne réside pas dans les algorithmes distribués qui ont été utilisés, qui sont à ce jour bien maîtrisés en univers synchrone (nous avons d'ailleurs le plus souvent utilisé directement ou adapté légèrement des algorithmes existants). Notre contribution se situe plutôt dans l'implantation de ces protocoles, qui repose essentiellement sur des tâches périodiques, afin que leurs coûts d'exécution puissent être aisément intégrés dans une analyse d'ordonnançabilité (voir § 3.2.3). Par exemple, nous n'utilisons pas de tâche déclenchée sur interruptions levées lors de l'arrivée de trames réseau.

Notons que les propriétés introduites dans le paragraphe 3.2.1.2, hormis la propriété de silence sur défaillances, spécifient toutes un comportement du support d'exécution en temps borné. Par conséquent, la cible des trois protocoles que nous avons conçus pour assurer ces propriétés est l'ensemble des systèmes temps-réel *strict*, et

ces protocoles ne sont pas nécessairement adaptés à des systèmes où les garanties de temps de réponse ne sont pas impératives.

Mécanismes de détection d'erreurs. Les systèmes d'exploitation que nous avons utilisés dans le prototype n'ont pas à la base été conçus pour exécuter des applications ayant des besoins de tolérance aux fautes. Ainsi, leur comportement en présence de fautes (notamment la couverture de l'hypothèse de silence sur défaillance) n'est pas satisfaisant (voir § 3.5). C'est pourquoi nous avons ajouté à ces systèmes d'exploitation un ensemble de mécanismes de détection d'erreurs. Ils intégrés à l'intergiciel HADES, donc à l'extérieur du système d'exploitation. Ils détectent les erreurs temporelles et par valeur et arrêtent le calculateur courant dès qu'une erreur est détectée.

Les mécanismes de détection d'erreurs intégrés à HADES sont résumés dans le tableau 3.4. Ils peuvent être classés en trois catégories : (a) des mécanismes détectant des erreurs par comparaison d'informations redondantes ; (b) des mécanismes de vérification de contrôle de flot et de temps d'exécutions ; (c) des mécanismes de détection d'erreurs de bas niveau, intégrés au processeur et au système d'exploitation utilisé dans le prototype. Dès qu'un mécanisme de détection d'erreurs détecte une erreur (temporelle ou par valeur) sur un calculateur, ce dernier est arrêté.

Synchronisation d'horloges. Le protocole de synchronisation d'horloges que nous avons conçu garantit les propriétés d'*accord* (*agreement* – l'écart maximal entre deux horloges locales quelconques est borné par une constante appelée *précision*) et d'*exactitude* (*accuracy* – la vitesse de progression de chaque horloge est contenue dans un intervalle borné). Ce protocole est une adaptation du protocole de LUNDELIUS-LYNCH [WL88].

Le protocole de LUNDELIUS-LYNCH a été sélectionné car il repose essentiellement sur des activités périodiques et garantit la propriété d'accord malgré les arrêts de calculateurs (voir [AP98] pour une évaluation expérimentale du comportement d'un ensemble d'algorithmes de synchronisation d'horloges en présence de fautes). Deux modifications ont été apportées au protocole de LUNDELIUS-LYNCH. D'une part, pour obtenir la meilleure précision possible sans utiliser de matériel spécifique (la précision dépend de la latence réseau maximale), le protocole utilise directement le réseau physique sans passer par le service de diffusion fiable : pour masquer ω omissions successives, chaque calculateur diffuse $\omega + 1$ messages contenant la valeur de son horloge. D'autre part, pour éviter que les horloges ne reculent lors des re-synchronisations, au lieu de changer la valeur d'une horloge, nous modifions la durée entre deux incrémentations de sa valeur.

Gestion de groupe. Le protocole de gestion de groupes (*group membership*) (voir [ACCP99] pour plus de détails) est chargé de déterminer sur chaque calculateur la liste des calculateurs arrêtés. Une première caractéristique de ce service, commune avec celui de diffusion fiable, est d'être conçu de manière à ne pas dépendre de la

Mécanisme	**Description**
	a. Mécanismes de détection d'erreurs à base de redondance
CODING	*Encodage de données lors des communications locales*, via utilisation d'une somme de contrôle (*checksum*).
ROBUST	*Structures de données robustes* (listes, piles) via l'ajout de redondance dans les données de contrôle de ces structures de données (e.g. chaînage redondant dans les listes).
CODING_STATIC	*Encodage des structures de données statiques.* Une somme de contrôle est ajoutée à la structure de données décrivant les propriétés statiques des tâches. La vérification de conformité est effectuée quand le support d'exécution est inactif (tâche *idle*).
	b. Vérifications temporelles et de flot de contrôle
DEADLINE	*Détection des dépassements d'échéances* des tâches
TIMEOUT	*Détection des dépassements de temps d'exécution au pire cas* des tâches.
ARRIVAL	*Vérification de la périodicité* des tâches (strictement périodique ou sporadique).
CALLGRAPH	*Vérification du graphe de flot de contrôle* du support d'exécution, par confrontation entre l'enchaînement des appels de fonction et un graphe de flot de contrôle généré avant exécution grâce à la réécriture du code source du support d'exécution.
AUTOMATON	*Vérifications des états de protocoles.* Ce mécanisme vérifie, pour les protocoles implantés sous forme d'automates, la validité des transitions (e.g. état de chaque tâche qui peut être bloqué, prêt, actif, terminé).
ECI	*Error capturing instructions* insérant des instructions déclenchant une exception dans les zones de code et de données qui ne sont pas utilisées en fonctionnement normal.
VMEMORY	*Virtual memory*, déclenchant une exception quand une tâche tente d'accéder à un segment mémoire qui n'est pas projeté dans l'espace d'adressage courant. Un espace d'adressage séparé est associé à chaque tâche du support d'exécution.
STRUCTURE	*Vérification des structures de données* sans utilisation de redondance (vérification d'indices de tableaux, des chaînages dans les listes).
SEMAN	*Vérifications sémantiques* vérifiant par le biais d'assertions disposées dans le code du support d'exécution, que des invariants liés à la sémantique du fonctionnement du support d'exécution sont vérifiés.
	c. Mécanismes de détection d'erreurs de bas niveau
CPU	CPU représente l'ensemble des mécanismes de détection d'erreurs du processeur (ici, processeur Pentium). Cette classe de mécanismes de détection d'erreurs ne peut pas être désactivée.
OS	OS représente l'ensemble des mécanismes de détection d'erreurs du noyau (codes d'erreur retournés par les appels système, paniques).

TAB. 3.4 – Mécanismes de détection d'erreurs

précision de synchronisation d'horloges[4]. Une seconde caractéristique commune aux deux services est de nécessiter l'utilisation d'un réseau physique où le temps d'accès pire cas au médium est borné, comme par exemple FDDI ou ATM.

Le protocole de gestion de groupes que nous avons conçu offre la propriété de *ponctualité* qui est essentielle dans notre cadre (la détection de l'arrêt ou du démarrage d'un calculateur est effectuée en temps borné). Ce protocole offre également la propriété de *vivacité* (l'arrêt ou le démarrage d'un calculateur est nécessairement détecté par les autres) et la propriété d'*intégrité* (pas de fausse détection de l'arrêt d'un calculateur).

Pour cela, nous utilisons un protocole très simple, puisqu'il consiste à diffuser périodiquement un message de vie sur le réseau physique et à consulter les messages de vie reçus d'autres calculateurs durant la dernière période. Si le calculateur i est $\omega + 2$ périodes sans recevoir de trames du calculateur j, alors le calculateur j est nécessairement arrêté, et si le calculateur i reçoit une trame du calculateur j qui était arrêté, alors nécessairement, il vient de redémarrer.

Diffusion fiable. Le protocole de diffusion fiable que nous avons conçu (voir [ACCP99] pour plus de détails) offre diverses propriétés dont les plus importantes sont la *ponctualité* (*timeliness* – diffusion d'un message en temps borné), la *validité* (*validity* – si un message est diffusé, et si l'émetteur ne s'arrête pas, alors le message est nécessairement livré aux destinataires non arrêtés), l'*accord* (*agreement* – si un message est adressé à deux destinataires non arrêtés C_1 et C_2, si C_1 le livre et que C_2 ne s'arrête pas pendant le déroulement du protocole, alors C_2 le livre également), et la livraison selon l'ordre causal.

Sur chaque calculateur, le protocole est exécuté périodiquement (utilisation d'une tâche périodique qui est chargée d'exécuter le protocole, de lire les messages à émettre dans une file d'attente, et d'appeler les tâches destinataires des messages à livrer). Un message est diffusé $\omega + 1$ fois. Lorsqu'un calculateur destinataire d'un message le reçoit, il diffuse successivement $\omega + 1$ acquittements. Un calculateur destinataire d'un message le livre à l'application s'il reçoit un acquittement pour ce message de tous les calculateurs destinataires et non arrêtés, sinon le message est oublié (pour garantir la propriété d'accord). L'accord étant obtenu en une phase, ce protocole suppose qu'au plus un calculateur s'arrête durant la diffusion d'un message. La causalité est garantie grâce à l'utilisation d'horloges vectorielles.

3.4.2 Composants COTS et temps-réel strict

Utiliser des composants COTS (au niveau matériel et système d'exploitation) comme briques de base pour construire un support d'exécution dédié aux applications

[4]La synchronisation d'horloges étant entièrement implantée par logiciel, la précision de la synchronisation est nécessairement beaucoup plus élevée (c'est-à-dire moins bonne) qu'en utilisant du matériel spécifique. Par conséquent, nous avons préféré concevoir un algorithme qui ne dépend pas de la précision de la synchronisation.

temps-réel strict nécessite d'obtenir de manière sûre les caractéristiques temporelles des composants logiciels COTS qui sont utilisés (voir paragraphe 3.4.2.1).

Mais cette première approche a des limites. Il est par exemple possible que les informations concernant les composants logiciels ou matériels utilisés ne soient pas assez complètes pour que leur comportement temporel soit identifié de manière sûre. Il est alors nécessaire d'adapter dynamiquement l'ordonnancement de manière à faire face à des comportements temporels incertains. Nos travaux dans cette direction sont introduits dans le paragraphe 3.4.2.2.

3.4.2.1 Caractérisation du comportement temporel

Afin de caractériser le comportement temporel de logiciel COTS, nous avons appliqué la méthode d'analyse statique de WCET présentée dans le paragraphe 3.3 à un noyau de système temps-réel COTS nommé RTEMS [OAR98], dont les sources sont publics. Par ailleurs, notre méthode d'analyse statique de WCET, et plus précisément ses éléments de prise en compte de la microarchitecture sont adaptés à des processeurs COTS (le prototype d'analyseur a comme cible un processeur Pentium). Les résultats qualitatifs et quantitatifs de l'analyse de ce noyau sont détaillés dans les articles [CP01b] et [CP01d], et sont abordés dans le paragraphe 3.5.3, qui évalue les performances de l'outil d'analyse statique de WCET que nous avons développé. Notre expérience acquise lors de l'analyse de ce noyau (voir § 3.5.3) montre qu'avec un accès au code source du noyau et aux caractéristiques temporelles de l'architecture, il est possible d'obtenir de manière sûre et pas excessivement pessimiste (facteur 1.8 en moyenne) les temps d'exécution au pire cas des appels systèmes de RTEMS. Remarquons toutefois que l'accès au code source restreint la gamme de composants COTS pouvant être caractérisés par une méthode d'analyse statique.

3.4.2.2 Ordonnancement en univers incertain

Caractériser le comportement temporel de composants logiciels sur étagères (application, système d'exploitation) n'est pas toujours possible dans le cadre général. En effet, pour obtenir de manière sûre le temps d'exécution au pire cas de tels composants, il faut soit avoir accès à leur code source, en utilisant une méthode d'analyse statique de temps d'exécution au pire cas (par exemple celle ayant été présentée au paragraphe 3.3), soit avoir des spécifications du composant suffisamment détaillées pour identifier le pire scénario d'exécution du composant. Quand ce type d'information n'est pas disponible, il est uniquement possible d'obtenir un temps d'exécution mesuré pour un scénario d'exécution (données d'entrées par exemple) donné, qui pourra potentiellement être dépassé lors de scénarios d'exécution différents.

De manière analogue, les lois d'arrivée des tâches ou des fautes peuvent être connues de manière imparfaite. Il est alors nécessaire que le système intègre des mécanismes d'adaptation dynamique, permettant de s'adapter aux conditions opérationnelles courantes. Ceci nécessite notamment d'utiliser des algorithmes d'ordonnancement *dynamiques*, permettant dans le cas de changement de conditions opéra-

tionnelles, de reconfigurer le système (par exemple arrêter des tâches d'importance moindre pour assurer un service minimum). L'objectif est ici d'adapter la qualité du service rendu aux conditions opérationnelles courantes (on sort ici du cadre du temps-réel strict si l'on ne connait pas les pires conditions d'exécution – WCET, lois d'arrivée – au moins pour un sous ensemble des tâches).

Afin d'atteindre cet objectif, nous construisons actuellement une infrastructure de simulation qui permettra d'évaluer plusieurs ordonnanceurs dynamiques existants [Str95, BSS95]. Ces ordonnanceurs proposent tous de superviser de façon dynamique la configuration du système en réponse aux évolutions du comportement de l'environnement. Pour l'évaluation de ces mécanismes, il s'agit de définir quelles sont les métriques de comparaison (taux d'échéances manquées en tenant compte de l'importance des tâches, réactivité à un changement de conditions), et quels sont les échantillons de test permettent d'obtenir des résultats pertinents pour les comparer (échantillons statistiques ou utilisation de traces réelles d'exécution). Ces résultats nous permettront d'identifier les failles des algorithmes existants et de proposer un nouvel algorithme d'ordonnancement dynamique.

3.5 Expérimentation : l'environnement Hades

L'ensemble des recherches qui ont été présentées dans ce chapitre ont été intégrées dans un prototype. Ce prototype est composé d'*outils de développement et d'analyse* d'applications temps-réel (décrits dans le § 3.5.1), et d'un *support d'exécution* associé, nommé HADES pour *Highly Available Distributed Embedded System* (décrit dans le § 3.5.2). Nous examinons les performances de ce prototype dans les paragraphes 3.5.3 et 3.5.4.

3.5.1 Outils de développement et d'analyse d'applications temps-réel

Une chaîne d'outils pour le développement et l'analyse d'applications temps-réel tolérantes aux fautes a été conçue. Les outils de cette chaîne s'exécutent sur Sun/Solaris. Les outils sont brièvement présentés ci-dessous, dans l'ordre de leur utilisation lors de la phase de conception :

- Éditeur (graphique et textuel) de la structure des applications (Fig. 3.8), sous forme de graphes décorés avec des attributs, notamment les attributs temporels des tâches – lois d'arrivée, échéances – (modèle de tâches présenté au § 3.2.1).
- Outil de duplication des tâches, insérant à la structure des tâches la redondance nécessaire au recouvrement d'erreurs, selon la méthode présentée dans le paragraphe 3.2.2.
- Outil d'analyse statique de programmes permettant à partir d'un code source écrit en C, d'obtenir son pire temps d'exécution sur une architecture matérielle donnée (actuellement Pentium). Cet outil, dont les principes de fonctionnement ont été présentés dans le paragraphe 3.3, intègre une modélisation de cache d'instruction, pipeline et prédiction de branchement [CP00b], de manière à obtenir une analyse de temps sûre, sans être trop pessimiste.
- Outil d'analyse d'ordonnançabilité, qui à partir de la description de tâches (graphes des tâches, fréquence et échéances de tâches) analyse si les échéances des tâches pourront toujours être respectées. Cet outil d'ordonnançabilité, dont le mode de fonctionnement a été présenté dans le § 3.2.3.2, prend en compte les contraintes imposées par l'introduction de mécanismes de recouvrement d'erreurs dans la structure des tâches.

Si l'analyse d'ordonnançabilité échoue à trouver un ordonnancement faisable, le concepteur de l'application modifie son application (changement du nombre de tâches ou de leur code, ou encore de la stratégie de redondance utilisée pour les fiabiliser). Ce processus est réitéré jusqu'à ce qu'une solution satisfaisant toutes les échéances soit trouvée.

Nous donnons dans les paragraphes qui suivent plus de détails sur les outils d'édition des applications, de fiabilisation et d'analyse de temps d'exécution au pire cas.

3.5.1.1 Outils d'édition et fiabilisation des applications

La figure 3.8 illustre le fonctionnement de l'outil d'édition en montrant une partie des fenêtres visibles lors de la phase d'édition d'une application. La partie haute de la figure montre la fenêtre de saisie de la structure des tâches sous la forme de graphes. D'autres fenêtres permettent de saisir les attributs, notamment temporels, des tâches (lois d'arrivée, échéances, placement, etc) et de saisir le code source C des nœuds les composant.

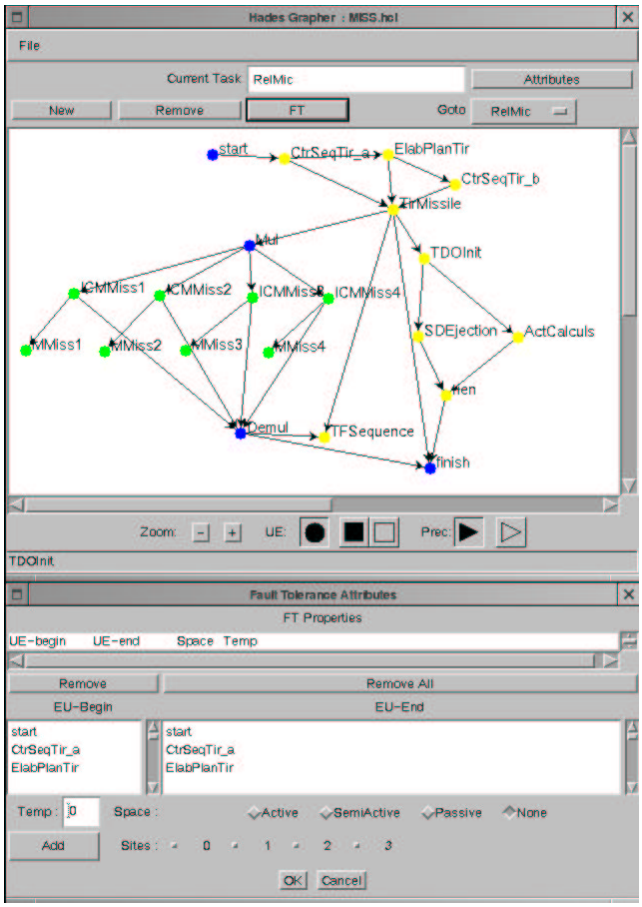


FIG. 3.8 – Éditeur graphique d'applications

La partie basse de la figure montre comment sont spécifiés les attributs de tolérance aux fautes. Sont indiqués via l'interface graphique les motifs à dupliquer, ainsi que la stratégie de redondance utilisée (active, semi-active, passive). L'outil de dupli-

cation travaille ensuite par transformation de la structure de graphe de l'application.

3.5.1.2 Outil d'analyse de WCET (Heptane)

L'outil d'analyse statique de WCET que nous avons développé est nommé Heptane pour Hades *Embedded Processor Timing ANalyzEr*. Sa structure interne est schématisée sur la figure 3.9. Les zones de couleur blanche sur le schéma représentent les modules spécifiques à l'architecture Pentium, les autres zones étant indépendantes de l'architecture cible.

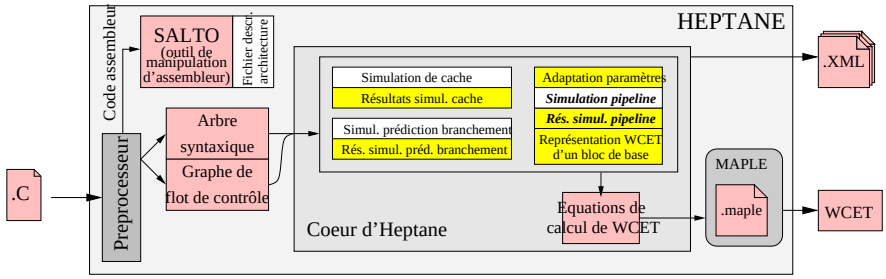


Fig. 3.9 – Structure interne de l'outil d'analyse statique Heptane

Heptane est composé de quatre éléments :

- Un préprocesseur génère l'arbre syntaxique du programme analysé. Le préprocesseur analyse la syntaxe des programmes sources (code C et annotations). Il génère d'une part l'arbre syntaxique des programmes, et d'autre part un source C modifié, dans lequel sont insérées des étiquettes permettant par la suite d'établir une correspondance entre l'arbre syntaxique et le graphe de flot de contrôle.
- Un outil externe, nommé Salto [BRS96], dédié à la restructuration de code source assembleur, est utilisé par Heptane. Salto est utilisé pour obtenir le graphe de flot de contrôle du programme à partir de son code assembleur, ainsi que les informations dépendantes du matériel (utilisation des ressources internes à la microarchitecture) ;
- Le cœur d'Heptane inclut des modules pour la gestion du cache d'instruction, du pipeline et de la prédiction de branchement.
- L'outil externe Maple est utilisé pour évaluer les expressions symboliques produites en sortie d'Heptane.

Heptane est développé en Caml, excepté le préprocesseur et les parties exploitant Salto, qui sont développées en C et C++.

Heptane prend en entrée un programme source C Ansi et génère, outre la valeur brute du WCET, un ensemble de fichiers XML donnant des détails sur les phases

intermédiaire du calcul de WCET.

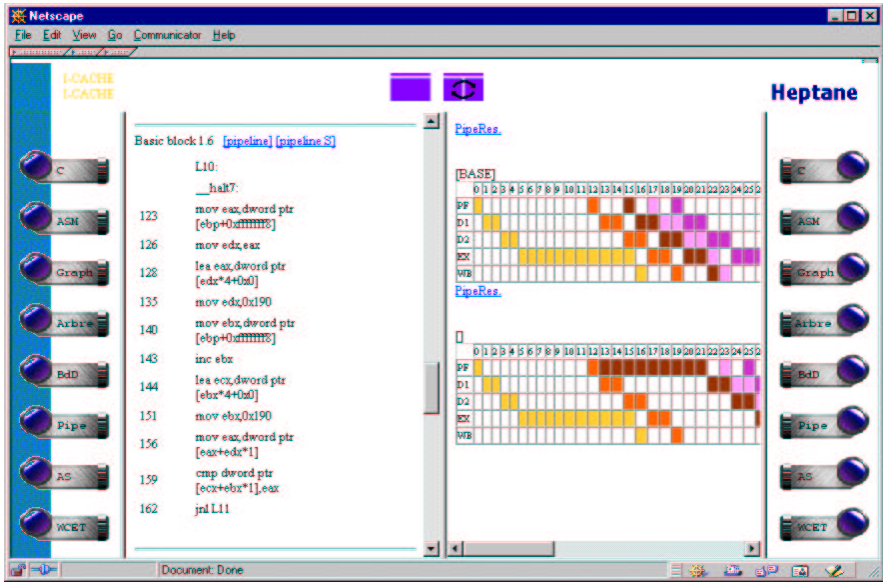


FIG. 3.10 – Outil d'analyse statique HEPTANE : visualisation de la simulation de pipeline

La figure 3.10 visualise pour un bloc de base donné, dont le code source assembleur est donné sur la partie gauche de la figure, le résultat de la simulation de pipeline (partie droite de la figure). Le résultat de la simulation de pipeline est visualisé en montrant l'occupation des cinq étages du pipeline entier du Pentium par les différentes instructions du bloc de base. Les deux simulations de pipeline apparaissant dans la figure correspondent aux deux niveaux de boucles qui sont considérés pour intégrer les résultats de l'analyse de cache d'instructions et de prédiction de branchement (voir § 3.3.4.4).

3.5.2 Support d'exécution temps-réel tolérant aux fautes

Le support d'exécution HADES se présente sous la forme d'une couche de logiciel intermédiaire (*intergiciel*) intercalée entre un noyau de système d'exploitation temps-réel *sur étagères* et l'application. Nous décrivons sa structure interne puis donnons quelques détails sur chacun des modules logiciels le composant.

3.5.2.1 Contenu et structure du logiciel

La structure interne du support d'exécution HADES est représentée sur la figure 3.11 (zones en grisé sur la figure).

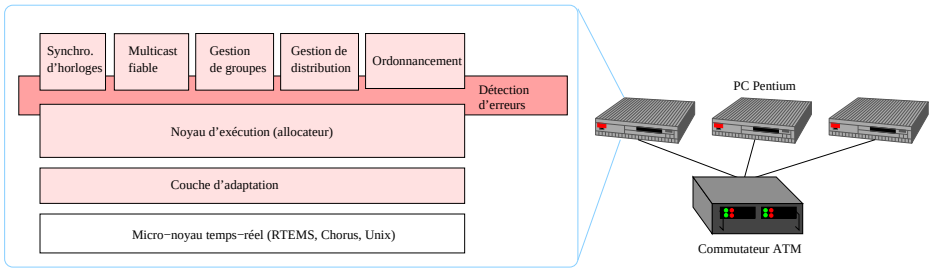


FIG. 3.11 – Support d'exécution Hades

Le support d'exécution HADES est structuré en trois niveaux, nommés respectivement *couche d'adaptation*, *noyau d'exécution* et *services*.

3.5.2.2 Couche d'adaptation

Le niveau bas, nommé *couche d'adaptation* est une sur-couche au système d'exploitation temps-réel qui permet d'en uniformiser l'interface d'appel, en ce qui concerne la gestion des processus, de la mémoire et des interruptions. Les entités manipulées par la couche d'adaptation sont le *segment* (zone de mémoire contigüe de taille fixe), l'*activité* (ou *thread* - unité d'exécution) et le *processus* (ensemble d'activités et segments, localisé sur un calculateur). Le nombre de ces entités est connu statiquement, de manière à ne pas avoir à recourir à l'allocation dynamique de mémoire. Il est calculé par un outil de dimensionnement exploitant les informations connues statiquement sur les tâches.

L'intérêt de la couche d'adaptation est de faciliter le portage du support d'exécution HADES sur différents noyaux temps-réel. À ce jour, un portage a été effectué sur les noyaux temps-réel Chorus et RTEMS et un simulateur a été développé au dessus de Sun/Solaris.

3.5.2.3 Noyau d'exécution (allocateur)

Le niveau intermédiaire du logiciel constituant le support d'exécution est nommé *noyau d'exécution* ou encore *allocateur*.

Le rôle de l'allocateur est de gérer l'exécution de *tâches*, une tâche correspondant au modèle ayant été défini dans le paragraphe 3.2.1 (graphe orienté acyclique dans lequel chaque nœud représente un calcul et chaque arc une contrainte de précédence

entre deux calculs, décoré avec les attributs statiques des tâches, comme par exemple leurs échéances). En particulier, l'allocateur fournit un ordonnanceur *par défaut*, qui est un ordonnanceur préemptif à priorités fixes (il est également possible de définir des ordonnancements différents de l'ordonnancement par défaut par l'intermédiaire de services externes à l'allocateur - voir § 3.5.2.4).

L'allocateur implante comme ordonnanceur par défaut un ordonnancement préemptif à priorités fixes. À chaque nœud appartenant à une tâche est associée une priorité, qui ne change pas pendant son exécution. L'allocateur assure qu'à tout instant, c'est le nœud éligible avec la priorité la plus importante qui est exécuté. Un nœud est éligible si toutes les contraintes conditionnant son exécution (contraintes de précédence, contraintes d'exclusion via l'utilisation de ressources) sont vérifiées. L'ordonnanceur par défaut exploite un ensemble de files d'exécution triées par priorités.

3.5.2.4 Services

La couche la plus haute du support d'exécution HADES est constituée d'un ensemble de services nécessaires à l'exécution de tâches distribuées. Les services sont développés en respectant le modèle de tâches HADES, de manière à pouvoir les intégrer dans l'analyse d'ordonnançabilité du système (voir § 3.2.3.2). Les tâches constituant les services sont gérées par l'ordonnanceur par défaut intégré à l'allocateur. De plus, de manière à assurer des temps de réponse bornés pour le support d'exécution, ces tâches sont plus prioritaires que celles de l'application.

Les principaux services du support d'exécution, schématisés sur la figure 3.11 sont les suivants :

- Services de synchronisation d'horloges, de diffusion fiable et de gestion de groupe. Le rôle de ces services et l'algorithmique sous-jacente sont ceux qui ont été présentés dans le paragraphe 3.4.1.2.
- Service de distribution. Ce service est appelé à chaque action d'exécution nécessitant une communication avec un autre calculateur (e.g. validation d'une contrainte de précédence distribuée). Il s'appuie sur les services de diffusion fiable et de gestion de groupes pour assurer la validation de contraintes de précédence en temps borné, et ceci même en présence de fautes.
- Service d'ordonnancement. Ce service implante un ordonnancement différent de l'ordonnanceur par défaut (ordonnancement à *deux niveaux*) pour les tâches applicatives le désirant. Ce service dispose d'un ensemble prédéfini de priorités à sa disposition, et insère les exécutions de nœuds dans les files associées selon une politique d'ordonnancement qui lui est propre.

Nous avons implanté plusieurs services d'ordonnancement (voir [ACCP99] pour plus de détails). L'un d'entre eux exploite des plans générés statiquement (par exemple par l'algorithme présenté dans le paragraphe 3.2.3.2). La figure 3.12 explicite le fonctionnement du système d'ordonnancement à deux niveaux dans le cas où on utilise cet ordonnanceur, nommé sur la figure *Oplan*.

La figure schématise les nœuds prêts dans les files d'exécution des différentes

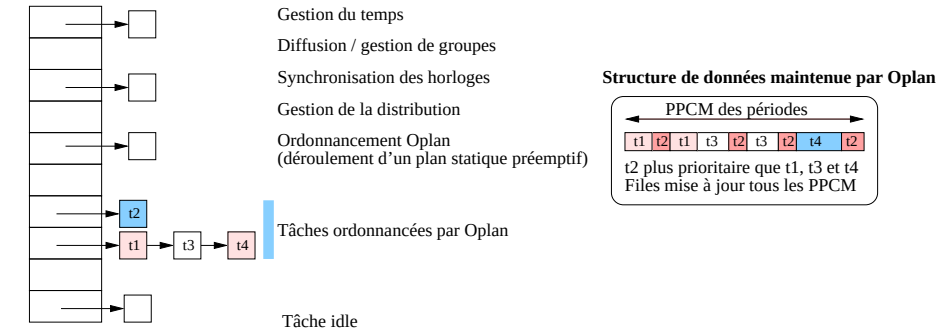


FIG. 3.12 – Système d'ordonnancement à deux niveaux d'HADES

priorités (les nœuds les plus prioritaires apparaissent en haut de la figure). L'ordonnanceur par défaut assure que c'est le nœud éligible de plus haute priorité qui est exécuté à tout instant (dans la figure, le nœud du service de gestion du temps, chargé de mettre à jour périodiquement l'horloge du système et de gérer les chiens de garde, est exécuté). L'ordonnanceur *Oplan* s'exécute à une priorité moindre. Il maintient comme structure de données un plan généré statiquement. Périodiquement (tous les PPCMs de la période des tâches à exécuter) ce plan est utilisé pour mettre à jour les files d'exécution associées à l'ordonnanceur *Oplan*. Plusieurs files d'exécution sont exécutées car *Oplan* déroule un plan contenant des préemptions. Par exemple, comme $t2$ préempte $t1$ dans le plan, il lui est associé une priorité supérieure à celle de $t1$.

3.5.2.5 Outil d'injection de fautes

Le support d'exécution HADES intègre un outil d'injection de fautes permettant d'étudier son comportement en présence de fautes (voir § 3.4.1.1). L'outil d'injection de fautes simule l'occurrence de fautes physiques transitoires. L'injection est effectuée par logiciel, en introduisant la manifestation des fautes (les *erreurs*) dans la mémoire du support d'exécution. Le mode de déclenchement de chaque erreur est spatial et temporel, et concerne toutes les régions du support d'exécution (code, données, pile), injecteur de fautes exclus. Chaque erreur correspond à l'inversion d'un bit dans un octet.

Au terme d'un délai aléatoire, une erreur est injectée dans une des zones citées au paragraphe précédent, et les registres de déboguage du Pentium sont programmés pour détecter l'accès à l'adresse mémoire où l'erreur a été injectée. Au bout d'un certain temps, si cette adresse n'a pas été accédée, l'erreur est supprimée, et une nouvelle erreur est injectée à une autre adresse.

– Si le processeur détecte un accès à cette erreur en *lecture*, alors l'erreur a été *activée*, et l'injecteur de fautes a terminé son travail. Une défaillance peut alors

potentiellement se produire.
- Si le processeur détecte un accès à cette erreur en écriture (l'erreur a donc été injectée dans une donnée), alors l'injecteur de fautes analyse l'instruction qui a accédé à cette erreur pour savoir si l'écriture a été précédée d'une lecture à la même adresse (par exemple addition d'une constante à une donnée pointée par un registre). Si c'est le cas, l'erreur a été activée, et sinon elle a été écrasée par une écriture avant d'être activée. Dans la seconde éventualité, une erreur est injectée de nouveau, et l'exécution se poursuit.

Après qu'une erreur ait été activée, l'exécution se poursuit un certain temps. Si cette erreur est détectée par le support d'exécution HADES, alors des informations sur cette détection sont mémorisées par l'injecteur de fautes, et l'exécution continue. Au terme d'un délai borné après l'activation d'une erreur, et quelles que soient ses conséquences (détection ou non, exception matérielle, aucune manifestation), des informations sur l'expérience en cours sont envoyées sur la machine hôte et la machine cible est redémarrée.

Notons qu'un soin tout particulier a été porté pour limiter l'intrusion de l'injection de fautes. Ainsi, pour déterminer si l'écriture d'une donnée a été précédée d'une lecture, plutôt que de décoder l'instruction qui a accédé à la lecture, nous utilisons une table pré-calculée à l'initialisation du système. D'autre part, les informations sur les erreurs injectées ne sont pas transmises pendant l'exécution, mais juste avant le re-démarrage (reboot) du calculateur (d'où l'intérêt de limiter au maximum la propagation d'une erreur à l'injecteur de fautes afin d'avoir la meilleure confiance possible dans les résultats d'une campagne d'injection de fautes).

3.5.3 Performances de l'outil d'analyse de WCET Heptane

La méthode d'analyse statique de WCET présentée dans le paragraphe 3.3 a été évaluée sur différents programmes : des petits programmes de test, constituant le *SNU real-time benchmarks suite* fourni par l'Université de Séoul (groupe temps-réel), et un code plus conséquent, le code du noyau temps-réel RTEMS [OAR98]. L'intérêt du travail d'analyse de RTEMS est double. D'une part, nous avons expérimenté l'utilisation d'une méthode d'analyse statique de WCET sur du code de *taille conséquente*. À l'heure actuelle, les outils d'analyse statique de WCET sont évalués principalement sur des petits codes numériques. D'autre part, nous avons appliqué notre méthode d'analyse sur du code *système* et non pas utilisateur comme cela est pratiqué usuellement. Ceci permet d'obtenir une évaluation sûre du WCET d'un système d'exploitation. L'utilisation de ce type de méthode est inhabituelle dans le domaine des systèmes d'exploitation temps-réel, pour lesquels les performances sont évaluées grâce à des jeux de tests (*benchmarks*).

Hormis quelques différences sur les types de boucles rencontrés, les résultats de l'analyse de ces deux types d'applications sont similaires. C'est pourquoi nous nous concentrons ici sur les résultats de l'analyse du noyau RTEMS (voir aussi [CP01d] et [CP01b]). On se reportera à l'article [CP01c] pour les résultats de l'analyse sur les petits programmes de test du SNU.

Après avoir brièvement présenté le noyau analysé, nous présentons les résultats qualitatifs et quantitatifs de l'analyse.

3.5.3.1 Le noyau temps-réel RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems) [OAR98] est un noyau temps-réel dont les sources sont publics. RTEMS comporte les éléments suivants :

- des capacités multi-tâches et un ordonnancement préemptif, à base de priorités,
- la communication et la synchronisation entre tâches, par l'intermédiaire de sémaphores, d'événements et de signaux,
- un mécanismes d'héritage de priorités pour éviter les problèmes d'inversions de priorités [SRL90],
- l'allocation dynamique de la mémoire,
- des capacités de configuration, permettant notamment d'augmenter le noyau avec des *fonctions d'extension*, appelées à des points spécifiques de son exécution.

La version de RTEMS que nous avons analysée (4.0.0 pour cible i386 spécialisée pour cible monoprocesseur) offre deux interfaces de programmation : une interface conforme à la norme POSIX, et une interface spécifique RTEMS, sur laquelle nous nous sommes concentrés. Nous avons analysé le cœur du noyau RTEMS, à savoir les 12 appels systèmes clé de gestion des tâches et de synchronisation par sémaphores (voir tableau ci-dessous).

Nom	**Description**
rtems_clock_get	Informations de date et heure du système
rtems_semaphore_create	Création de sémaphore
rtems_semaphore_obtain	Prise de sémaphore
rtems_semaphore_release	Libération de sémaphore
rtems_task_create	Création de tâche
rtems_task_delete	Destruction de tâche
rtems_task_ident	Renvoi de l'identifiant d'une tâche
rtems_task_start	Démarrage d'une tâche
rtems_task_suspend	Suspension d'une tâche
rtems_task_resume	Reprise d'une tâche
rtems_task_set_priority	Affectation d'une priorité à une tâche
rtems_task_wake_after	Réveil d'une tâche après un certain délai

Pour donner une idée du volume de code concerné, les 12 appels systèmes analysés sont répartis dans 91 fichiers source (C, assembleur, fichiers d'inclusion) ; le code source de ces fichiers (hors commentaires) comporte 14532 lignes, et définit 355 fonctions.

3.5.3.2 Résultats de l'analyse : aspects qualitatifs

Nous relatons ici les résultats de notre expérimentation en examinant les problèmes liés aux restrictions au langage C imposées par notre méthode d'analyse

(§ 3.3) rencontrés lors de l'analyse de RTEMS. Nous examinons tour à tour ces différentes restrictions (appels dynamiques de fonctions, graphes de flot de contrôle non structurés, bornes sur le nombre maximum d'itération des boucles insérées via un système d'annotations – § 3.3.3) et pour chacune d'elles montrons leur influence sur l'analyse de RTEMS.

De manière générale, nous avons remarqué lors de l'analyse du code source de RTEMS, l'absence de fonctions récursives. Ainsi, il n'a pas été nécessaire d'introduire un système d'annotations pour identifier le niveau maximum de récursivité. Par ailleurs, contrairement aux petits programmes numériques que nous avons analysés dans l'article [CP01c], aucune boucle emboîtée n'a été identifiée dans le code source de RTEMS.

Appels dynamiques de fonctions. Les appels de fonctions via l'utilisation de pointeurs (ci-après appels *dynamiques* de fonctions) posent problème pour l'analyse statique de WCET car quand de tels appels sont utilisés, la fonction appelée n'est connue que dynamiquement. Parmi les 368 appels de fonctions présents dans le code analysé, nous n'avons trouvé que 15 appels dynamiques. Essentiellement, ces appels dynamiques correspondent à des appels à des *fonctions d'extensions*, qui sont des fonctions pouvant être connectées dynamiquement au noyau RTEMS et appelées à des points spécifiques de son exécution (création de tâche, changement de contexte entre tâches, entrée/sortie). Ces fonctions d'extension permettent, de manière maîtrisée, de modifier le comportement du noyau. Pour rendre RTEMS analysable, nous avons dû imposer à l'utilisateur du noyau une identification *statique* des fonctions d'extension ajoutées. Cette identification peut bien entendu être différente pour deux utilisations différentes du noyau.

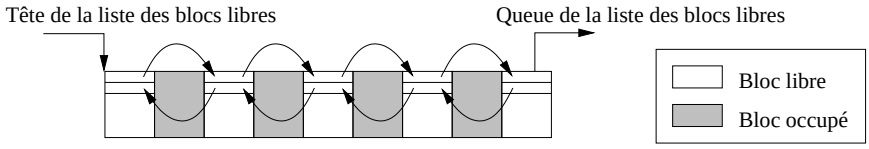
Graphes de flot de contrôle non structurés. Notre méthode d'analyse s'appuyant sur la structure syntaxique des programmes, elle ne supporte pas les branchements autres que ceux introduits par les structures de contrôle du langage (boucles, conditionnelles). De tels branchement peuvent exister dans du code de bas niveau à cause d'inclusions de code assembleur comportant des instructions de branchement, ou encore à cause de l'utilisation de commandes *goto*. La restriction consistant à interdire de tels branchements n'a pas posé de problème majeur lors de l'analyse de RTEMS. En effet, un seul fichier assembleur, dont le code ne contenait pas de branchement, a du être analysé (code de sauvegarde et restauration du contexte). De plus, quatre boucles contenant des constructions *goto* étaient présentes dans le code source, mais ont pu être restructurées sans problèmes.

Borne sur le nombre d'itérations des boucles. L'analyse des boucles est d'une grande importance pour l'analyse statique de WCET, car elle permet d'identifier le pire chemin d'exécution dans le programme. Rappelons que notre méthode d'analyse nécessite une définition manuelle, via un système d'annotations (voir § 3.3.3)

du nombre maximum d'annotation des boucles. Nous avons identifié trois types de boucles lors de l'annotation des boucles de RTEMS.

- *Boucles à nombre d'itérations constant.*
  Dans ces boucles, représentant un quart du nombre total de boucles, le nombre d'itérations figure en clair dans l'en-tête de la boucle. Clairement, le nombre maximum d'itérations de ces boucles pourrait être obtenu automatiquement.
- *Boucles dont le nombre maximum d'itérations dépend du nombre d'objets RTEMS présents dans le système.*
  Ces boucles représentent la moitié des boucles présentes dans RTEMS. Leurs nombres d'itérations dépendent de variables globales telles que par exemple :
  - le nombre maximum d'objets (toutes catégories confondues) existants dans le système à un instant donné,
  - le nombre de tâches en attente sur un sémaphore,
  - le nombre de tâches prêtes à s'exécuter,
  - le nombre d'extensions connectées à un point d'extension donné.

  RTEMS demande que le nombre maximum de tâches, et en général d'objets, soit connu à la compilation du système. De plus, pour supprimer les appels dynamiques de fonction, nous avons imposé de connaître statiquement les fonctions d'extension au noyau. Donc, le nombre maximum d'itérations de toutes les boucles de cette deuxième catégorie a pu être borné.
  En revanche, les bornes qui ont été établies peuvent être pessimistes et mériteraient d'être réduites. Par exemple, au pire, le nombre maximum de tâches en attente sur un sémaphore a été fixé au nombre maximum de tâches. Établir des bornes plus précises sur ces valeurs nécessiterait une connaissance de l'application en cours d'exécution.
- *Boucles dont le nombre d'itérations dépend du contexte d'exécution courant.*
  Ces boucles représentent un quart du nombre total de boucles, et leur nombre maximum d'itérations a été difficile à borner de manière précise. Prenons deux exemples de telles boucles.
  Le premier exemple est une boucle située dans l'algorithme d'allocation dynamique de mémoire de RTEMS, utilisée lors de la création des tâches et de leurs suppressions afin d'allouer et de libérer leur pile. RTEMS utilise la stratégie d'allocation de mémoire *first-fit* avec une taille minimum des blocs alloués de *MIN_BLOCK_SIZE*. Les blocs libres sont gérés par une liste doublement chaînée. La fonction d'allocation de mémoire contient une boucle qui effectue une recherche sur cette liste de blocs libres jusqu'à ce qu'un bloc dont la taille est au moins celle demandée soit trouvé. Dans le pire cas (fragmentation mémoire maximale), la mémoire est constituée d'une succession de blocs libres et de blocs occupés de taille *MIN_BLOCK_SIZE* (voir la figure ci-dessous). Le pire nombre d'itérations de cette boucle est alors *HEAP_SIZE* / (2 * *MIN_BLOCK_SIZE*), où *HEAP_SIZE* est la taille de la zone mémoire consacrée à l'allocation dynamique.



Bien que l'on ait pu exhiber une borne supérieure pour le nombre d'itérations de cette boucle, force est de constater que cette estimation est très pessimiste. L'établissement d'une borne moins pessimiste nécessiterait de connaître statiquement la pire fragmentation mémoire possible pour une application donnée. Un deuxième exemple de boucle, dont le nombre maximum d'itérations dépend du contexte courant d'exécution, est la boucle principale de l'ordonnanceur. Le nombre d'itérations de cette boucle dépend du nombre d'interruptions susceptibles de produire un changement de contexte pouvant être déclenchées dans un intervalle de temps donné. Borner le nombre d'itérations de cette boucle nécessite de connaître dans le pire des cas le nombre d'interruptions pouvant se produire pendant cet intervalle, ce qui est dépendant du contexte d'exécution du programme.

Pour les deux derniers types de boucles, qui représentent les 3/4 des boucles présentes dans le code analysé, des méthodes automatiques n'auraient pu être utilisées pour détecter automatiquement leur nombre maximum d'itérations. L'utilisation d'une méthode d'annotations, bien que contraignante et sujette aux erreurs, reste ici incontournable.

3.5.3.3 Résultats de l'analyse : aspects quantitatifs

Ce paragraphe présente de manière quantitative les résultats de l'analyse du noyau RTEMS. Nous comparons pour chaque appel système son temps d'exécution obtenu par analyse statique de WCET avec son WCET réel. Ce dernier est obtenu en mesurant le temps d'exécution de chaque appel système dans son pire scénario d'exécution. Nous avons identifié ce pire scénario d'exécution grâce à une étude approfondie du code source du noyau, qui nous a permis d'avoir une connaissance fine (quoique sujette aux erreurs) des différents algorithmes utilisés dans le code.

Par exemple, pour l'appel système de création de tâche *rtems_task_create*, qui utilise l'allocation dynamique de mémoire, le pire scénario d'exécution de cet appel se produit quand l'algorithme d'allocation dynamique prend le plus de temps à s'exécuter, c'est à dire quand la mémoire est fragmentée (voir § 3.5.3.2). Pour obtenir le WCET réel de cet appel système, nous avons donc fragmenté la mémoire avant de l'appeler.

Les mécanismes internes du Pentium (registres de décompte d'événements) nous ont permis de mesurer les temps d'exécution (nombre de cycles). De plus, comme l'outil HEPTANE n'intègre pas la gestion du cache de données et du pipeline entier *double* du Pentium, ces deux éléments ont été désactivés pour effectuer les mesures, de manière à pouvoir comparer des grandeurs comparables.

Résultats globaux. Le tableau 3.5 confronte les WCETs des douze directives obtenues par analyse statique (colonne *"est"*) avec les WCETs mesurés (colonne *"exec"*). Les résultats sont exprimés en nombre de cycles sur un processeur Pentium 90 MHz.

	WCET		
Appel système RTEMS	*est*	*exec*	ratio
rtems_clock_get	1420	1302	1.09
rtems_semaphore_create	27050	11878	2.28
rtems_semaphore_obtain	288484	97417	2.96
rtems_semaphore_release	40414	17025	2.37
rtems_task_create	435550	309618	1.41
rtems_task_delete	72679	47275	1.54
rtems_task_ident	150093	98439	1.52
rtems_task_resume	19045	11521	1.65
rtems_task_set_priority	24728	20179	1.23
rtems_task_start	138940	36532	3.80
rtems_task_suspend	20106	16237	1.24
rtems_task_wake_after	19823	19923	1.17
Moyenne			1.86

TAB. 3.5 – Confrontation entre résultats d'analyse et résultats expérimentaux

On remarque que le ratio entre le nombre d'instructions exécutées et obtenues par analyse est toujours supérieur à 1, ce qui montre (sur l'exemple) que l'analyse est sûre. Le rapport entre les WCET estimés et obtenus par exécution est en moyenne de 1.86, l'idéal étant une valeur de un, montrant que l'analyse de WCET a réussi à déterminer de manière exacte le pire temps d'exécution. Cette valeur de 1.86 ne nous semble pas excessivement pessimiste, d'autant plus qu'il nous semble possible de réduire encore le pessimisme, comme indiqué ci-dessous.

Une analyse approfondie du code source du noyau nous a permis d'identifier deux principales sources de pessimisme. La première est la présence de chemins d'exécutions *infaisables*. Par exemple, la fonction *Thread_Dispatch*, qui effectue le changement de contexte apparaît plusieurs fois dans le graphe d'appel de certains appels système (par exemple *rtems_semaphore_obtain*) alors qu'elle n'est appelée qu'une seule fois. La deuxième source de pessimisme provient de quelques fonctions qui peuvent être appelées aussi bien par l'utilisateur que par le noyau RTEMS lui-même. Dans ce deuxième cas, il s'avère que la majorité des paramètres de ces fonctions est constante, et donc seulement un sous-ensemble de leur code est utilisé. D'où une surestimation du WCET des fonctions due à la prise en compte de chemins d'exécution impossibles lors du calcul du WCET. Nous étudions actuellement la possibilité d'éliminer ces sources de pessimisme en appliquant une technique d'évaluation partielle sur le code source de RTEMS (voir chapitre 4 donnant les perspectives ouvertes par notre travail).

Apports de la modélisation de microarchitecture. Un aspect important à évaluer est l'apport en terme de précision de l'évaluation des WCETs de la prise en compte de la microarchitecture. Nous avons évalué cet apport en comparant les

	I	II	III	IV	V	VI	VII
Appel système	*exec.*	*est. model*	*sans btb*	*sans icache*	*sans pipeline*	*est. sans model*	$\frac{est.\ ss\ model}{est\ model}$
rtems_clock_get	1302	1420	1536	3189	7617	9504	6.69
rtems_semaphore_create	11878	27050	29395	46443	77649	183945	6.80
rtems_semaphore_obtain	97417	288484	312852	573408	930473	1834782	6.36
rtems_semaphore_release	17025	40414	43473	82791	160809	290607	7.19
rtems_task_create	309618	435550	471981	1032372	1403786	2949511	6.77
rtems_task_delete	47275	72679	78491	140794	235861	462594	6.36
rtems_task_ident	98439	150093	162041	330119	461517	965088	6.43
rtems_task_resume	11521	19045	20551	43736	52525	118268	6.21
rtems_task_set_priority	20179	24728	26842	58638	83827	169876	6.87
rtems_task_start	36532	138940	150052	269154	579978	1007453	7.25
rtems_task_suspend	16237	20106	21707	44222	59783	126654	6.30
rtems_task_wake_after	16923	19823	21391	45523	53613	120627	6.09
Moyenne							6.61

TAB. 3.6 – Apports de la prise en compte de la micro-architecture

WCETs obtenus par HEPTANE en prenant en compte tous les éléments architecturaux avec les WCETs obtenus en configurant HEPTANE de manière à ne pas inclure la prise en compte de la micro-architecture.

Les résultats sont regroupés dans le tableau 3.6. La colonne I donne le WCET mesuré, tandis que la colonne II donne le WCET estimé avec prise en compte de l'architecture. Les colonnes III, IV et V donnent l'estimation du WCET en désactivant soit la prise en compte de la prédiction de branchement (III), soit celle du cache d'instructions (IV) soit du pipeline (V). La colonne VI donne l'estimation du WCET sans modélisation de micro-architecture. Enfin la colonne VII confronte les estimations de WCET obtenues par analyse statique, avec et sans prise en compte de la micro-architecture.

Les résultats contenus dans le tableau (principalement la colonne VII) montrent que la prise en compte de la microarchitecture est incontournable (la non prise en compte de la microarchitecture multiplie les WCETs par un facteur moyen de 6.6).

Le graphique de la figure 3.13 présente les résultats du tableau 3.6 sous une forme différente. La première portion de chaque histogramme (celle du bas, correspondant à la colonne I du tableau) est le WCET mesuré. La portion suivante (II-I) est le pessimisme de l'analyse (cache d'instructions, pipeline et prédiction de branchement considérés) par rapport au WCET mesuré. Les trois portions suivantes représentent respectivement les pessimismes des analyses sans prise en compte de la prédiction de branchement (III-II), du cache d'instructions (IV-II) et du pipeline (V-II) par rapport à une analyse qui prend en compte ces trois éléments.

On remarque que les gains les plus importants sont obtenus par la prise en compte

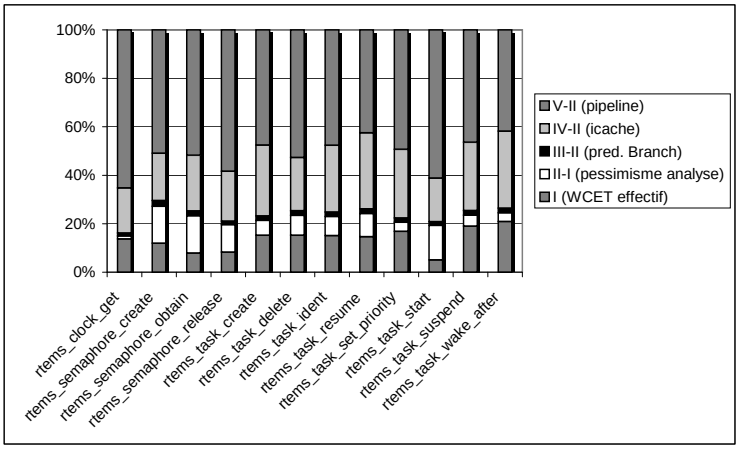


FIG. 3.13 – Performances comparées des mécanismes de prise en compte de la micro-architecture

du pipeline, puis du cache d'instructions et de la prédiction de branchement. Les gains obtenus en modélisant la prédiction de branchement sont moins importants que pour les deux autres éléments architecturaux car ils ne concernent qu'un sous-ensemble des instructions.

3.5.4 Performances du support d'exécution HADES

Ce paragraphe est consacré à une analyse quantitative des performances du support d'exécution HADES. Nous examinons dans le paragraphe 3.5.4.2 le comportement du support d'exécution en présence de fautes. Les paragraphes 3.5.4.3 et 3.5.4.4 sont consacrés au comportement temporel du support d'exécution. Ils examinent respectivement les performances au pire cas des protocoles inclus dans HADES et le taux d'occupation du processeur induit par la présence de ces protocoles. Toutes les mesures de performances données dans ce paragraphe ont été obtenues en sollicitant le support d'exécution par une application de charge, décrite dans le paragraphe 3.5.4.1.

De plus amples détails sur les performances du support d'exécution, notamment ses performances sous différentes charges, peuvent être trouvés dans l'article [CPC+01].

3.5.4.1 Application de charge

L'application utilisée pour analyser les performances du support d'exécution HADES est une application de charge synthétique. Cette application a été conçue de manière à utiliser l'ensemble des fonctionalités du support d'exécution. Son exécution est déterministe et les résultats de son exécution connus, de manière à pouvoir vérifier

simplement et rapidement la correction de ses résultats.

L'application de charge est la partie logicielle d'une application de suivi d'objet. L'objectif est pour un ensemble d'objets esclaves de suivre la trajectoire d'un objet maître. La position des objets est donnée par des capteurs, et leur déplacement géré par des actionneurs (ici, les capteurs et actionneurs sont émulés par logiciel). Le calcul des trajectoires des esclaves est réalisé par une tâche distribuée sur les quatre machines. Cette tâche est exécutée périodiquement toutes les 250 ms, et son échéance est 250 ms. Pour pouvoir aisément déterminer la correction du déroulement de l'application de charge, l'objet maître est déplacé selon une trajectoire prédéfinie.

3.5.4.2 Analyse du comportement du support d'exécution en présence de fautes

Le comportement du support d'exécution en présence de fautes (version portée sur le noyau Chorus) a été évalué en utilisant une méthode d'injection de fautes par logiciel (voir § 3.4.1.1). L'évaluation a porté sur un ensemble de métriques : couverture de l'hypothèse de silence sur défaillances, efficacités comparées des mécanismes de détection d'erreurs, latence et coûts de la détection d'erreurs, recouvrement entre mécanismes de détection d'erreurs. Nous donnons ci-dessous les principaux résultats de l'évaluation (une description complète de ces résultats pourra être trouvée dans [CP01a]).

Couverture de l'hypothèse de silence sur défaillance. Pour calculer la couverture du silence sur défaillance, des fautes sont injectées dans la mémoire du support d'exécution. Nous considérons uniquement les fautes *activées*, c'est-à-dire les fautes pour lesquelles le mot corrompu par l'injecteur de fautes a été utilisé par le processeur. Une faute activée peut aboutir aux manifestions suivantes sur les résultats de l'application de charge :

- STOP : Pas de résultats (arrêt d'au moins un calculateur).
- INCORRECT : Résultats incorrects (valeur et/ou instant de production du résultat incorrects).
- CORRECT : Résultats corrects dans le domaine des valeurs *et* le domaine temporel.

Les deux premières classes de manifestations données ci-dessus peuvent être subdivisées en deux, selon que l'erreur s'est propagée ou non sur un autre calculateur (les fautes sont injectées dans la mémoire de l'unique calculateur dont on désire évaluer le silence sur défaillance). On appelle par la suite les catégories résultantes STOP-NOPROPAG, STOP-PROPAG, INCORRECT-NOPROPAG, INCORRECT-PROPAG et CORRECT. Un calculateur est silencieux sur défaillance si les résultats de l'application de charge sont corrects (catégorie CORRECT) ou s'il s'arrête sans propager d'erreur (catégorie STOP-NOPROPAG). La couverture est calculée sur l'ensemble des erreurs activées, de cardinal noté N_{act} :

$$coverage_{fail-silence} = \frac{(N_{CORRECT} + N_{STOP-NOPROPAG})}{N_{act}}$$

Le tableau 3.7 donne les manifestations observées par injection de fautes, ainsi que les couvertures de silence sur défaillance correspondantes. Deux configurations de HADES sont distinguées : (a) sans mécanismes de détection d'erreurs, c'est-à-dire avec uniquement CPU et les paniques noyau qui ne peuvent pas être désactivées (on a alors le mode de défaillance intrinsèque du système d'exploitation utilisé) ; (b) avec tous les mécanismes de détection d'erreurs du § 3.4.1.1. Dans le tableau 3.7 et dans la suite de ce paragraphe, nous utilisons l'abbréviation MDE pour nommer un mécanisme de détection d'erreurs.

Manifestation	**(a) Pas de MDE**		**(b) Tous les MDEs**	
Fautes injectées	39025		29591	
Fautes activées	3152		3012	
CORRECT	1839	(58.3%)	1122	(37.3%)
STOP-NOPROPAG	703	(22.3%)	1862	(61.8%)
STOP-PROPAG	102	(3.2%)	22	(0.7%)
INCORRECT-NOPROPAG	33	(1.1%)	2	(0.1%)
INCORRECT-PROPAG	475	(15.1%)	4	(0.1%)
Couverture silence sur défaillance	80.64 %		99.07 %	
Intervalle de confiance (niveau de confiance de 95 %)	[79.49 %, 81.79 %]		[98.80 %, 99.33 %]	

TAB. 3.7 – Couverture du silence sur défaillance

Quand aucun MDE n'est utilisé, 19.4% des fautes activées aboutissent à une violation du silence sur défaillance, ce qui montre que sans MDE, le système n'est clairement pas silencieux sur défaillance. Quand tous les mécanismes de détection d'erreurs sont utilisés, 99.1% des fautes activées aboutissent à un comportement silencieux sur défaillance. 99.1% est indéniablement plus faible que les taux obtenus dans des systèmes où du matériel dédié à la tolérance aux fautes est utilisé (voir par exemple [Fuc96]). Néanmoins, les MDEs intégrés à HADES sont efficaces, car ils divisent le nombre de violations du silence sur défaillance par un facteur 22.

Le tableau 3.7 montre qu'un grand pourcentage de violations du silence sur défaillance provient de propagations d'erreurs entre calculateurs. Un examen détaillé des expériences ayant produit ces violations [CP01a] nous a amené au constat suivant. Dans un large pourcentage des expériences (90%), l'application n'a pas fourni de résultat incorrect (l'erreur a été détectée sur un calculateur distant avant de se propager à l'application). Bien que dans ce cas il y ait eu propagation de l'erreur et donc violation du silence sur défaillance, ce type d'erreur peut être traité par certains mécanismes de détection d'erreurs, car il ne correspond ni à une faute par valeur ni à une faute temporelle.

Par ailleurs, nous avons examiné en détails les différents cas de violation du silence sur défaillance et pour chacun d'eux nous ne voyons pas comment éviter la propagation de l'erreur par logiciel uniquement, à un coût (en temps de calcul et occupation

mémoire) raisonnable, sans utiliser de matériel spécifique. Nous pensons donc nous être approchés de la meilleure couverture du silence sur défaillance atteignable sans utiliser de matériel spécifique.

Notons que la plupart des mécanismes de détection d'erreurs intégrés à HADES sont utilisables dans d'autres supports d'exécution, même s'ils ne sont pas dédiés au temps-réel, car on y trouve les mêmes types de données (par exemple des files d'attentes pour l'ordonnanceur) et de protocoles (communication, allocation mémoire). On peut donc présumer que l'utilisation dans un autre support d'exécution des mêmes mécanismes de détection d'erreurs que dans HADES aboutirait à une couverture du silence sur défaillance comparable. En revanche, il est délicat d'estimer l'impact de l'utilisation de ces mécanismes de détection d'erreurs sur d'autres types de logiciels.

Efficacité des mécanismes de détection d'erreurs. La figure 3.14 montre, sur un ensemble d'expérience E le nombre de fois où ce mécanisme a été le premier à détecter une erreur. Nous nommons cette métrique *pourcentage de premières détections*[5].

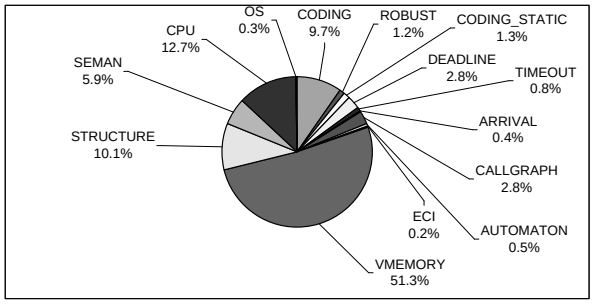


FIG. 3.14 – Pourcentage de premières détections des MDEs

La figure montre que la majorité des erreurs (51.3%) est détectée par le mécanisme VMEMORY. Un pourcentage plus faible d'erreurs est détecté par les MDEs CODING, STRUCTURE et CPU (approximativement 10% chaque). L'efficacité de STRUCTURE est due au fait que le support d'exécution HADES utilise de manière intensive les listes et les tableaux. Le mécanisme CALLGRAPH détecte environ 3% des erreurs. Les autres mécanismes, et en particulier ceux qui contrôlent les temps d'exécution des tâches sont très peu efficaces puisqu'ils détectent moins de 2.5% d'erreurs.

[5]Cette métrique a été évaluée sur l'ensemble E d'expériences dans lesquelles une erreur a été détectée par un MDE m de manière justifiée, c'est-à-dire que m a détecté une erreur et que cette erreur a abouti à une défaillance du système. Cette notion de détection justifiée, ainsi que ses conséquences sur l'évaluation de l'efficacité des mécanismes de détection d'erreurs, est explicitée dans [CP01a]

Coût des mécanismes de détection d'erreurs. La figure 3.15 détaille les coûts en espace et en temps d'exécution des MDEs. Ces coûts ont été mesurés en ajoutant les MDEs un par un, par ordre décroissant de pourcentages de premières détections (voir figure 3.14). Le coût mémoire du noyau Chorus (196Ko à 191Ko selon que l'on utilise ou non des espaces d'adressage séparés) ne sont pas indiqués sur la figure 3.15.

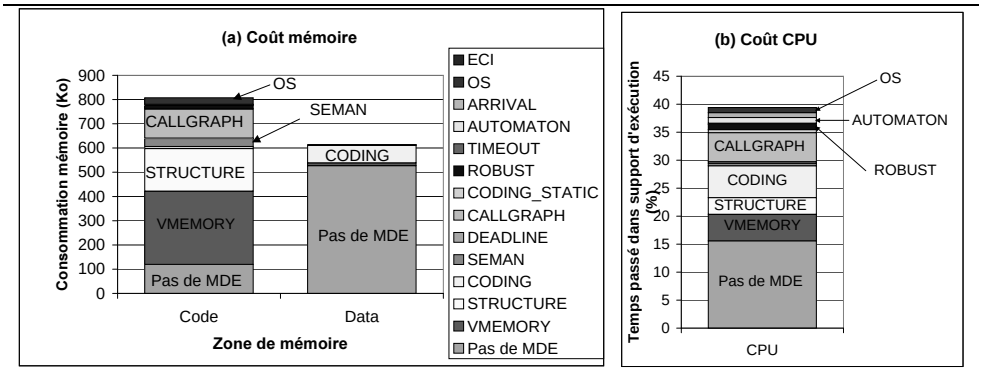


FIG. 3.15 – Coûts mémoire et CPU des MDEs

La partie gauche de la figure détaille l'occupation mémoire additionnelle occasionnée par l'ajout de chaque MDE par rapport à un support d'exécution sans mécanisme de détection d'erreurs. La taille de code du support d'exécution (Chorus exclus) est multipliée par un facteur 6.7 lors de l'introduction de tous les MDEs. Le mécanisme VMEMORY multiplie à lui seul le volume de code par un facteur 3.5. Ceci s'explique par la duplication du code de certaines bibliothèques en mémoire (e.g. libc), cette duplication étant due au fait que la version de Chorus que nous utilisons n'offre pas de support direct pour les librairies partagées. Le mécanisme STRUCTURE ajoute également un surcoût en espace significatif. Ce surcoût provient du fait de l'utilisation massive de fonctions C *inline* dans la gestion des structures de données pour des raisons de performances. Le mécanisme CALLGRAPH augmente la taille du code de manière non négligeable. Cette augmentation est due à la présence dans HADES d'un grand nombre de fonctions de petites tailles, introduites pour des raisons de modularité. Concernant l'augmentation de la taille des données, seul le mécanisme CODING occasionne une augmentation perceptible du volume de données. Tous les autres mécanismes utilisent des variables allouées en pile, ce qui n'a pas d'impact sur la taille de la zone de données du support d'exécution.

Le surcoût en temps des MDEs (pourcentage de temps passé à s'exécuter dans le support d'exécution) est donné sur la partie droite de la figure 3.15. Le temps total passé dans le support d'exécution est multiplié par un facteur 2.5 par l'introduction de mécanismes de détection d'erreurs. Les MDEs les plus consommateurs de

temps CPU sont, par ordre décroissant de consommation CPU : CODING (14.4%), CALLGRAPH (13.1%), VMEMORY (12.0%) et STRUCTURE (7.5%). Les autres MDEs ont un coût inférieur à 3%. Si l'on met en regard le pourcentage de premières détections des mécanismes et leur coût processeur, les mécanismes SEMAN, VMEMORY, DEADLINE, TIMEOUT, et STRUCTURE offrent le meilleur compromis efficacité/coût.

Un facteur 2.5 est à notre avis plutôt satisfaisant pour des MDEs entièrement implantés par logiciel. Une autre solution aurait été d'utiliser de la redondance temporelle pour les activités du support d'exécution (les exécuter successivement plusieurs fois et comparer leurs résultats pour détecter les erreurs). Néanmoins l'utilisation de redondance temporelle pour des activités système est non triviale à implanter, les résultats de ce type d'activités étant principalement des effets de bord (entrées/sorties, modification de structures de données comme les files d'exécution). Par ailleurs, nous ne sommes pas convaincus que le coût induit par l'utilisation d'une telle technique soit moins important que celui de nos mécanismes, à cause de la difficulté pour comparer les résultats des exécutions successives.

Afin de diminuer le coût induit par la détection d'erreurs, nous avons analysé les redondances entre MDEs. Nous avons ainsi montré qu'en éliminant certains d'entre eux, il est possible de diminuer les coûts en mémoire et processeur (x1.1 et x1.7 respectivement) avec seulement une faible diminution de la couverture du silence sur défaillances (-0.8%). Plus de détails sur ces résultats pourront être trouvés dans [CP01a].

3.5.4.3 Performances au pire cas des protocoles

Les procotoles intégrés dans le support d'exécution HADES (synchronisation des horloges, diffusion fiable, gestion de groupes) ont été conçus de manière à ce qu'ils aient un comportement temporel au pire cas *garanti*, et ce même en présence de fautes (arrêts de calculateurs, omissions sur le réseau). Ce paragraphe est dédié à une étude des performances au pire cas de ces trois protocoles, caractérisés par les trois grandeurs suivantes :

- précision de l'algorithme de synchronisation des horloges, noté γ,
- latence de diffusion d'un message, noté δ_{mcast} (temps au pire cas entre l'envoi d'un message sur un calculateur et sa livraison sur le calculateur récepteur),
- latence de détection d'un arrêt/redémarrage de calculateur, noté δ_{detect} (temps au pire cas entre un arrêt ou redémarrage de calculateur et la réception d'une notification de cet arrêt sur un calculateur quelconque).

Avoir des valeurs numériques faibles pour ces trois grandeurs est très important, car de ces valeurs dépend l'ordonnançabilité globale du système (voir § 3.2.3).

Le tableau 3.8 contient les formules donnant les performances au pire cas des trois protocoles (voir [Che99] pour plus de détails concernant ces formules). Dans le tableau, Δ_{trans} est la latence maximum du réseau au niveau physique. δ et ϵ définissent la latence du réseau, coûts du logiciel inclus : la latence de communication appartient à l'intervalle $[\delta - \epsilon, \delta + \epsilon]$. ρ est la dérive maximale des horloges. P_{com} est la période de la tâche mettant en œuvre le protocole de diffusion fiable, et $P_{retrans}$ est l'inter-

Métrique	Formule	NoSW NoSW (ms)	SW (ms)
Précision synchro. horloges (γ)	$5\epsilon+4\rho(\delta+\rho J_t)+4\rho R+2\rho J_t(1+\rho)$, avec $J_t = wP_{retrans}+r_{sync}$	6,09	7,18
Latence diffusion (δ_{mcast})	$(3w+4)P_{com}+\Delta_{trans}+2r_{com}$	14,17	17,41
Latence gestion groupe (δ_{detect})	$(w+2)P_{com}+\Delta_{trans}+r_{com}$	8,17	9,79

TAB. 3.8 – Performance au pire cas des protocoles HADES

valle entre deux retransmissions de messages par l'algorithme de synchronisation des horloges. Enfin, r_{sync} et r_{com} sont les temps de réponse des tâches mettant en œuvre les algorithmes de synchronisation des horloges et de diffusion (délai entre arrivée de la tâche – début de sa période – et la fin de son exécution).

Les deux colonnes de droite du tableau 3.8 donnent des valeurs numériques pour les performances au pire cas des trois protocoles, avec (par construction) $P_{com} = P_{retrans} = 2ms$, $R = 1s$, $w = 1$. D'après les caractéristiques du matériel utilisé, $\rho = 10^{-5}$ et $\Delta_{trans} = 166\mu s$.

Deux applications numériques des formules sont données : une qui ignore les coûts logiciels, c'est-à-dire les valeurs de r_{com} et r_{sync} (colonne NoSW) et l'autre intégrant ces coûts logiciels (colonne SW). Les valeurs de r_{sync} et r_{com} ont été mesurées sur le prototype en prenant les pire valeurs observées lors de l'exécution de l'application de charge (r_{sync} et $r_{com} = 1.6ms$).

Deux conclusions peuvent être tirées de ce tableau de performances.

- La première est que, sans surprise, les performances des protocoles sont moins bonnes que si ces protocoles étaient intégrés au niveau matériel. La différence de performance la plus flagrante est au niveau de la précision de synchronisation des horloges : en utilisant un support matériel pour la synchronisation des horloges, comme par exemple dans TTP/C [Gmb99], on arrive à une précision de 1 à 10 μs (250 μs pour le réseau Byteflight [BPG00]), contre quelques millisecondes dans notre cas.
- Une deuxième conclusion est que le coût du logiciel (piles de protocole) ne peut dans aucun cas être négligé par rapport aux latences imposées par le matériel. Dans les protocoles du support d'exécution HADES, ils représentent en moyenne 20% de la performance au pire cas du protocole.

Influence de la durée entre deux interruptions d'horloge. Les protocoles intégrés dans HADES sont entièrement implantés par logiciel, sous la forme de tâches périodiques. Par construction, dans le support d'exécution HADES, comme de nombreux supports d'exécution temps réel, un sablier (*timer*) est utilisé pour générer périodiquement des interruptions (*ticks* ou *interruptions* d'horloges). Le traitement d'une de ces interruptions met à jour l'horloge locale du système et lance l'ordonnanceur, qui à son tour est chargé de lancer les tâches applicatives. Une conséquence de ce mode de fonctionnement est que la période la plus courte pour les tâches est l'intervalle séparant deux interruptions d'horloge successives. La figure 3.16 montre l'impact de l'intervalle entre deux interruptions d'horloge successives sur les perfor-

mances au pire cas des protocoles intégrés à HADES. Dans la figure, on prend comme période pour les protocoles de diffusion et de gestion de groupe la plus petite valeur possible (intervalle entre deux ticks).

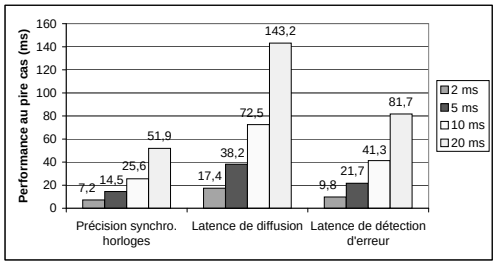


FIG. 3.16 – Influence de l'intervalle entre deux interruptions d'horloge sur les performances au pire cas des protocoles

La figure montre que plus l'intervalle est petit, meilleures sont les performances des trois protocoles. Les performances varient de manière linéaire avec l'intervalle entre deux ticks d'horloge. Cet intervalle a une influence prépondérante sur les performances au pire cas des protocoles.

Influence des mécanismes de détection d'erreurs. La figure 3.17 montre l'influence de l'introduction de mécanismes de détection d'erreurs (voir § 3.4.1.1) sur les performances au pire cas des trois protocoles, pour un intervalle entre deux ticks d'horloge de $5ms$.

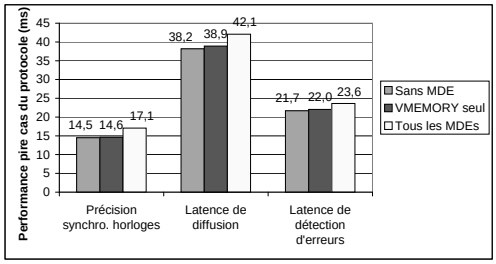


FIG. 3.17 – Influence des mécanismes de détection d'erreurs sur les performances au pire cas des protocoles

La figure montre que les performances sont, bien entendu, moins bonnes avec mécanismes de détection d'erreurs que sans, mais l'introduction de mécanismes de

détection d'erreurs a moins d'impact sur les performances que le changement de l'intervalle entre deux ticks d'horloge.

3.5.4.4 Taux d'occupation du processeur

La deuxième métrique que nous avons utilisée pour appréhender les performances temporelles du support d'exécution est le pourcentage de temps processeur passé à l'intérieur du support d'exécution (taux d'occupation du processeur). Connaître le taux d'occupation du processeur par le support d'exécution est important, car plus ce taux est élevé, moins de tâches applicatives pourront être acceptées par le test d'ordonnançabilité. Cette métrique inluence donc également l'ordonnançabilité du système.

Le taux d'occupation du processeur a été obtenu par instrumentation du support d'exécution (ajout de points de lecture du temps lorsque le processeur est attribué/requisitionné à une des tâches du support d'exécution), en exécutant une application de charge.

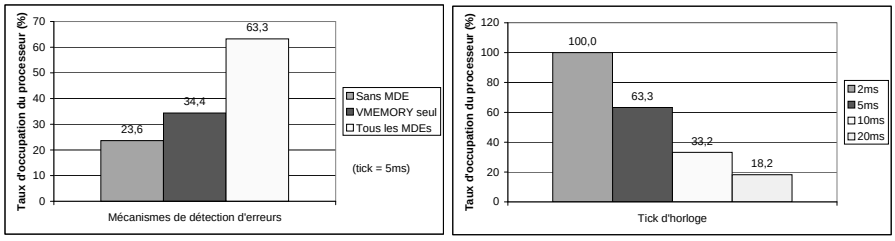


FIG. 3.18 – Taux d'occupation du processeur

La partie gauche de la figure 3.18 montre, pour un intervalle entre deux interruptions d'horloge de $5ms$, l'influence de l'introduction de mécanismes de détection d'erreurs. On observe que le temps passé dans le support d'exécution est augmenté d'un facteur 2.5 par l'introduction de mécanismes de détection d'erreurs.

La partie droite de la figure montre la variation du taux d'occupation du processeur en fonction de l'intervalle entre deux interruptions d'horloge. On observe sur cette partie de la figure que bien que les performances des protocoles soient meilleures pour des intervalles de courte durée, on observe des taux d'occupation du processeur très importants, ce qui diminue de manière significative le temps processeur disponible aux applications. Ainsi, pour un intervalle de $2ms$ entre deux interruptions d'horloge, le système est surchargé (il ne reste plus de temps disponible pour l'exécution des applications, car 100% du temps est passé dans le support d'exécution).

Ces chiffres peuvent sembler au premier abord très élevés, mais remarquons qu'ils sont directement influencés par la vitesse du processeur utilisé (ici un Pentium 90MHz). Ainsi, une occupation processeur de 63% correspondrait à un taux

d'occupation d'environ 6% sur un processeur à 1 GHz. On pourrait donc en utilisant un processeur plus rapide descendre l'intervalle entre deux interruptions d'horloge largement au dessous de la milliseconde, et ainsi améliorer les performances pire cas des protocoles inclus dans HADES de manière significative.

3.6 Discussion

Nous avons proposé un ensemble de méthodes et d'outils pour concilier temps-réel et tolérance aux fautes lors de l'exécution d'applications temps-réel critiques.

Un point original dans notre démarche a consisté à intégrer les mécanismes de tolérance aux fautes (notamment le recouvrement des erreurs) dès la phase de conception des applications, et a donc consisté à considérer de manière fortement couplée temps-réel et tolérance aux fautes. La démarche classique est de concevoir de manière *séparée* les mécanismes de tolérance aux fautes puis de s'intéresser au respect des échéances (voir par exemple [HS94, OS93, GMM97, WBDP98]). L'intérêt de notre démarche est de pouvoir intégrer facilement, par construction, le coût des mécanismes de tolérance aux fautes dans l'analyse d'ordonnançabilité du système, et ce indépendamment de la méthode utilisée pour le recouvrement d'erreurs. De plus, cette manière de procéder rend l'introduction de mécanismes de recouvrement d'erreurs *automatisable* et *sélectif* (seules les parties de l'application qui sont jugées critiques sont dupliquées). Par ailleurs, il est possible d'*adapter* la méthode de recouvrement d'erreurs aux besoins de l'application en sélectionnant la méthode de recouvrement d'erreurs la plus appropriée, voire en intégrant des méthodes de recouvrement d'erreurs autres que celles proposées. D'autres travaux que les nôtres utilisent la démarche de transformation de la structure des applications pour y intégrer des mécanismes de recouvrement d'erreurs (e.g. [Mos93, GLSS01]). Outre quelques différences sur les modèles de fautes considérés, nos travaux se distinguent par le caractère sélectif et adaptable de l'introduction de recouvrement d'erreurs (dans [Mos93] et [GLSS01], une méthode de recouvrement d'erreurs unique est utilisée pour l'ensemble des tâches), ainsi que par le traitement des problèmes de déterminisme des calculs dupliqués.

Concernant les briques de base nécessaires à l'application de méthodes d'analyse d'ordonnançabilité, nous avons proposé une méthode d'analyse statique de programmes permettant à partir d'un programme source d'obtenir son pire temps d'exécution sur une architecture complexe dotée de cache, pipeline et prédicteur de branchement. De nombreux travaux sur l'analyse statique de WCET ont été réalisés dans la dernière décennie (voir [PB00] pour une vue d'ensemble). Ils concernent principalement la réduction du pessimisme de l'analyse : analyse de haut niveau [EG97, HAM+99, EES+99, FMW97, LM95] ou de bas niveau [Mue00, LMW96, EE99, HAM+99]. Nos principales contributions dans le domaine sont (i) la diminution du pessimisme par une méthode d'annotation symbolique des boucles ; (ii) la diminution du pessimisme de l'analyse par prise en compte du mécanisme de prédiction de branchement, et (iii) la prise en compte *simultanée* de plusieurs éléments architecturaux (cache d'instructions, pipeline, prédiction de branchement). L'expérimentation de la méthode proposée, via la construction d'un logiciel d'analyse prototype, a montré que les restrictions imposées par la méthode d'analyse au niveau du programme source ne sont pas excessivement contraignantes, puisque la méthode d'analyse a pu être utilisée avec succès sur un code complexe et relativement bas niveau. Les expérimentations ont également montré que les temps obtenus par analyse sont relativement proches des pires temps d'exécution (facteur moyen de 1.8). La modélisation

de micro-architecture intégrée à l'analyseur est absolument incontournable, car son absence multiplie par un facteur supérieur à 6 les temps au pire cas obtenus par la méthode d'analyse, facteur de surestimation clairement déraisonnable. Soulignons que la méthode d'analyse nécessite à la fois un accès au code source des programmes, ainsi que des informations précises sur l'architecture interne du processeur pour connaître leur comportement temporel. D'autres méthodes doivent être trouvées si de telles informations ne sont pas disponibles. Nous revenons sur ce point dans le chapitre 4 donnant quelques perspectives à notre travail.

Afin d'évaluer nos travaux de recherche, nous avons construit un ensemble de logiciels prototypes. En particulier, nous avons développé un support d'exécution sous la forme d'un intergiciel intercalé entre un système d'exploitation temps-réel COTS et le code de l'application.

D'autres études que la notre visant à utiliser des composants COTS pour construire des systèmes temps-réel critiques ont été menées. Par exemple, le projet Armada [ADcF+99] vise à développer un support d'exécution pour applications temps-réel critiques, à base de composants COTS (réseau, processeurs, système d'exploitation). Il inclut des protocoles similaires à ceux fournis par HADES (communications, gestion de groupes), mais contrairement à HADES donne des garanties moins fortes sur le respect des échéances. Un autre système apparenté est Guards [Pow01], qui vise des applications temps-réel strict, et est construit principalement, mais pas exclusivement, à partir de composants COTS. Les processeurs et système d'exploitation sont des composants COTS, mais le réseau d'interconnexion est un composant spécifique transportant une horloge globale et fournissant un protocole de consensus. Contrairement à ces deux exemples de systèmes apparentés, dans le support d'exécution HADES, tous les mécanismes de tolérance aux fautes sont implantés exclusivement *par logiciel* et le support d'exécution est conçu pour des applications *temps-réel strict*.

Nous avons examiné les conséquences du choix d'une solution *intergiciel* (*middleware*), dans laquelle les mécanismes de tolérance aux fautes sont intégralement mis en œuvre par logiciel au dessus d'un noyau de système COTS non modifié. D'un point de vue qualitatif, ce choix s'est avéré satisfaisant, car nous avons été capables de vérifier l'ordonnançabilité globale du système (support d'exécution compris).

D'un point de vue quantitatif, nous avons analysé le comportement de notre support d'exécution en présence de fautes en utilisant une méthode d'injection de fautes par logiciel. En particulier, nous avons évalué la couverture de l'hypothèse de silence sur défaillances et obtenu une couverture de 99.1%, ce qui ne convient pas pour des applications de haute criticité. Pour de telles applications, il nous semble indispensable de déporter la détection des erreurs à un niveau plus bas (au niveau du système d'exploitation ou au niveau du matériel). Une couverture de plus de 99% nous semble toutefois raisonnable pour des applications de criticité moindre, pour lesquelles les contraintes de coûts sont importantes et les conséquences de dysfonctionnements dus à une faute non catastrophiques. Étant donnée la grande diversité des applications temps-réel, il est délicat de donner de manière précise les différents domaines où une telle couverture est acceptable. Il nous semble toutefois que certaines applications

de commande de procédés industriels peuvent se satisfaire de cette couverture, si de par la nature du produit manufacturé les pertes financières occasionnées par un dysfonctionnement (rare) du système ne sont pas trop importantes.

Concernant les temps de réponse au pire cas des protocoles du support d'exécution, des performances de l'ordre de la milliseconde ont été obtenues, conditionnant ainsi la gamme d'échéances pouvant être supportées par notre support d'exécution ou des supports d'exécution similaires à base de matériel (processeur, réseau) et de logiciel (système d'exploitation) COTS : les échéances les plus courtes supportables pour les tâches distribuées doivent être supérieures à la milliseconde. Ici encore, le choix d'intégration de composants COTS (matériel et système d'exploitation), conditionne le type d'applications qui peut tirer profit d'un support d'exécution comme le nôtre, construit à base de matériel et système d'exploitation COTS. Soulignons toutefois que l'analyse de performances que nous avons menée a montré que les performances au pire cas des protocoles du support d'exécution sont d'autant meilleures que les tâches les implantant sont lancées souvent, et que la période minimale de lancement des tâches est influencée par la vitesse du processeur utilisé. Ainsi, l'évolution constante et soutenue des performances de processeurs rend de plus en plus courts les pire temps de réponse des protocoles tels que ceux intégrés dans HADES. De ce fait, des applications d'échéances de plus en plus courtes pourront être supportées avec l'augmentation de puissance des processeurs.

Chapitre 4

Bilan et perspectives

4.1 Bilan des travaux

Nous avons présenté dans ce document nos travaux de recherche concernant la conception de supports d'exécution pour applications ayant à la fois des contraintes de temps de réponse et de tolérance aux fautes. La conception de supports d'exécution a été abordée au travers de deux domaines d'applications ayant des besoins très différents envers le support d'exécution : des applications parallèles de longue durée s'exécutant sur réseaux de stations de travail hétérogènes, et des applications temps-réel critiques s'exécutant sur une architecture distribuée. Dans nos travaux, nous nous sommes attachés à prendre en compte deux types d'exigences : des exigences *opérationnelles* (cohabitation de mécanismes pour supporter les fautes et assurer des temps de réponse appropriés pour les applications), et des exigences *économiques* (diminution des coûts de construction et de maintenance du support d'exécution via l'intégration de composants existants).

4.1.1 Exigences opérationnelles : cohabitation entre temps de réponse et tolérance aux fautes

Afin de concilier temps de réponse et tolérance aux fautes, et ce quelle que soit la classe d'applications visée, notre démarche a été de concevoir des méthodes de tolérance aux fautes adaptées aux besoins du cadre applicatif.

Ainsi, concernant le recouvrement d'erreurs, nous avons sélectionné pour les applications parallèles une méthode à base de sauvegarde d'état/reprise, et avons éliminé d'emblée les méthodes à base de redondance des calculs, trop consommatrices en ressources de calcul. Il est en effet plus judicieux pour ces applications d'utiliser les ressources de calcul pour accélérer les applications que pour tolérer les fautes. En revanche, pour les applications temps-réel strict, le facteur qui influence la conception d'une méthode de recouvrement d'erreur est son influence sur le temps d'exécution des tâches et pas le volume de ressources consommées, qui doit être compatible avec

les échéances des tâches. Ainsi, nous nous sommes orientés pour ce type d'applications sur des méthodes à base de duplication des calculs, consommatrices en ressources de calcul, mais ayant moins d'influence sur les temps d'exécution des tâches que les méthodes de sauvegarde d'état en mémoire stable disque. Pour sélectionner le type de méthode de recouvrement d'erreurs le mieux adapté à un cadre applicatif donné, il faut, à notre avis, non seulement prendre en compte les contraintes venant du type d'application visé (par exemple privilégier les méthodes à faible consommation de ressources de calcul pour les applications parallèles, ou les méthodes de latence compatible avec les échéances pour les applications temps-réel) mais également les facteurs technologiques liés à l'architecture (par exemple, rapidité du réseau influençant les performances des algorithmes de consensus pour la duplication active de tâches, présence de support matériel pour la tolérance aux fautes).

Une fois le type de méthode de tolérance aux fautes sélectionné, se pose le problème de l'implanter et de juger de la pertinence d'optimisations, telles que celles que nous avons proposées pour diminuer les temps de sauvegarde d'état de processus (§ 2.2.3 - sauvegardes incrémentales et asynchrones de points de reprise en stockage stable). Si ce type d'optimisations est valide pour les applications parallèles, ne nécessitant pas de temps de réponse garanti pour les mécanismes de tolérance aux fautes, il n'en est pas de même pour les applications temps réel strict, pour lesquelles les pires temps d'exécution des logiciels doivent être connus. Par exemple, nous n'avons pas estimé pertinent d'intégrer un transfert incrémental des données modifiées dans le cadre de la redondance passive de processus (§ 3.2.2) car l'introduction de ce type d'optimisation, si l'on ne connaît pas statiquement les données modifiées par un calcul, n'améliore pas le pire temps de transfert d'état entre calculateurs, mais seulement son temps moyen.

Outre le choix d'un type de méthode de tolérance aux fautes, dans les systèmes temps-réel strict, se pose le problème de *garantir* que le système, dans sa globalité (mécanismes de tolérance aux fautes compris) respectera ses échéances. Pour faciliter la vérification des échéances, nous avons choisi de traiter de manière fortement couplée, introduction de mécanismes de recouvrement d'erreurs et vérification des échéances, en agissant par transformation de la structure des tâches. Ce type de méthode a pu être mis en place car dans les applications temps-réel strict, toutes les synchronisations et communications entre tâches doivent être connues statiquement pour vérifier le respect des échéances, et de ce fait la transformation de la structure des tâches peut être réalisée de manière *automatique*. Malgré l'attrait de l'automatisation, ce type de méthode serait non trivial à transposer à des systèmes non temps-réel strict, car le plus souvent toutes les informations sur les communications et synchronisations n'apparaissent pas explicitement et ne sont pas connues statiquement.

Enfin, que le support d'exécution doive ou non tolérer les fautes, la présence de contraintes temps-réel strict nécessite, ou tout du moins encourage fortement, la mise en place de méthodes d'analyse d'ordonnançabilité. Ce type de méthode requiert la connaissance, avant exécution, du pire temps d'exécution des tâches considérées de

manière isolée. Notre contribution dans ce domaine a porté principalement sur la mise en place de méthodes et outils d'obtention des pires temps d'exécution. Des expérimentations ont montré l'intérêt de ce type de méthode, tant d'un point de vue qualitatif que quantitatif, bien que certains travaux sont envisageables pour rendre les résultats d'analyse encore moins pessimistes (voir § 4.2.1.1).

4.1.2 Exigences économiques : intégration de matériels et logiciels existants

Quel que soit le domaine d'application visé, intégrer des composants existants dans un support d'exécution pose deux types de problèmes.

Le premier est de *caractériser* le comportement des composants qui sont intégrés. D'une part, nous nous sommes intéressés au comportement de systèmes d'exploitation COTS en présence de fautes, par une méthode d'injection de fautes logicielle. Nous avons utilisé ce type de méthode pour caractériser un support d'exécution pour applications temps-réel strict, et de ce fait avons privilégié l'aspect non intrusif du processus d'injection de fautes, pour être compatible autant que faire se peut avec des échéances temps-réel. Ce type de méthode peut être utilisé pour caractériser d'autres types de composants, avec toutefois des résultats différents selon leur capacité à détecter et/ou masquer les fautes.

D'autre part, nous nous sommes intéressés à déterminer le comportement temporel au pire cas de logiciels (en particulier de systèmes d'exploitation), par analyse statique de programme. La méthode proposée peut être appliquée à d'autres logiciels que ceux que nous avons analysés dans la mesure où ils sont utilisés dans le cadre de systèmes temps-réel *strict*, pour lesquels c'est le pire temps d'exécution du logiciel qui prime.

Un deuxième problème soulevé par l'intégration de composants existants est de *complémenter* leur fonctionnalités pour qu'ils répondent aux contraintes du domaine. Par exemple, pour supporter l'exécution d'applications parallèles en utilisant des composants (système d'exploitation, matériel) hétérogènes, nous avons développé un intergiciel complétant les fonctions du système d'exploitation pour transformer les formats de données entre architectures de type différent. Nous avons également développé sous la forme d'un intergiciel des mécanismes de détection d'erreurs (détection de données et comportements incorrects) pour augmenter les capacités de confinement des erreurs d'un système d'exploitation. Bien que cette deuxième étude ait été menée dans le cadre des systèmes temps-réel strict, les mêmes mécanismes peuvent être réutilisés pour rendre plus robustes (assurer un meilleur confinement des erreurs) des systèmes d'exploitation non temps-réel, voire d'autres types de logiciels.

4.1.3 Expérimentations

Tous nos travaux de recherche ont été validés via la construction de logiciels prototypes. Nous pensons que cette étape d'expérimentation est incontournable lorsque

l'on construit des supports d'exécution. Nous tirons plusieurs leçons des expérimentations que nous avons menées en construisant des supports d'exécution prototypes. Une première constatation est la faible capacité des composants COTS (en particulier de l'ensemble formé par un processeur généraliste comme le Pentium et un système d'exploitation temps-réel) à confiner les erreurs. Bien que la couverture de l'hypothèse de silence sur défaillance puisse être augmentée de manière importante par ajout de mécanismes de détection d'erreurs implantés par logiciel, la couverture résultante reste trop faible pour supporter des applications de haute criticité. Une deuxième leçon concerne les temps de réponse de protocoles pouvant être obtenus en les construisant au-dessus de systèmes d'exploitation COTS. Nous pensons que les systèmes d'exploitation actuels ne comportent pas assez de capacités d'extension ou de personnalisation. Certaines possibilités d'extension/personnalisation, comme par exemple la possibilité d'exécuter des processus en mode superviseur pour les applications parallèles, nous auraient permis de diminuer les temps de réponse de manière significative.

4.2 Perspectives

Nous dressons ci-dessous quelques perspectives ouvertes par notre travail et que nous comptons explorer.

4.2.1 Systèmes temps-réel

4.2.1.1 Maîtrise du comportement temporel de logiciels temps-réel

Maîtriser les temps d'exécution au pire cas des logiciels est de première importance dans le cadre des systèmes temps-réel strict. Nous envisageons de poursuivre nos travaux concernant la détermination des temps d'exécution au pire cas de logiciels dans deux directions complémentaires.

Une première direction est d'obtenir les temps d'exécution au pire cas par analyse statique de programmes dans la continuité des premiers travaux que nous avons déjà effectués (voir section 3.3). Plusieurs voies de travail peuvent être identifiées :

- *Analyse de bas niveau* (architecture) : réduction du pessimisme de l'analyse par la prise en compte de l'architecture interne du processeur. On pourra par exemple prendre en compte des caractéristiques architecturales supplémentaires telles que les caches de données ou l'exécution dans le désordre. Il faut toutefois être attentifs à la complexité grandissante de l'architecture interne des processeurs. Nous nous orienterons de préférence vers des processeurs utilisés dans le monde de l'embarqué, dont la structure interne semble plus maîtrisable que celle des calculateurs haute-performance.
- *Analyse de haut niveau* : réduction du pessimisme de l'analyse via une identification précise des pires chemins d'exécution dans les programmes sources. Des pistes, à explorer en collaboration avec des chercheurs en analyse statique de programmes, sont par exemple la détermination automatique des pires nombres

d'itérations possibles de boucles, ou l'identification des chemins infaisables. Par ailleurs, nous avons d'ores et déjà débuté une étude visant à appliquer une méthode d'évaluation partielle avant l'analyse de temps pire cas pour en réduire le pessimisme.

- *Modularité et reciblage* : l'analyse statique de WCET met en jeu un ensemble de techniques, potentiellement interchangeables, que ce soit pour l'analyse de haut niveau ou de bas niveau. Il est nécessaire que l'analyseur qui intègre ces techniques soit *modulaire*, de manière à pouvoir par exemple intégrer la modélisation de nouveaux éléments architecturaux, recibler l'analyseur vers un autre processeur, ou encore intégrer des modules d'analyse de haut niveau comme par exemple la détermination des chemins infaisables. Nous travaillons actuellement dans cette direction, et envisageons la diffusion de l'analyseur de WCET HEPTANE sous la forme de logiciel libre à destination de la communauté temps-réel.

Obtenir les temps d'exécution au pire cas par analyse statique de programme présente deux inconvénients qui peuvent être jugés inacceptables selon le logiciel à évaluer et l'architecture cible. D'une part, l'utilisation d'une telle méthode d'analyse nécessite d'avoir accès au code source du logiciel à analyser. D'autre part, il est indispensable d'avoir des informations précises sur la structure interne du processeur, ce qui est de plus en plus difficile (sans contact privilégié avec un constructeur) étant donnée la complexité grandissante des processeurs. Ces deux raisons nous amènent actuellement à étudier les outils de caractérisation de temps de réponse de type *boîte noire*, nécessitant moins d'informations sur le matériel et le logiciel, au détriment de la sûreté des résultats. L'idée est ici d'obtenir les temps d'exécution par exécution du logiciel, en orientant la génération des données d'entrée du logiciel de manière à se placer autant que faire se peut dans le scénario aboutissant au pire temps d'exécution du logiciel. La méthode triviale consistant à générer toutes les configurations possibles de paramètres d'entrée posant des problèmes d'explosion combinatoire, il est nécessaire de réduire la taille de l'espace de recherche. Il existe de nombreux points communs entre le domaine du test de logiciel et le type de méthode que nous cherchons à développer, en particulier le problème de réduction de la combinatoire de génération des tests. En revanche, ici l'objectif n'est pas de vérifier la conformité du logiciel par rapport à ses spécifications (dont nous de disposons pas) mais de déterminer son pire temps d'exécution. Soulignons aussi qu'un des problèmes est de générer des données d'entrée pour le logiciel qui soient correctes, afin d'évaluer le pire temps de réponse du logiciel et non pas son temps de réponse en présence de fautes d'origine logicielle.

Par ailleurs, il nous semble indispensable d'expérimenter les deux types de méthodes d'obtention des WCETs (analyse *statique* et *dynamique*) sur du code de taille conséquente. Notre cible privilégiée sera le code de systèmes d'exploitation temps-réel.

4.2.1.2 Systèmes d'exploitation temps-réel

On observe actuellement une complexification des logiciels temps-réel, notamment dans le monde de l'embarqué (automobile, téléphonie par exemple). En particulier, pour des raisons de coût, on trouve de plus en plus souvent plusieurs applications, potentiellement de criticité différente, embarquées sur le même calculateur et partageant les mêmes ressources. Par ailleurs, de par la complexité croissante des logiciels embarqués, ces derniers sont souvent développés et validés par des intervenants différents (par exemple les équipementiers dans le domaine de l'automobile). Il nous semble nécessaire de trouver des moyens pour faire *cohabiter* ces différents logiciels sur la même architecture. Une voie de recherche possible serait de travailler au niveau du système d'exploitation pour fournir des mécanismes d'isolation temporelle et/ou permettre la cohabitation de différentes politiques d'ordonnancement.

4.2.2 Tolérance aux fautes

Les systèmes informatiques sont de plus en plus souvent utilisés en remplacement de systèmes mécaniques/hydrauliques pour assurer des fonctions critiques. Dans ce cadre, la tolérance aux fautes reste un point clé dans la construction des systèmes d'exploitation temps-réel. Un point nous semblant évoluer dans le monde du temps-réel embarqué, notamment dans l'automobile, est la mise sur le marché (actuellement ou dans un futur proche) de matériels de communications assurant des propriétés fortes : synchronisation des horloges, contrôles temporels sur les accès au médium de communications, détection/masquage des fautes. De tels matériels sont par exemple TTP/C [Gmb99], TTcan [FMD+00], Byteflight [BPG00] ou encore Flexray [Bre01]. Par exemple, TTP fournit un accès au réseau selon une stratégie TDMA et fournit par matériel des horloges synchronisées, de la diffusion fiable et une gestion de groupes. L'existence de tels matériels, ne serait-ce que par la présence d'horloges fortement synchronisées entre les calculateurs, nécessite de revoir les fonctions à fournir au niveau du système d'exploitation, et ouvre de ce fait des perspectives de recherches.

Par ailleurs, un constat que nous avons effectué en évaluant le comportement du support d'exécution HADES en présence de fautes, est que l'utilisation de processus séparés, via l'utilisation d'un support matériel pour la traduction d'adresses (MMU) constitue un très bon mécanisme d'isolation des erreurs. Or, ce type de mécanisme, même s'il n'est pas assorti d'un va-et-vient des données entre mémoire et disque, est rarement fourni par les systèmes d'exploitation temps-réel. En effet, il introduit des délais pour l'accès aux données qui sont variables (selon la présence ou l'absence de l'adresse dans le cache de traduction d'adresses) et de temps au pire-cas importants (pire temps pour parcourir les tables de traduction d'adresses). Il nous semble intéressant d'étudier dans quelle mesure ce mécanisme peut être utilisé, même sous une forme dégradée, dans les systèmes d'exploitation temps-réel, de manière à concilier bonne isolation contre les fautes et temps-réel. Une idée serait par exemple d'offrir la notion d'espace d'adressage temps-réel, en trouvant un moyen d'assurer que tous les accès mémoire sont des succès dans le cache de traduction d'adresses pour que les

temps des accès mémoire soient à la fois bornés et de borne raisonnable.

Un autre constat que nous avons effectué lors de l'étude du comportement du support d'exécution HADES en présence de fautes est la *redondance* entre certains des mécanismes qu'il inclut pour s'auto-tester et ainsi être silencieux sur défaillance. Il nous semble nécessaire de proposer des méthodes (moins consommatrices en temps que l'exploration complète des différentes combinaisons possibles de mécanismes de détection d'erreurs) pour identifier les mécanismes redondants et ainsi trouver le meilleur compromis entre couverture du silence sur défaillance et coût associé.

Enfin, il nous semble utile de poursuivre des travaux dans l'analyse d'ordonnançabilité de systèmes tolérants aux fautes. Pour l'instant, nous nous sommes concentrés sur un modèle de fautes particulier (calculateurs à silence sur défaillances, réseau fiable et de latence de transmission bornée) et sur des méthodes de recouvrement d'erreurs particulières (duplication active, passive, et semi-active des tâches). Il nous semble intéressant d'étudier les nouveaux problèmes issus d'autres modèles de fautes et/ou méthodes de recouvrement d'erreurs.

4.2.3 Du temps-réel vers l'embarqué

De nombreux systèmes exhibent non seulement des contraintes de temps-réel, qu'elles soient strictes ou souples, mais également des ressources limitées, que ce soit du point de vue mémoire ou énergie. Jusqu'à présent, nous nous sommes uniquement intéressés à l'aspect temps-réel et tolérance aux fautes présentes dans un sous-ensemble des applications embarquées, et comptons nous pencher sur les autres contraintes de l'embarqué. Un exemple d'étude possible est la prévision a priori de la consommation énergétique de logiciels dont le code source est accessible. L'outil d'analyse statique de WCET que nous avons conçu, s'il est augmenté d'informations sur la consommation énergétique des instructions pourrait être utilisé à cet effet, dans la mesure où de telles informations peuvent être obtenues a priori.

Bibliographie

[ABC+95] Arlat (J.), Blanquart (J. P.), Costes (A.), Crouzet (Y.), Deswarte (Y.), Fabre (J. C.), Guillermain (H.), Kaâniche (M.), Kanoun (K.), Laprie (J. C.), Mazet (C.), Powell (D.), Rabéjac (C.) et Thévenod (P.). – *Guide de la sûreté de fonctionnement*. – Cépaduès, 1995.

[ACCP99] Anceaume (E.), Cabillic (G.), Chevochot (P.) et Puaut (I.). – A flexible run-time support for distributed dependable hard real-time applications. *In : Proceedings of the 2nd International Symposium on Object-oriented Real-time distributed Computing*, pp. 310–319. – St Malo, France, mai 1999.

[ACD+96] Amza (M.), Cox (A.), Dwarkadas (S.), Cyams (C.), Li (Z.) et Zwaenepoel (W.). – Treadmarks: Shared memory computing with networks of workstations. *IEEE Computer*, vol. 29, n2, février 1996, pp. 18–28.

[ADcF+99] Abdelzaher (Tarek F.), Dawson (Scott), chang Feng (Wu), Jahanian (Farnam), Johnson (Scott), Mehra (Ashish), Mitton (Todd), Shaikh (Anees), Shin (Kang G.), Wang (Zhiqun), Zou (Hengming), Bjorkland (M.) et Marron (P.). – ARMADA middleware and communication services. *Real-Time Systems*, vol. 16, n2, 1999, pp. 127–153.

[AG96] Adve (Sarita V.) et Gharachorloo (Kourosh). – Shared memory consistency models: A tutorial. *IEEE Computer*, vol. 29, n12, décembre 1996, pp. 66–76.

[AP98] Anceaume (E.) et Puaut (I.). – *Performance Evaluation of Clock Synchronization Algorithms*. – RR n3526, INRIA, octobre 1998.

[BBI+94] Banâtre (M.), Belhamissi (Y.), Issarny (V.), Puaut (I.) et Routeau (J. P.). – Arche: A framework for parallel object-oriented programming above a distributed architecture. *In : Proc. of the 14th International Conference on Distributed Computing Systems*, pp. 510–517. – Poznan, Poland, juin 1994.

[BBI+95a] Banâtre (M.), Belhamissi (Y.), Issarny (V.), Puaut (I.) et Routeau (J.P.). – Adaptive placement of method executions within a customizable distributed object-based runtime system: Design, implementation and performance. *In : Proc. of the 15th International Conference on Distributed Computing Systems*, pp. 279–286. – Vancouver, Canada, juin 1995.

[BBI+95b] Banâtre (M.), Belhamissi (Y.), Issarny (V.), Puaut (I.) et Routeau (J.P.). – Isatis: A customizable distributed object-based runtime system. *In : Proc. of the 1995 French-Japanese workshop on object-based parallel and distributed computation (OBPDC'95)*, éd. par Springer-Verlag. – Tokyo, Japan, juin 1995.

[BBM88] Banâtre (J. P.), Banâtre (M.) et Muller (G.). – Ensuring data security and integrity with a fast stable storage. *In : Proc. of the 4th International Conference on Data Engineering*, pp. 285–293. – Los Angeles, USA, février 1988.

[BCSS90] Barton (J. H.), Czeck (E. W.), Segall (Z. Z.) et Siewiorek (D. P.). – Fault injection experiments using FIAT. *IEEE Transactions on Computers*, vol. 39, n 4, avril 1990, pp. 575–581.

[BGW93] Barak (A.), Guday (S.) et Wheeler (R.G.). – *The MOSIX Distributed Operating System, Load Balancing For Unix.* – Springer Verlag, 1993, *Lecture Notes in Computer Science.*

[BHV+90] Barrett (P.A.), Hilborne (A.M.), Veríssimo (P.), Rodrigues (L.), Bond (P.G.), Seaton (D.T.) et Speirs (N.A.). – The Delta-4 Extra Performance Architecture (XPA). *In : Proc. of the 20th International Symposium on Fault-Tolerant Computing Systems*, pp. 481–488. – Newcastle, U.K., juin 1990.

[BJ95] Bharrat (S.) et Jeffay (K.). – *Predicting Worst Case Execution Times on a Pipelined RISC Processor.* – Technical Report nTR94-072, University of North Carolina, Chapel Hill, avril 1995.

[BPG00] Berwanger (J.), Peller (M.) et Griessbach (R.). – A new high performance data bus system for safety-related applications. – BMW, *http : //www.byteflight.com/*, 2000.

[Bre01] Bretz (E. A.). – By-wire cars turn the corner. *IEEE Software*, avril 2001. – http://www.flexray-group.com/.

[BRS96] Bodin (F.), Rohou (E.) et Seznec (A.). – Salto: System for assembly-language transformation and optimization. *In : Proc. of the Sixth Workshop on Compilers for Parallel Computers.* – décembre 1996.

[BSP+95] Bershad (B. N), Savage (S.), Pardyak (P.), Sirer (E. G.), Fiuczynski (M. E.), Becker (D.) et C. Cha (bers). – Extensibility, safety and performance in the SPIN operating system. *In : Proc of the 15th Symposium on Operating Systems Principles.* – Copper mountain resort, Colorado, décembre 1995.

[BSS91] Bernard (G.), Steve (D.) et Simatic (M.). – Placement et migration de processus dans les systèmes répartis faiblement couplés. *Technique et Science Informatiques (TSI)*, vol. 10, n 5, mai 1991, pp. 375–392.

[BSS95] Bondavalli (A.), Stankovic (J.) et Strigini (L.). – *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, chap. Adaptable Fault Tolerance for Real-Time Systems, pp. 187–207. – Kluwer Academic Publishers, 1995.

[But97] Buttazzo (G. C.) (édité par). – *Hard real-time computing systems - predictable scheduling algorithms and applications.* – Kluwer Academic Publishers, 1997.

[Cab96] Cabillic (G.). – *Exécution d'applications parallèles sur architectures distribuées hétérogènes : propositions et mise en œuvre.* – Thèse de doctorat, Université de Rennes I, octobre 1996.

[CF78] Censier (L. M.) et Feautrier (P.). – A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, vol. 27, n 12, décembre 1978, pp. 1112–1118.

[Che99] Chevochot (Pascal). – *Conception de systèmes distribués temps-réel strict tolérants aux fautes.* – Thèse de doctorat, Université de Rennes I, décembre 1999.

[CL85] Chandy (K. M.) et Lamport (L.). – Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, vol. 3, n1, février 1985, pp. 63–75.

[CMP95] Cabillic (G.), Muller (G.) et Puaut (I.). – The performance of consistent checkpointing in distributed shared memory systems. *In : Proc. of the 14th Symposium on Reliable Distributed Systems*, pp. 96–105. – Bad Neuenahr, Germany, septembre 1995.

[CMS98] Carreira (João), Madeira (Henrique) et Silva (João Gabriel). – Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, vol. 24, n2, février 1998, pp. 125–136.

[Col01] Colin (Antoine). – *Estimation de temps d'exécution au pire cas par analyse statique et application aux systèmes d'exploitation temps-réel.* – Thèse de doctorat, Université de Rennes I, octobre 2001.

[CP91] Crétienne (P.) et Picouleau (P.). – *The basic scheduling problem with interprocess communication delays.* – Rapport technique n91.6, MASI, janvier 1991.

[CP96a] Cabillic (G.) et Puaut (I.). – Dealing with heterogeneity in stardust: an environment for parallel programming on networks of heterogeneous workstations. *In : Proc. of the Europar'96 conference*, éd. par in Computer Science (Lecture Notes). pp. 114–119. – Lyon, France, août 1996.

[CP96b] Cabillic (G.) et Puaut (I.). – Répartition de charge dans Stardust : un environnement pour l'exécution d'applications parallèles en milieu hétérogène. *In : Actes de l'école sur le placement dynamique et la répartition de charge.* – juillet 1996.

[CP97] Cabillic (G.) et Puaut (I.). – Stardust: an environment for parallel programming on networks of heterogeneous workstations. *Journal of Parallel and Distributed Computing*, vol. 40, n1, janvier 1997, pp. 65–80.

[CP99] Chevochot (P.) et Puaut (I.). – Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies. *In : Proc. of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 356–363. – Hong-Kong, China, décembre 1999.

[CP00a] Chevochot (P.) et Puaut (I.). – Holistic schedulability analysis of a fault-tolerant real-time distributed run-time support. *In : Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pp. 355–362. – Cheju Island, South Korea, décembre 2000.

[CP00b] Colin (A.) et Puaut (I.). – Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, vol. 18, n2-3, mai 2000, pp. 249–274.

[CP01a] Chevochot (P.) et Puaut (I.). – Experimental evaluation of the fail-silent behavior of a distributed real-time run-time support built from COTS components. *In : Proc. of the 2001 International Conference on Dependable Systems and Networks*, pp. 304–313. – Göteborg, Sweden, juillet 2001.

[CP01b] Colin (A.) et Puaut (I.). – Analyse de temps d'exécution au pire cas du système d'exploitation temps-réel RTEMS. *In : Seconde Conférence française sur les systèmes d'exploitation*, pp. 73–84. – Paris, France, avril 2001.

[CP01c] Colin (A.) et Puaut (I.). – A modular and retargetable framework for tree-based wcet analysis. *In : Proc. of the 13th Euromicro Conference on Real-Time Systems*, pp. 37–44. – Delft, The Netherlands, juin 2001.

[CP01d] Colin (A.) et Puaut (I.). – Worst-case execution time analysis of the RTEMS real-time operating system. *In : Proc. of the 13th Euromicro Conference on Real-Time Systems*, pp. 191–198. – Delft, The Netherlands, juin 2001.

[CPC+01] Chevochot (P.), Puaut (I.), Cabillic (G.), Colin (A.), Decotigny (D.) et Banâtre (M.). – Hades: a distributed system for dependable hard real-time applications built from COTS components. *IEEE Transactions on Computers*, 2001. – À paraître.

[CPR+92] Chérèque (M.), Powell (D.), Reynier (P.), Richier (J.-L.) et Voiron (J.). – Active replication in Delta-4. *In : Proceedings of the 22th International Symposium on Fault-Tolerant Computing*, pp. 28–37. – Boston, Massachusetts, USA, juin 1992.

[CS93] Cap (C. H.) et Strumpen (V.). – Efficient parallel computing in distributed workstation environments. *Parallel Computing*, vol. 19, 1993, pp. 1221–1234.

[EE99] Engblom (J.) et Ermedahl (A.). – Pipeline timing analysis using a trace-driven simulator. *In : Proc. of the 6th International Conference on Real-Time Computing Systems and Applications.* – IEEE Computer Society Press, décembre 1999.

[EES+99] Engblom (J.), Ermedahl (A.), Sjödin (M.), Gustafsson (J.) et Hansson (H.). – *Towards Industry Strength Worst-Case Exection Time Analysis.* – Rapport technique n 99/02, Uppsala, Sweden, Department of Computer Systems, Uppsala University, 1999.

[EG97] Ermedahl (A.) et Gustafsson (J.). – Deriving annotations for tight calculation of executing time. *In : Proc. Euro-Par'97 Parallel Processing.* pp. 1298–1307. – Springer-Verlag, août 1997.

[EJW96] Elnozahy (E.N.), Johnson (D.B.) et Wang (Y.M.). – *A Survey of Rollback-Recovery Protocols in Message-Passing Systems.* – Rapport technique n96-181, Carnegie Mellon University DCS, octobre 1996.

[EKO95] Engler (D. R.), Kaashoek (M. F.) et O'Toole (J.). – Exokernel: An operating system architecture for application-level resource management. *In : Proc of the 15th Symposium on Operating Systems Principles.* – Copper Mountain Resort, Colorado, décembre 1995.

[FMD+00] Furher (T.), Muller (B.), Dieterle (W.), Hartwich (F.), Hugel (R.) et Walther (M.). – Time triggered communication on CAN (Time Triggered CAN - TTCAN. *In : Proc. of the 7th International CAN Conference.* – 2000. http://www.can.bosch.com/.

[FMW97] Ferdinand (C.), Martin (F.) et Wilhelm (R.). – Applying compiler technique to cache behavior prediction. *In : ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pp. 37–46. – juin 1997.

[Fol93] Folliot (B.). – *Méthodes et outils de partage de charge pour la conception et la mise en œuvre d'applications dans les systèmes répartis hétérogènes.* – Thèse de PhD, Institut Blaise Pascal, avril 1993.

[Fuc96] Fuchs (E.). – An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. *In : Proceeding of the 2nd European Dependable Computing Conference*, éd. par Springer-Verlag, pp. 73–90. – Taormina, Italy, octobre 1996.

[GBD+94] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Manchek (R.) et Sunderam (V.). – *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* – MIT Press, 1994.

[Gha95] Gharachorloo (K.). – *Memory Consistency Models for Shared-Memory Multiprocessors.* – Thèse de PhD, Stanford University, 1995.

[GLL+90] Gharachorloo (K.), Lenoski (D.), Laudon (J.), Gibbons (P.), Gupta (A.) et Hennessy (J.). – Memory consistency and event ordering in scalable shared memory multiprocessors. *In : Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pp. 15–26. – Seattle, Washington, mai 1990.

[GLS94] Grop (W.), Lusk (E.) et Skjellum (A.). – *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* – MIT Press, 1994.

[GLSS01] Girault (A.), Lavarenne (C.), Sighireanu (M.) et Sorel (Y.). – Fault-tolerant static scheduling for real-time distributed embedded systems. *In : Proc. of the 21st International Conference on Distributed Computing Systems.* – Phoenix, USA, April 2001.

[Gmb99] GmbH (TTTech (Time Triggered Technology)). – Data sheet of TTP/C-C1 communication controller - as8201. – TTTech, *http : //http : //www.tttech.com/*, juillet 1999.

[GMM97] Ghosh (S.), Melhem (R.) et Mossé (D.). – Fault-tolerant scheduling on a hard real-time multiprocessor system. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, n3, mars 1997, pp. 272–284.

[HAM+99] Healy (C.), Arnold (R.), Mueller (F.), Whalley (D.) et Harmon (M.). – Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, vol. 48, n1, janvier 1999.

[HS94] Hou (C.-J.) et Shin (K.G.). – Replication and allocation of task modules in distributed real-time systems. *In : Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pp. 26–35. – Austin, Texas, USA, juin 1994.

[HSL78] Hopkins (A.L.), Smith (T.B.) et Lala (J.H.). – FTMP — a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, vol. 66, n10, octobre 1978, pp. 1221–1239.

[ISL98] Iftode (L.), Singh (J. P.) et Li (K.). – Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, vol. 31, n4, July/August 1998, pp. 451–473.

[KCG+95] Kermarrec (A.), Cabillic (G.), Gefflaut (A.), Morin (C.) et Puaut (I.). – A recoverable distributed shared memory integrating coherence and recoverability. *In : Proc. of the 25th International Symposium on Fault-Tolerant Computing Systems*, pp. 289–298. – juin 1995.

[KCZ92] Keleher (P.), Cox (A. L.) et Zwaenepoel (W.). – Lazy release consistency for software distributed shared memory. *In : Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pp. 13–21. – mai 1992.

[KKT95] Kanawati (G.A.), Kanawati (N.A.) et Tang (D.). – FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, vol. 44, n2, 1995, pp. 248–260.

[KMH96] Kim (S.-K.), Min (S. L.) et Ha (R.). – Efficient worst case timing analysis of data caching. *In : Proceedings of the 1996 Real-Time technology and Applications Symposium*. pp. 230–240. – IEEE Computer Society Press, juin 1996.

[Lal85] Lala (P.K.). – *Fault Tolerant and Fault Testable Hardware Design*. – Prentice Hall, 1985.

[Lam81] Lampson (B.). – Atomic transactions. *In : Distributed Systems and Architecture and Implementation : an Advanced Course*, pp. 246–265. – Springer Verlag, 1981.

[LBJ+94] Lim (S.-S), Bae (Y. H.), Jang (G. T.), Rhee (B.-D.), Min (S. L.), Park (C. Y.), Shin (H.), Park (K.), Moon (S.-M.) et Kim (C.-S.). – An accurate worst case timing analysis for RISC processors. *In : Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS94)*, pp. 97–108. – décembre 1994.

[LH89] Li (K.) et Hudak (P.). – Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, vol. 7, n4, novembre 1989, pp. 321–357.

[Li86] Li (K.). – *Shared Virtual Memory on Loosely Coupled Multiprocessors*. – Thèse de PhD, Yale University, 1986.

[LM95] Li (Y.-T. S.) et Malik (S.). – Performance analysis of embedded software using implicit path enumeration. *In : ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, éd. par Gerber (Richard) et Marlowe (Thomas). pp. 88–98. – New York, NY, USA, novembre 1995.

[LMW96] Li (Y.-T. S.), Malik (S.) et Wolfe (A.). – Cache modeling for real-time software: Beyond direct mapped instruction cache. *In : Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*. IEEE, pp. 254–263. – IEEE Computer Society Press, décembre 1996.

[mic90] microsystems (Sun). – Network programming guide - external data representation standard. – Protocol specification, 1990.

[Mok83] Mok (A. K.). – *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. – Thèse de PhD, Massachusetts Institute of Technology (MIT), mai 1983.

[Mos93] Mossé (D.). – *Design, Development, and Deployment of Fault-Tolerant Applications for Distributed Real-Time Systems*. – USA, Thèse de PhD, University of Maryland, 1993.

[MP97] Morin (C.) et Puaut (I.). – A Survey of Recoverable Distributed Shared Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, n9, septembre 1997.

[Mue94] Mueller (F.). – *Static Cache Simulation and its Application*. – Thèse de PhD, Departement of Computer Sciences, Florida State University, juillet 1994.

[Mue00] Mueller (F.). – Timing analysis for instruction caches. *Real-Time Systems*, vol. 18, n2, mai 2000, pp. 217–247.

[OAR98] On-Line Applications Research Corporation, Huntsville, AL, USA. – *RTEMS Applications C User's Guide*, octobre 1998, 4.0 édition. (http://www.oarcorp.com/RTEMS/rtems.html).

[OS93] Oh (Y.) et Son (S.H.). – *Scheduling Hard Real-Time Tasks with Tolerance of Multiple Processor Failures.* – Rapport technique n CS-93-28, University of Virginia, mai 1993.

[OS97] Ottosson (G.) et Sjödin (M.). – Worst-case execution time analysis for modern hardware architectures. *In : ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems (LCTRTS'97).* – juin 1997.

[Par93] Park (C. Y.). – Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, vol. 5, n1, 1993, pp. 31–62.

[PB00] Puschner (P.) et Burns (A.). – A review of worst-case execution-time analysis. *Real-Time Systems*, vol. 18, n2-3, mai 2000, pp. 115–128. – Guest Editorial.

[PBS+88] Powell (D.), Bonn (G.), Seaton (D.), Veríssimo (P.) et Waeselynck (F.). – The Delta-4 approach to dependability in open distributed computing systems. *In : Proc. of the 18th International Symposium on Fault-Tolerant Computing Systems*, pp. 246–251. – Tokyo, Japan, juin 1988.

[PK89] Puschner (P.) et Koza (C.). – Calculating the maximum execution time of real-time programs. *Real-Time Systems*, vol. 1, n2, septembre 1989, pp. 159–176.

[Pow91] Powell (D.) (édité par). – *Delta-4 : A Generic Architecture for Dependable Distributed Computing.* – Springer-Verlag, 1991.

[Pow01] Powell (D.) (édité par). – *A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems.* – Kluwer Academic Publishers, 2001.

[PS91] Park (C. Y.) et Shaw (A. C.). – Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, vol. 24, n5, mai 1991, pp. 48–57.

[PS97] Puschner (P.) et Schedl (A. V.). – Computing maximum task execution times – a graph based approach. *In : Proc. of IEEE Real-Time Systems Symposium.* pp. 67–91. – Kluwer Academic Publishers, 1997.

[PY90] Papadimitriou (C. H.) et Yannakakis (M.). – Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, vol. 19, n2, avril 1990, pp. 322–328.

[Ran75] Randell (B.). – System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, vol. 1, n2, juin 1975, pp. 220–232.

[RSL95] Reisinger (J.), Steininger (A.) et Leber (G.). – *Predictably Dependable Computing Systems*, chap. The PDCS Implementation of MARS Hardware and Software, pp. 209–224. – Springer-Verlag, 1995.

[SA00] Stappert (F.) et Altenbernd (P.). – Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, vol. 46, n4, 2000, pp. 339–355.

[Sch90] Schneider (F.B.). – Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, vol. 22, n4, décembre 1990, pp. 299–319.

[SMFA99] Salles (F.), Moreno (M. Rodriguez), Fabre (J.C.) et Arlat (J.). – Metakernels and fault containment wrappers. *In : Proc. of the 29th International Symposium on Fault-Tolerant Computing Systems*, pp. 22–29. – Madison (WI), juin 1999.

[SR88] Stankovic (J.A.) et Ramamritham (K.) (édité par). – *Tutorial on Hard Real-Time Systems.* – IEEE Computer Society Press, 1988.

[SRL90] Sha (L.), Rajkumar (R.) et Lehoczky (J.P.). – Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, vol. 39, n9, 1990, pp. 1175–1185.

[SSNB95] Stankovic (J.), Spuri (M.), Natale (M. Di) et Buttazzo (G.). – Implications of classical scheduling results for real-time systems. *IEEE Computer*, vol. 28, n6, juin 1995, pp. 16–25.

[Str95] Streich (H.). – Taskpair-scheduling: An approach for dynamic real-time systems. *International Journal of Mini and Microcomputers*, vol. 17, n2, 1995, pp. 77–83.

[SWG91] Singh (J.P.), Weber (W.D.) et Gupta (A.). – *SPLASH : Stanford Parallel Applications for Shared-Memory.* – Rapport technique nCSL-TR-91-469, Computer Systems Laboratory, Stanford University, avril 1991.

[TBW94] Tindell (K.), Burns (A.) et Wellings (A.). – An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems*, vol. 6, n1, mars 1994, pp. 133–151.

[WBDP98] Wellings (A.J.), Beus-Dukic (Lj.) et Powell (D.). – Real-time scheduling in a generic fault-tolerant architecture. *In : Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98)*, pp. 390–398. – Madrid, Spain, décembre 1998.

[WL88] Welch (J. Lundelius) et Lynch (N.). – A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, vol. 77, 1988, pp. 1–36.

[XP90] Xu (J.) et Parnas (D.L.). – Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, vol. 16, n3, mars 1990, pp. 360–369.

[YW94] Young (C.) et Wells (D.). – *Ray Tracing Creations, Second Edition.* – Waite Group Press, novembre 1994.

[ZBN93] Zhang (N.), Burns (A.) et Nicholson (M.). – Pipelined processors and worst case execution times. *Real-Time Systems*, vol. 5, n4, octobre 1993, pp. 319–343.

[Zho88] Zhou (S.). – A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, vol. 14, n8, septembre 1988, pp. 1327–1341.

[ZSLW92] Zhou (S.), Stumm (M.), Li (K.) et Wortman (D.). – Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, n5, septembre 1992, pp. 540–554.