DRAC: Un système de contrôle d'exécution pour multiprocesseur à mémoire partagée

Mauricio Pillon

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
/_/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique"

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Maurício Aronne PILLON

Le 30 novembre 2004

Titre :

DRAC : Un système de contrôle d'exécution pour multiprocesseur à mémoire partagée.

Directeur de thèse : Brigitte PLATEAU

JURY

M. GUY MAZARÉ	Président
M. DANIEL LITAIZE	Rapporteur
M. FRANK CAPPELLO	Rapporteur
M^me^ BRIGITTE PLATEAU	Directeur de thèse
M. OLIVIER RICHARD	Co-encadrant

A mes parents, Mara et João-Luis ...

Olha guri, repares o que estás fazendo
depois que fores é difícil de voltar
Passei-te um pito e continuas remoendo
Teu sonho moço neste rancho abandonar

Ecoute mon gars, réfléchis à ton avenir
Une fois parti, c'est difficile de revenir.
On fume ensemble et tu y repenses sans cesse
à partir de chez nous, à tes rêves qui naissent.

Olha guri, lá no povo é diferente
E certamente faltará o que tens aqui
Eu só te peço não esqueças de tua gente
De vez em quando manda uma carta, guri

Ecoute mon gars, les gens sont différents là-bas.
Ce qui est à toi ici, sans doute te manquera.
Une seule demande : n'oublie pas ta famille,
Pense à nous écrire de temps en temps, mon gars.

Se vais embora, por favor não te detenhas
Sigas em frente não olhes para trás
Que assim não vais ver a lágrima insistente
Que molha o rosto do teu velho, meu rapaz

Si tu t'en vas, n'hésite pas, je t'en prie ;
Va de l'avant, ne regarde pas en arrière
pour ne pas voir la larme têtue qui brille,
mon fils, sur le visage de ton vieux père.

Olha guri, pra tua mãe cabelos brancos
E pra este velho que te fala sem gritar
Pesa teus planos, eu quero que sejas brando
Se acaso fores pega o zaino para enfrenar

Ecoute mon gars, pour ta mère des cheveux blancs et
ce vieux qui te parle sans crier.
Mesure bien tes projets, tu dois être modéré ;
Mais si vraiment tu pars, monte le cheval bai.

Olha guri, leva uns cobres de reserva
Pega uma erva pra cevar teu chimarrão
E leva um charque que é pra ver se tu conservas
Uma pontinha de amor por este chão

Ecoute mon gars, prends quelques sous en réserve
pour ton chimarrao, emporte un peu d'herbe
et de la viande fumée,
en souvenir, pour que tu conserves
un petit brin d'amour pour notre terre.

Um Pito – Osvaldir e Carlos Magrão

Remerciements

Mes parents m'ont appris que la seule chose que l'on garde pour soit durant toute la vie est l'expérience et la connaissance. Tati, ma femme, m'a appris quant à elle la valeur de l'amour et de la compréhension dans les moments plus difficiles. Aujourd'hui, je vois que ma personnalité est le résultat d'un ensemble de petits et de grands événements. Ces quatre dernières années ont été pour moi l'occasion de nombreuses avancées, ainsi je souhaiterais apporter quelques remerciements.

Tout d'abord j'aimerais remercier le CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) pour m'avoir soutenu financièrement pendant le développement de ma thèse. Je remercie spécialement Mme. Eli Ribeiro et Mme. Elza Pires pour leurs compétences et leur disponibilité.

Je tiens à remercier tous les membres de mon jury : M. Guy Mazare, président du jury, M. Daniel Litaize et M. Franck Cappello, les rapporteurs, Mme. Brigitte Plateau, ma directrice, et M. Olivier Richard, mon co-encadrant. Je remercie particulièrement Brigitte et Olivier pour l'opportunité qu'ils m'ont offerte de travailler avec eux et pour leurs efforts permanents pour me comprendre tout au long de ce travail.

Je ne peux bien sûr pas oublier tous ceux qui m'ont motivé à venir en France, Celso Maciel da Costa, Paulo Fernandes, Fernando Dotti et tous mes collègues de master de la PUCRS, ainsi que Jacques Chassin et Jacques Briat enfin. "Muito obrigado !". Un grand merci "au comité d'accueil" : aux personnes qui m'ont dépanné plusieurs fois en me prêtant leurs appartements (João-Luis Tavares et Roberta Hessel), en allant me chercher au milieu de la nuit à Bourgoin Jailleu (Luiz-Gustavo et Kelly Fernandes) ou simplement en m'aidant à me familiariser avec la culture de la région (Andrea Charão et Nicolas Maillard).

De ce côté de l'océan, le chemin a été long... mais bien que loin de ma famille, je ne me suis jamais senti seul. Des heures au téléphone ou sur Internet par ICQ ou par visio-conférence, j'ai toujours eu le soutient des membres de ma belle famille, les Barbiero's, de la famille de ma mère, les Aronne's et enfin de la famille de mon père, les Pillon's. Je vous remercie, vous comptez énormément pour moi.

Au laboratoire ID, une nouvelle famille s'est fondée autour de moi. Je tiens particulièrement à remercier tous mes collègues du labo et plus spécialement Jesus Verduzco, Luiz-

Angelo Estefanel, Nicolas Capit et Pedro (Pierre) Neyron. Je remercie également tout les autres, mais la liste serait évidemment trop longue pour tous les citer... Enfin, pour les longues discussions de travail, je remercie en particulier Georges Da-Costa, Guillaume Huard, Jean-Marc Vincent, Jean-François Mehaut et Jacques Briat.

À la troupe brésilienne, je donne mes remerciements pour m'avoir apporté la "chaleur" du Brésil durant les longs hivers enneigés grenoblois. Je me souviendrais toujours des très bons moments passés avec vous tous. Merci à Ricardo Bispo, Rodrigo Neves et famille, Patricia Jacques, Rafael et Monica Avila, Jose-Celso Freire et famille, Flavio Wagner et famille, Leb et Marcia, Daniel Lima et famille ainsi qu'à Tetsu et Carla Koike.

Finalement, je tiens à remercier ma petit soeur, encore une petit fille quand je suis parti, aujourd'hui une femme, qui m'a beaucoup manquée toutes ces années. À ma femme, je dois la plus part de cette thèse. Lorsque j'étais découragé, elle me motivait ; lorsque je prenais un mauvais chemin, elle me guidait. Pendant ces années loin de notre pays et de notre famille, nous étions une seule personne ! Il n'y a plus rien à dire, tu a été simplement magnifique ! Merci.

Enfin, pour leur aide pour la rédaction (en français s'il vous plaît) et les longues relectures de ce manuscrit qui n'aurait certainement pas abouti sans eux, je remercie Olivier Richard et Pierre Neyron à nouveau.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Les besoins continus en puissance de calcul restent un moteur important dans l'évolution des technologies des ordinateurs. Dans le domaine scientifique, par exemple, on trouve facilement des applications capables d'épuiser la puissance de calcul même sur des machines parmi les plus récentes.

De nos jours, une grande partie de cette puissance est issue de l'utilisation des machines parallèles, tels que les multiprocesseurs ou les grappes. Néanmoins, obtenir les meilleures performances de ces machines reste difficile et dépend généralement de plusieurs facteurs.

Dans le cas spécifique qui nous intéresse ici et qui est celui des multiprocesseurs, le rapport entre les capacités de la hiérarchie mémoire et la vitesse des processeurs est à l'origine d'un des problèmes de performances les plus importants. On parle de contention ou de goulot d'étranglement mémoire afin de signifier que la saturation de l'accès à la partie haute de la hiérarchie mémoire est responsable d'une baisse de performances. Il est clair que l'augmentation du nombre de processeurs ne fait qu'accentuer le phénomène.

La technologie de la fabrication des processeurs évolue en effet généralement plus rapidement que celle de la mémoire centrale. L'interconnexion entre la mémoire centrale et les processeurs est un des points cruciaux dans l'architecture des multiprocesseurs, en effet ce point est fréquent.

Actuellement, les deux approches générales d'interconnexion sont par bus partagé ou par commutateur. La première suggère clairement un problème de contention sur le bus lorsque le nombre de processeurs augmente. La deuxième approche induit une complexité supérieure pour le commutateur et donc un coût et, en conséquence, un problème de réalisation. Bien entendu l'augmentation de la taille des caches permet de retarder ce

phénomène mais il ne l'élimine pas. De plus, le coût de fabrication des mémoires caches reste élevé et certaines applications possèdent de faibles localités spatiales et temporelles.

Que ce soit avec un bus partagé ou avec un commutateur, une des principales sources d'inefficacité sur les multiprocesseurs vient donc de la hiérarchie mémoire. Néanmoins, suivant la manière dont les machines multiprocesseurs sont exploitées, le phénomène peut être plus ou moins prononcé. Le type de modèle de programmation et l'ordonnancement des processus sont des facteurs importants, et dans l'idéal, ils doivent prendre en compte les limites de l'architecture. Cela étant, le développement des applications parallèles apporte aussi des contraintes comme la synchronisation. L'optimisation au niveau logiciel est donc aussi complexe qu'au niveau matériel.

L'obtention de performances optimales sur un multiprocesseur repose ainsi sur une utilisation équilibrée des ressources matérielles. Même si l'ordre d'exécution des instructions est décidé au niveau matériel, celui-ci ne contient pas assez d'informations pour empêcher la saturation d'une ou plusieurs ressources. Au niveau système et logiciel, l'ordonnanceur rencontre le problème inverse. En effet, il peut connaître les processus à exécuter, et changer l'ordre d'exécution des processus, mais cependant, il n'a pas d'informations précises du niveau matériel. Dans ce contexte, l'apparition des compteurs matériels de performances offre une nouvelle voie de recherche à ce problème. En effet, ils permettent l'observation précise des événements matériels par les applications et le système d'exploitation.

Une des premières catégories de logiciels ayant utilisée les compteurs matériels de performances a été celle des outils d'analyse de performances. L'observation des activités matérielles reliées aux informations du niveau applicatif permet à ces outils une analyse plus précise du comportement des applications. Par conséquent, il est possible de déterminer l'origine de certaines erreurs et d'identifier des problèmes d'inefficacité, comme le faux partage dans le cas des machines multiprocesseurs.

L'autre catégorie d'applications et de systèmes pouvant tirer parti des compteurs matériels est celle des mécanismes d'adaptations et des systèmes de contrôle. En effet, les compteurs remontent au niveau applicatif des événements qui renseignent sur le niveau d'utilisation des ressources matérielles. Ce type d'informations, sous certaines conditions, permet de rétroagir sur l'allocation de ces ressources aux différentes applications.

1.1 Objectifs de la thèse

Ce travail de thèse se place dans le contexte de la problématique des contentions mémoires sur les machines multiprocesseurs. Nous proposons l'utilisation des compteurs matériels en tant qu'élément d'un système de contrôle permettant de modifier l'ordonnancement de l'exécution des processus en présence d'une contention.

La politique de contrôle retenue consiste à maximiser le rendement de la machine ce qui correspond généralement au point de vue de l'administrateur. Le contrôle d'exécution des processus est basé sur l'estimation des performances via l'observation de l'utilisation mémoire. Ce mécanisme d'estimation est l'issue d'une étude sur l'impact des capacités des hiérarchies mémoires sur les performances des multiprocesseurs.

Parmi les architectures processeurs des machines étudiées, nous avons trouvé des événements matériels capables d'établir un lien entre le niveau d'utilisation mémoire et les performances des applications. Puisque l'observation de l'utilisation mémoire est possible via les compteurs matériels en cours d'exécution, l'estimation des performances l'est aussi.

1.2 Plan de la thèse

Ce mémoire de thèse est composé de 7 chapitres, dont l'introduction et la conclusion. Les paragraphes suivants résument l'ensemble des chapitres.

Chapitre 2 : Ce chapitre intitulé **Multiprocesseurs et compteurs matériels** est consacré à la présentation des éléments de base nécessaires à l'ensemble des études de cette thèse. Dans une première partie, nous présentons les principales évolutions des performances des processeurs et des mémoires. Nous en retirons un déséquilibre des capacités des débits mémoire en fonction de celles de calculs des processeurs. Par la suite, il est présentée l'étude des modèles de programmation parallèle et les ordonnanceurs disponibles pour les machines multiprocesseurs. Enfin, dans la dernière partie, nous décrivons en détail les compteurs matériels de performances des quelques architectures processeurs que nous avons étudiés.

Chapitre 3 : Ce chapitre intitulé **Outils et bibliothèques de contrôle du système**, présente un état de l'art des bibliothèques d'utilisation des compteurs matériels de performances, des outils d'analyse de performances et des systèmes de contrôle. Le nombre et le format des compteurs matériels varient d'une architecture à l'autre. C'est dans ce contexte que des bibliothèques ont été développées afin de simplifier l'utilisation des compteurs. En réduisant la complexité d'utilisation des compteurs matériels, ces bibliothèques ont permis une intégration simple des observations des activités matérielles dans les outils d'analyse de performances. Ainsi, ces outils disposent des informations en provenance du niveau matériel, du système et de l'applicatif ce qui permet une analyse plus fine de l'ensemble du système et de l'application. Finalement, les outils de contrôle en cours d'exécution peuvent prendre en compte les activités matérielles pour leurs prises de décisions.

Chapitre 4 : Dans ce chapitre intitulé **Observation de l'utilisation mémoire sur les machines multiprocesseurs** nous décrivons l'étude du comportement mémoire des applications observées via les compteurs matériels liés à la hiérarchie mémoire. Sur certaines architectures, il est possible d'établir un rapport entre le débit de l'utilisation du bus mémoire et l'accélération obtenue. Nous cherchons ici à obtenir les éléments nécessaires à l'élaboration d'un système de contrôle de processus. Dans une deuxième partie, une observation et une analyse des comportements des applications en fonctions du temps ont été proposées. Pour conclure, nous présentons une étude préliminaire sur l'influence provoquée par l'activité d'autres ressources telles que celles issues d'une carte réseau ou des activités des entrées et sorties de périphériques de stockage sur la hiérarchie mémoire.

Chapitre 5 : Dans ce chapitre intitulé **DRAC : Un système de contrôle d'exécution**, nous présentons l'architecture d'un système de contrôle de processus pour machine multiprocesseurs. La première partie est consacrée au principe du système avec ces objectifs ainsi que les problèmes provoqués par l'exécution de processus avec dépendance et synchronisation. Ensuite, nous décrivons les trois modules centraux de l'architecture, *le moniteur mémoire*, *l'ordonnanceur* et *le gestionnaire des jobs*. Enfin, dans la dernière partie, le prototype de ce système est détaillé.

Chapitre 6 : Dans ce chapitre intitulé **Modélisation et évaluation**, nous commençons par définir un modèle général et simplifié prenant en compte l'impact de la hiérarchie mémoire sur les performances d'un multiprocesseur. Puis, nous avons défini des modèles d'ordonnancements qui permettent d'évaluer les temps d'exécution selon trois scénarios d'ordonnancement différents (pire, moyen, meilleur). Finalement, la dernière partie contient les évaluations du prototype du système DRAC sur trois architectures processeurs différentes.

Pour conclure, un bilan général du travail ainsi que des pistes pour des travaux futurs seront présentés dans le dernier chapitre.

2

Multiprocesseurs à mémoire partagée et compteurs matériel

Dans la première partie de ce chapitre nous présentons une classification couramment employée pour introduire les architectures multiprocesseurs à mémoire partagé. La caractéristique importante de ces architectures est le médium d'accès à la mémoire centrale. Très souvent, sur les architectures avec un faible nombre de processeurs, il s'agit d'un seul bus mémoire partagé par l'ensemble des processeurs. L'avantage de cette approche est la simplicité de sa réalisation, cependant, celle-ci engendre généralement la formation de goulots d'étranglement mémoire qui perturbent l'exécution des applications et provoquent des chutes de performance. Nous présentons succinctement les modèles de programmation parallèle et les types d'ordonnancement utilisée sur ce type d'architecture. Dans le contexte des problèmes de performances, une aide précieuse est apportée par les compteurs matériels de performances, car ceux-ci permettent d'observer en détail l'utilisation des ressources et d'identifier les problèmes liés à la saturation de la hiérarchie mémoire. La dernière partie du chapitre est donc consacrée à la présentation de ces compteurs matériels de performances.

2.1 Architecture et exploitation des multiprocesseurs

Dans le monde de l'informatique la course à la puissance ne s'arrête jamais. Quelle que soit la puissance informatique disponible, elle n'est jamais suffisante. Au départ, cette augmentation de puissance était essentiellement obtenue en accélérant régulièrement la

fréquence d'horloge. Malheureusement, de nos jours nous commençons à atteindre les limites fondamentales de la matière [106].

La réduction des coûts de fabrication a permis une croissance phénoménale du marché des ordinateurs personnels, et même les machines avec plusieurs processeurs sont devenues courantes. Un nouveau problème apparaît donc, car l'obtention de performances optimales sur de telles machines ne dépend plus uniquement de la vitesse des processeurs, mais aussi de l'interconnexion de la mémoire et de la façon d'exploiter l'architecture de la machine.

2.1.1 L'architecture des multiprocesseurs à mémoire partagée

Dans le cadre de cette thèse, nous nous intéressons aux machines multiprocesseurs à mémoire partagée. La classification de l'ensemble des ordinateurs proposé par Flynn [47] reste l'une des plus utilisées dans la communauté. Selon le modèle de Flynn, les machines multiprocesseurs à mémoire partagée sont classées comme MIMD, soit une machine ayant plusieurs flots d'exécution simultanés sur plusieurs données.

Les machines multiprocesseurs à mémoire partagée, aussi appelés systèmes fortement couplés, ou simplement multiprocesseurs, sont caractérisées par plusieurs processeurs partageant un même espace d'adressage. Cependant, selon le type de lien entre la mémoire physique et les processeurs les temps, d'accès à un segment mémoire peuvent être différents d'un processeur à l'autre. Ces machines sont ainsi sous divisées en deux familles : les machines UMA (*Uniform Memory Access*), et les machines NUMA (*Non-Uniform Memory*) [105]. La figure 2.1 montre cette taxinomie, en haut, l'architecture UMA avec un seul bus de connexion à la mémoire partagée pour l'ensemble des processeurs, et en bas, l'architecture NUMA avec des mémoires locales à chaque processeur. Nous nous intéressons plus précisément à l'architecture UMA, elle a l'avantage d'être plus simple à réaliser et le désavantage d'être plus sensible au problème de «scalabilité».

Au cours de ce travail, nous utilisons le terme multiprocesseur pour désigner les multiprocesseurs à mémoire partagée de type UMA. Les problèmes posés par ce type d'architecture et les causes d'inefficacité sont déjà bien connus [65]. Ils sont plus ou moins prononcés selon le nombre de processeurs et la technologie utilisée pour l'implantation du réseau d'interconnexion mémoire.

La différence de fréquences entre le bus mémoire et les processeurs est la première source d'inefficacité. Prenons l'évolution des vitesses du processeur et du bus mémoire des machines Intel *Pentium* au cours de ces dix dernières années (voir la figure 2.2) : la première génération avait une fréquence de processeur de l'ordre de $100MHz$ avec le *Pentium Pro*, jusqu'à plus de 400Mhz pour le *Pentium II*, mais un bus mémoire à 60 ou $66MHz$. Les machines *Pentium III* ensuite ont atteint des fréquences de processeurs à $1100MHz$ et de bus à $133MHz$. Enfin, les processeurs *Pentium 4* sont actuellement aux

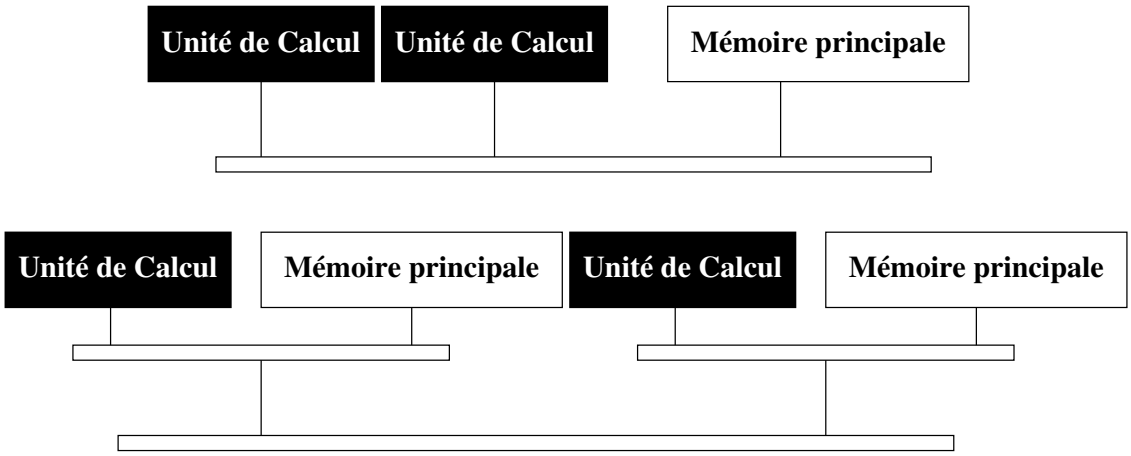

Figure 2.1 *En haut, l'architecture UMA, les machines ont un seul bus partagé de connexion à la mémoire. En bas, l'architecture NUMA, les machines ont une mémoire locale pour chaque processeur.*

alentours de $3000MHz$ et le bus arrive à $400MHz$, $533MHz$ ou $800MHz$.

Le rapport entre vitesse mémoire et processeur a donc commencé à 1.66 pour le *Pentium Pro*. Puis, il est monté jusqu'à 8.3 pour le *Pentium III*, atteignant enfin 4.25 pour les machines *Pentium 4* de nos jours. Cependant, il faut bien noter que les vitesses de bus annoncées pour les processeurs *Pentium 4* font référence à la technologie *quad pumped bus*, qui permet de transférer quatre signaux par cycle d'horloge dans la plupart des cas. Les vitesses de bus réelles sont donc de $100MHz$, $133MHz$ et $200MHz$, pour les bus annoncés à $400MHz$, $533MHz$ et $800MHz$, respectivement. En fait, les fréquences de bus réelles sont donc passées de $60MHz$ à $200MHz$. Finalement, en terme de débit nous sommes passé d'environ $500Mo/s$ pour le *Pentium Pro* à $1000Mo/s$ pour le *Pentium III*, puis à $3200Mo/s$ et $6400Mo/s$ pour le *Pentium 4*. La largeur de bus par contre est restée à $64bit$ (exception faite pour la toute dernière version de processeurs *Pentium 4* qui sont à $128bits$).

Les débits présentés ci-dessus sont issus du rapport entre la vitesse et la largeur des différents bus mémoire. Néanmoins, pour une analyse plus fine, ce ne sont pas les seuls paramètres à prendre en considération. En effet, le transit des données entre le processeur et la mémoire principale passe également par une puce de contrôle qui régit les accès à la mémoire et aux bus d'entrée et de sortie (*chipset*). L'entrée de cette puce de contrôle est donc astreinte au débit du bus mémoire et la sortie à celui de la mémoire principale. Dans certains ces connexions sont cadencés à des vitesses du même ordre, mais certaines architectures peuvent présenter un facteur 3 entre les deux.

Observons l'évolution les puces de contrôle des processeurs *Pentium 4* présentées dans le tableau 2.1 pour illustrer cela : la première génération de processeur *Pentium 4* compatible avec la puce de contrôle *Intel 845* était dotée d'un débit de bus mémoire de $3.2Go/s$,

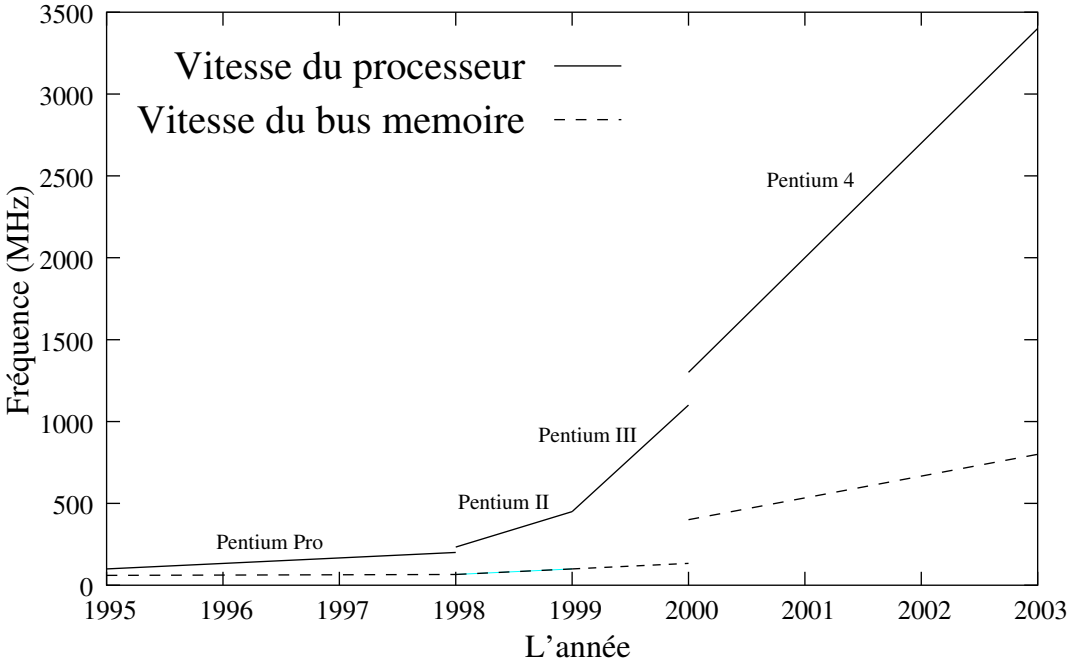

Figure 2.2 *Évolution des vitesses des processeurs et des bus mémoire pour les machines de la Famille Intel* Pentium. *On observe que le rapport entre la vitesse du processeur et du bus mémoire augmente de plus en plus avec les nouvelles architectures.*

or dans le meilleur des cas, le débit de la mémoire principale était de $2.1Go/s$. Par la suite, quand le débit de la mémoire a augmenté et atteint les $3.2Go/s$ avec la puce de contrôle *Intel 848P*, le débit du bus mémoire a augmenté également, et est passé à $4.2Go/s$, puis à $6.4Go/s$ avec ce même type de puce. Finalement, à l'arrivée de la technologie mémoire à doubles canaux (*Dual-channel*), le débit de la mémoire a rejoint celui du bus mémoire, à $6.4Go/s$. Cet équilibre est caractéristique sur les puces de contrôle *Intel 865* et *Intel 875*. Plus récemment, nous avons vu apparaître les puces de contrôle *Intel 9xx* qui supportent les mémoires DDR2 cadencée à $533MHz$ correspondant à un débit de $8.5Go/s$, avec des bus mémoire à $6.4Go/s$. Nous observons donc une inversion du rapport entre les débits au fur et à mesure de l'évolution des puces de contrôle. Dans un sens ou dans l'autre, le déséquilibre de toute façon est une seconde source d'inefficacité, si les besoins en débit des processeurs dépassent un des deux débits de la puce de contrôle.

Pour palier le déséquilibre entre les vitesses des différents éléments et augmenter l'efficacité de leurs processeurs, la réponse des fabricants a été d'augmenter la taille et le nombre de mémoires caches dans leurs processeurs. Les premières machines étaient généralement constituées de deux niveaux de caches (L1 et L2) qui étaient de toute petite taille. Actuellement, les tailles de ces caches sont plus grandes et il y a même des machines avec un troisième niveau de cache.

L'augmentation de la taille des caches a amélioré les performances, mais la conten-

Puce de contrôle	Fréquence du bus mémoire	Débit du bus mémoire	Débit maximal de la mémoire	Type de la mémoire
Intel 925X Express	$800MHz$	$6.4Go/s$	$8.5Go/s$	DDR2 ($533/400MHz$)
Intel 915G Express	$800/533MHz$	$6.4Go/s$	$8.5Go/s$	DDR2 ($533/400MHz$) ou DDR ($400/333Mhz$)
Intel 875P	$800/533MHz$	$6.4/4.2/3.2Go/s$	$6.4Go/s$	DDR ($400/333Mhz$)
Intel 865P	$800/533/400MHz$	$6.4/4.2/3.2Go/s$	$6.4Go/s$	DDR ($400/333/266Mhz$)
Intel 848P	$800/533/400MHz$	$6.4/4.2/3.2Go/s$	$3.2Go/s$	DDR ($400/333/266Mhz$)
Intel 845	$400MHz$	$3.2Go/s$	$1.06/1.6/2.1Go/s$	SDRAM ($133MHz$) ou DDR ($266/200Mhz$)

TAB. 2.1 *Dans ce tableau nous présentons les caractéristiques des bus mémoires et des mémoires principales, selon les puces de contrôle (*chipset*) compatible avec les processeurs* Pentium 4*. Nous observons, qu'au début, le débit de la mémoire principale était inférieur à celui du bus mémoire, et que plus récemment, ce rapport s'est inversé.*

tion sur le bus mémoire due à la différence entre sa vitesse et celle du processeur reste un problème. La solution matérielle la plus courante pour les multiprocesseurs est d'utiliser des commutateurs mémoires (avec un réseau d'interconnexion complexe) [106] [89]. Le commutateur permet d'augmenter le nombre de processeurs par machine, mais il ajoute un fort coût sur le prix. Les multiprocesseurs possédant des commutateurs mémoires à grande capacité sont des machines haut de gamme. Nous retrouvons, dans cette catégorie, des machines telles que SGI, CRAY, IBM SP entre autres. Les machines qui ont un seul bus mémoire, c'est à dire un réseau d'interconnexion simple, sont classées comme multiprocesseurs à coût faible.

Parmi les multiprocesseurs à coût faible, nous avons identifié deux types de liaison entre le processeur et la puce de contrôle. Le constructeur Intel [5] utilise l'approche autour d'un bus système partagé et le constructeur AMD [2] [1] l'approche point à point (commutateur non complexe). Le mécanisme de liaison point à point repose sur un lien entre la puce de contrôle et chaque processeur. La figure 2.3 présente ces architectures pour les machines biprocesseurs de manière simplifiée. Si la première approche suggère clairement une sensibilité possible au problème de contention sur le bus système partagé, il faut souligner que la deuxième approche impose une complexité supérieure pour la puce

de contrôle (*chipset*) et reporte le problème de la contention sur le lien entre la puce de contrôle et la mémoire principale.

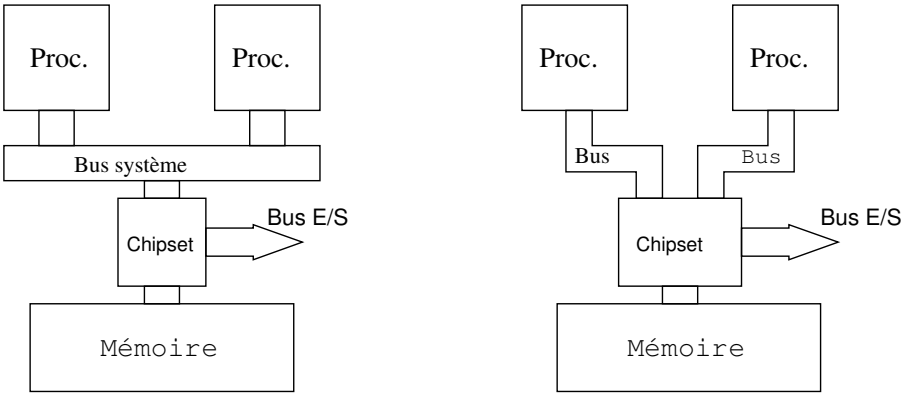

Figure 2.3 *À gauche, l'architecture du constructeur Intel basé sur l'approche bus système partagé, et à droite, l'architecture du constructeur AMD basé sur l'approche point à point.*

Depuis quelques années, nous avons vu l'apparition d'une nouvelle évolution : les processeurs capables d'exécuter des processus légers simultanément. La technologie SMT (*multithreading*) [109] [110] permet d'exécuter plusieurs processus indépendants sur l'ensemble des unités fonctionnelles à chaque cycle d'horloge. L'objectif de cette technologie est d'augmenter l'utilisation du processeur en exploitant le parallélisme de flot. Plus récemment, le fabricant Intel a lancé le premier processeur *Pentium 4* avec la technologie SMT. Selon les tests de performances présentés par Intel, ce processeur peut permettre un gain d'environ 30% [78]. Cela étant, les résultats de programmes de test qui font une utilisation intensive de la mémoire (*memory bound*) ont des performances bien inférieures. Cette chute de performances vient du problème de conflit de cache provoqué pour la technologie SMT [108] [28].

La technologie de fabrication des processeurs a évolué en réduisant la taille de gravure et en augmentant la puissance des processeurs. Cette évolution technologique a permis de rassembler plusieurs processeurs dans une seule puce. Dans la prochaine section, nous allons donc présenter ce type de machines multiprocesseurs : les machines à puce multiprocesseurs.

2.1.2 Puce multiprocesseur

De nos jours, les processeurs contiennent quelques dizaines de millions de transistors. Les prévisions montrent que d'ici moins de dix ans les processeurs devraient dépasser le milliard de transistors. Dans ce contexte, nous avons vu apparaître les architectures multiprocesseurs sur une même puce (*multiprocessor on chip*).

Une des principales difficultés sur ces architectures est la définition du réseau d'interconnexion mémoire entre les processeurs de la puce multiprocesseur. Dans le cas de multiprocesseur de puces multiprocesseurs, c'est l'interconnexion mémoire entre chaque puce multiprocesseur et la mémoire principale [111]. Les puces multiprocesseurs sont composées de plusieurs noyaux monoprocesseurs standards. La différence entre la hiérarchie mémoire d'un multiprocesseur et celle d'une puce multiprocesseurs est le niveau mémoire qui est partagé. Le multiprocesseur standard partage la mémoire principale alors que la puce multiprocesseur partage la mémoire cache L2 ou même la mémoire cache L1.

Nous présentons dans la figure 2.4 un exemple simplifié d'architecture de machine multiprocesseur constituée de deux puces multiprocesseurs. Chacune des puces accueille quatre processeurs qui partagent un seul bus système pour l'accès à la mémoire cache L2. Les deux puces, elles aussi, sont connectées sur un seul bus pour l'accès partagé à la mémoire L3 et à la mémoire principale. Coté résultats : ceux obtenus à travers des simulateurs ont montré que la puce multiprocesseur a plus de gain sur les applications sans dépendance de données [56] [87]. Par contre, l'exécution d'applications avec des données partagées provoque un grand nombre de conflits de cache [117] [83].

Le grain fin de parallélisme apparu avec la puce multiprocesseur a rendu plus complexes les mécanismes de contrôle matériel. Le partage de la mémoire cache L2 exige une amélioration des algorithmes de cohérence de cache. L'identification des violations de dépendances de données entre les mémoires caches peut être faite grâce à l'observation du bus système [113] ou par des tables de localisation et d'états des données [48] [70]. Le nombre de cycles pour l'accès à la mémoire cache a augmenté.

L'utilisation de ce type d'architecture exige des efforts supplémentaires de la part des fabricants, au niveau matériel, et des utilisateurs, pour le développement de logiciels parallèles. Nous pouvons citer deux exemples de multiprocesseurs avec des processeurs à puce multiprocesseurs : le processeur académique *Hydra Chip* de l'université de Stanfort [45] [57] et les processeurs commerciaux de la famille IBM Power4 [38] et Power5 [67].

Pour conclure, dans ces deux dernières sections, nous avons présenté les principales évolutions des technologies processeurs et mémoire. S'il est clair que l'exécution des applications sans dépendance de données a des performances meilleures à chaque nouvelle technologie, par contre, le partage de la hiérarchie mémoire reste un problème délicat pour les applications avec dépendances. Il est donc important de trouver des modèles de programmation adaptés à de telles architectures.

2.1.3 Modèles de programmation parallèle pour les multiprocesseurs

La performance d'une application dépend à la fois du niveau logiciel et du niveau matériel. Dans les sections précédentes, nous avons rappelé les principaux problèmes au

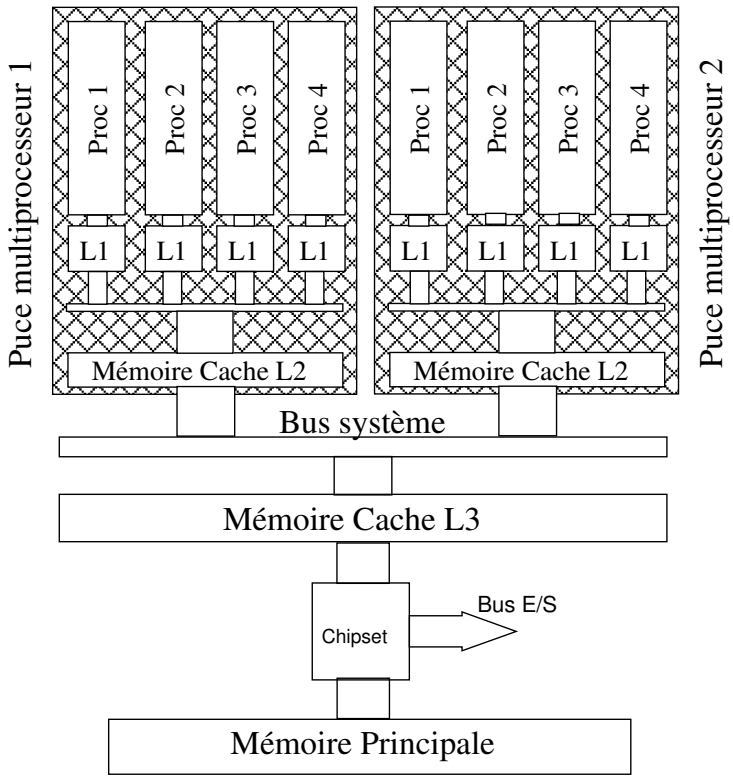

Figure 2.4 *Architecture simplifiée d'une machine multiprocesseur à mémoire partagée avec deux puces multiprocesseur. Le premier niveau mémoire partagé est la mémoire cache L2 pour l'ensemble des processeurs d'une même puce (dans ce cas, quatre). Le deuxième est la mémoire cache L3 partagée par la totalité des processeurs de la machine (dans cet exemple, 8 processeurs).*

niveau matériel du point de vue de la hiérarchie mémoire. Or l'ordonnancement des tâches à exécuter, ou la façon de paralléliser une application modifie l'ensemble des instructions traitées par les processeurs et, par conséquent, les performances de l'application. Cette section est consacrée à la présentation des modèles de programmation.

Le modèle de **multiprogrammation par processus** définit un processus comme une exécution séquentielle d'une suite d'instructions [26]. L'exécution concurrente de plusieurs processus qui coopèrent par le biais de la mémoire exige une synchronisation pour laquelle ils existants des mécanismes (sémaphores, verrous, conditions) et des ce approches (producteur/ consommateur, lecteur/ rédacteur) bien connus [106].

Implanter ce modèle nécessite de définir le grain de code correspondant à un processus, la stratégie d'allocation des processeurs aux processus et le grain d'entrelacement des processus entre eux. L'implantation de ce modèle repose sur le changement de contexte. Il s'agit d'être capable de sauvegarder le contexte d'exécution d'un processus puis ultérieurement de le restaurer de façon à permettre l'alternance de l'exécution des processus.

La **multiprogrammation par processus léger** est essentiellement une implantation à grain fin du concept de processus. Ce modèle contient une méthode de changement de contexte plus efficace que celle des processus. Les programmes avec processus légers peuvent avoir des données communes et des opérations de synchronisation classique.

Les processus légers ont été utilisés pour le développement de bibliothèques de programmation de plus haut niveau. Les bibliothèques de **multiprogrammation à mémoire partagée**, tel que OpenMP [27], rendent disponibles des interfaces de programmation parallèle explicites, simples et flexibles. Ce paradigme de programmation est basé sur l'exécution des processus parallèles, lesquels communiquent par une mémoire partagée.

Le parallélisme consiste à diviser un problème donné en plusieurs sous problèmes et à les résoudre simultanément. Jusqu'à présent, nous avons présenté les modèles de programmation basés sur le partage d'une mémoire commune. L'autre façon d'exécuter des programmes parallèles communicants est **l'échange de messages**. Le paradigme d'échange de messages réalise les transferts des données et la synchronisation entre les processus par l'envoi de messages. La bibliothèque la plus connue pour ce modèle de programmation est MPI (*Message Passing Interface*) [101]. Ce paradigme peut être utilisé tant sur des multiprocesseurs que sur des machines distribuées.

Enfin, avec l'apparition des grappes de multiprocesseurs a surgi le modèle de **multiprogrammation hybride**. Ce modèle consiste à paralléliser l'application à la fois sous le paradigme d'échange de messages et sous le paradigme à mémoire partagée. Les performances d'une application avec le modèle de programmation hybride dépendent du support de communication partagée (réseau et mémoire), du niveau de parallélisme employé et des accélérations des régions parallèles. Des travaux sur ce modèle de programmation ont montré que le grand nombre des paramètres rend le développement complexe et les gains ne sont pas évidents [69] [24] [25] [68].

Le choix du modèle le plus adapté est une tâche difficile. La complexité des architectures et le nombre de paramètres rendent le développement des programmes parallèles non trivial. Les caractéristiques de l'application, selon l'utilisation des ressources partagées, et de la machine sont un élément important pour le choix du modèle.

2.1.4 L'ordonnancement sur les multiprocesseurs

Dans les sections précédentes, nous avons montré quelques sources d'inefficacités au niveau de la hiérarchie mémoire sur les architectures des machines multiprocesseurs. Cependant, les choix faits pour l'ordonnancement de l'exécution des processus peuvent également provoquer la formation des goulots d'étranglements mémoire. En fait, la prise en compte des sources d'inefficacités de l'architecture par l'ordonnanceur est essentielle à l'optimisation des performances.

La politique d'ordonnancement sur les machines multiprocesseur est responsable du

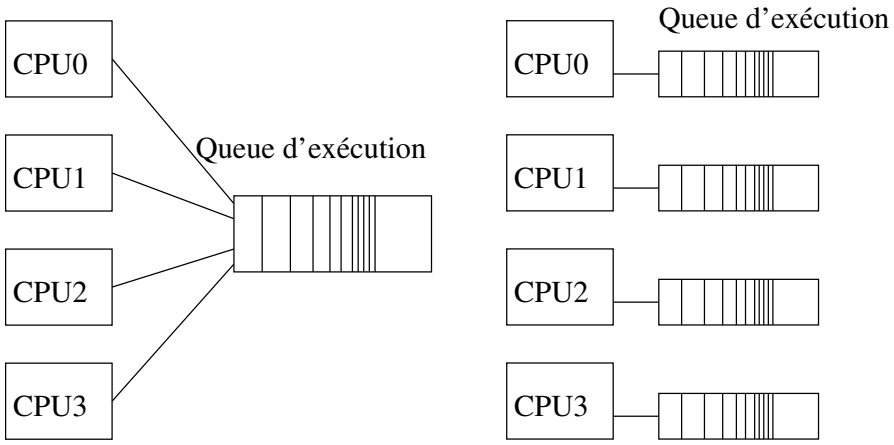

Figure 2.5 *Les ordonnanceurs sur des multiprocesseurs peuvent avoir deux types de queue d'exécution : ou bien une queue locale par processeurs ou bien une queue globale unique partagée par l'ensemble des processus. Si la première rend l'équilibrage de la charge plus simple, la deuxième permet la définition d'une politique d'ordonnancement par processeur.*

partage des ressources et du temps d'exécution entre les processus à exécuter. Cependant, avant même de définir une politique d'exécution, l'ordonnanceur de machines multiprocesseurs doit disposer d'une structure de queue d'exécution (*runqueue*), laquelle peut être globale ou locale à chaque processeur (figure 2.5).

Dans le cas où l'ordonnanceur dispose d'une queue d'exécution globale, l'équilibrage de charge entre les processeurs est certain, puisque chaque processus est ordonnancé sur un processeur à la fois. Néanmoins, le choix d'exécution d'un processus sur un processeur est aléatoire, et généralement alterne d'un processeur à l'autre à chaque nouvel ordonnancement. Ce changement de processeur empêche la réutilisation des données situées dans la mémoire cache, dans le cas d'une nouvelle exécution d'un même processus [71] [29].

L'utilisation d'une queue d'exécution locale à chaque processeur favorise la réutilisation des données présentes dans la mémoire cache, puisqu'elle facilite l'ordonnancement d'un processus toujours sur le même processeur. En effet, le fait d'avoir plusieurs queues peut provoquer une mise en attente de processus sur certains processeurs surchargés, pendant que d'autres processeurs, libres, n'ont pas de processus à exécuter.

Après le choix de la structure le l'ordonnanceur, il faut choisir les politiques d'ordonnancement. Elles peuvent être non-préemptive, par exemple avec un algorithme FIFO (*First In First Out*), ou préemptive, par exemple avec un algorithme de tourniquet [106] (*Round Robin*). Le premier algorithme lance les processus selon leurs ordres d'arrivées, alors que le deuxième alterne l'exécution des processus de la liste en respectant un temps maximum par cycle. On note qu'avoir des queues d'exécutions locales présente l'avantage d'offrir la possibilité d'utiliser une politique d'ordonnancement différente pour chaque

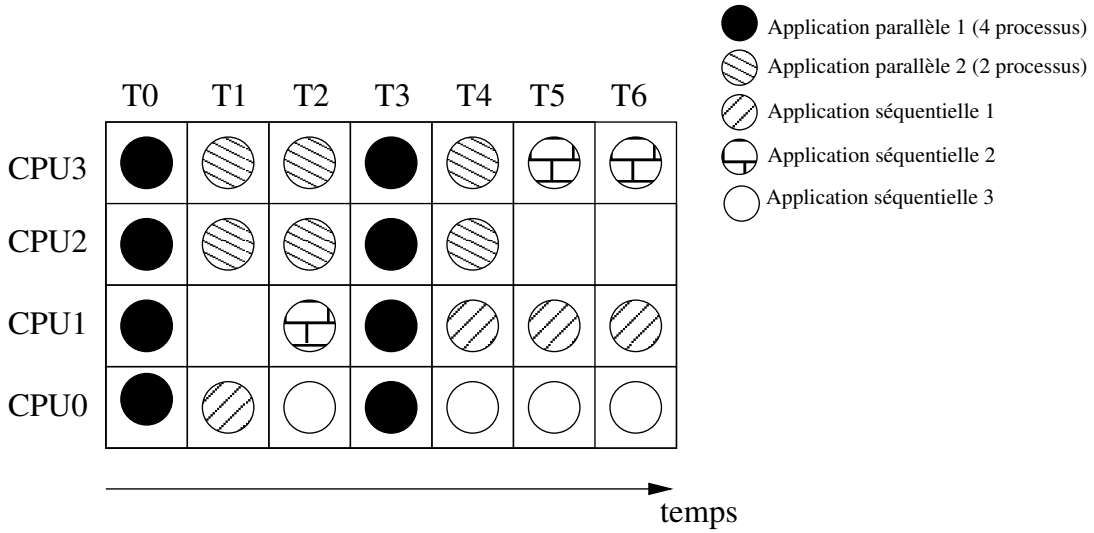

Figure 2.6 *L'algorithme d'ordonnancement* Gang Scheduling *a comme principe l'ordonnancement de tous les processus d'une application en même temps. Cet exemple montre qu'à chaque fois qu'un processus d'une application parallèle est ordonnancé, tous les autres sont aussi ordonnancés. De plus, il essaie d'ordonnancer toujours un processus sur le même processeur, sauf s'il reste des processeurs libres.*

processeur.

Dans le cas des applications parallèles, le problème de l'ordonnancement se complexifie encore [55]. En effet, les applications fortement synchronisées requièrent, évidemment, que la totalité de leurs processus s'exécute ensemble. Supposons qu'un des processus, pour une raison inconnue, soit retardé dans la queue d'exécution, les autres seront ordonnancés, cependant ils resterons bloqués puisqu'ils attendent l'arrivée du dernier processus du groupe. Ce scénario courant lors de l'exécution d'applications parallèles peut cependant être évité par l'utilisation de l'algorithme d'ordonnancement *gang scheduling* [61]. Son principe s'appuie sur une matrice, laquelle définit les processeurs sur la verticale et les temps d'exécution sur l'horizontale, comme nous le présentons dans l'exemple de la figure 2.6. Son mécanisme fait en sorte que les processus d'une même application parallèle soient toujours ordonnancés simultanément.

À titre d'exemple, dans le système d'exploitation Linux on dispose de deux ordonnanceurs différents, suivant la version du noyau utilisé 2.4 ou 2.6. Les principaux objectifs de ces deux ordonnanceurs sont : avoir un temps de réponse rapide et pas de situation de famine, respecter la localité des processus et obtenir un bon équilibrage de la charge [20] [98].

L'ordonnanceur du noyau Linux 2.4 a une seule queue d'exécution globale contenant l'ensemble des processus candidats à l'exécution. Quand le nombre de processus candi-

dats est important, le système peut mettre longtemps pour en choisir un. Afin d'éviter ce problème, un système de priorité d'exécution par processus est mis en place. À la fin d'un intervalle d'exécution, la priorité du processus sortant est décrémentée, ainsi le nombre de processus ayant la priorité la plus forte et qui sont donc candidats est réduit [12]. Même s'il utilise une queue globale, l'ordonnanceur Linux fait un lien entre le processus et le processeur pour favoriser la réutilisation des données de la mémoire cache en ordonnançant si possible chacun des processus sur un même processeur (principe d'affinité).

Récemment, nous avons vu l'apparition d'un nouvel ordonnanceur avec le noyau Linux 2.6 [74]. Contrairement au précédant ordonnanceur, celui-ci dispose d'une queue d'exécution locale par processeur. Chaque queue d'exécution peut accueillir deux types de processus : les processus normaux (*Normal tasks*) qui ont des priorités entre 0 et 99 ou les processus temps réel (*RT - Real time tasks*) qui ont des priorités de $100 - 139$. Les processus normaux sont ordonnancés selon leurs priorités, alors que les processus temps réel peuvent être ordonnancés par l'algorithme FIFO ou l'algorithme de type tourniquet.

Le choix de la politique d'ordonnancement dépend des objectifs et des caractéristiques des processus candidats. Les objectifs varient suivant les points de vue. D'un coté, les utilisateurs cherchent à augmenter la performance de leurs applications et donc à réduire le temps de calcul. De l'autre, l'administrateur veut maximiser l'utilisation de la machine. En règle générales, il est rare que les politiques d'ordonnancement satisfassent ces deux visions.

2.1.5 Synthèse

L'équilibrage de l'utilisation des ressources au niveau matériel, le modèle de programmation et les politiques d'ordonnancement contribuent ensemble au niveau de performance atteint par les machines. Les limites des technologies et la demande accrue de puissance de calcul ont favorisé l'évolution des techniques de parallélisme. Depuis l'apparition des machines parallèles, des multiprocesseurs, des grappes jusqu'aux puces multiprocesseurs, l'objectif du parallélisme a toujours été d'occuper au maximum toutes les ressources et d'augmenter la puissance de calcul.

Les limites physiques de la matière et la complexité du problème ne permettent pas d'obtenir des performances optimales pour toutes les applications. Des goulots d'étranglements sont présents sur la mémoire, sur le disque, sur le réseau, selon l'application et l'architecture. Puisque la résolution de tous ces goulots d'étranglement est un problème complexe, des travaux de recherches, dont ce travail de thèse, proposent des solutions par groupes d'applications.

L'intégration des informations de tous les niveaux de la machine permet l'étude et l'identification des goulots d'étranglement. Dans ce contexte, nous consacrons la prochaine section à la présentation des compteurs matériels de performances, qui nous permettront de les identifier.

2.2 Compteurs matériels

L'observation de l'exécution d'un ou plusieurs programmes permet de caractériser l'utilisation des ressources. Les techniques d'observation, comme la surveillance, le débogage interactif et l'étude des profils [54], aident à comprendre l'exécution des applications. Au début, les outils d'analyse et de traçage étaient basés sur l'activité des logiciels, et les informations prises au niveau du système d'exploitation ou de l'application elle-même. Or les informations disponibles au niveau logiciel ne sont pas précises, puisqu'il existe des événements matériels qui ne sont pas détectables par les couches supérieures.

L'apparition des compteurs matériels de performances a permis l'observation précise d'un grand nombre d'événements matériels. L'implantation des compteurs a été possible grâce à l'évolution des technologies de fabrication des processeurs. Le nombre des transistors a augmenté et le coût nécessaire pour ajouter des registres dédiés à l'observation est devenu très acceptable. Ainsi, grâce à ces compteurs nous sommes aujourd'hui capables de suivre l'activité d'un programme lors de son exécution, depuis le matériel jusqu'à l'application.

Malheureusement, le format et les types d'événements de ces compteurs varient selon l'architecture. La complexité d'utilisation dépend du format des compteurs, lesquels peuvent être décrits par un ou plusieurs registres. Les différences rendent peu portables les programmes qui utilisent les compteurs. De plus, comme leurs utilisation s'avère non triviale, les bibliothèques d'accès aux compteurs sont apparues (voir le chapitre 3 pour plus de détails). En outre il faut noter que, selon l'architecture, le nombre de compteurs peut varier de 2 à 18, tous ne permettent donc pas des analyses aussi poussées.

Dans le cadre de cette thèse, nous cherchons à observer l'utilisation des ressources sur les multiprocesseurs pour identifier des goulots d'étranglements. Sachant que la hiérarchie mémoire est un des goulots d'étranglement les plus connus et des plus importants, nous avons étudié les événements qui permettent d'observer l'activité de cette ressource. Les architectures choisies ont été celles les plus couramment utilisées dans le milieu académique, notamment pour les serveurs de calcul.

2.2.1 Famille Intel Pentium 6 (32 bits)

La famille de processeurs Intel *Pentium 6* regroupe les processeurs depuis le *Pentium Pro* jusqu'au *Pentium III*. Sur ces architectures il y a deux compteurs de performance disponibles : le registre **PerfEvtSel0** et le registre **PerfEvtSel1**. Chaque registre dispose de $40bits$ dont 32 sont dédiés à la configuration des événements matériels (voir figure 2.7). Les 8 premiers bits sont destinés à la spécification de l'événement, les 8 suivant définissent son masque. Les bits restant contiennent des options fines d'observation. Le masque sert à adapter le compteur d'événement, par exemple à l'observation des transactions envoyées par un seul processeur [7]. Chaque registre ne peut compter qu'un type d'événement à la

fois.

Nom du groupe d'événements	Nombre d'événements	Opérations observées
Data Cache Unit (DCU)	5	opérations de lecture et écriture sur l'ensemble de la mémoires
Instruction Fetch Unit (IFU)	5	instructions *fetch* de chargement
L2 Cache	11	opérations sur le cache L2
External Bus Logic (ELB)	19	opérations sur le bus mémoire
Floating Point Unit	6	opérations flotantes
Memory Ordering	5	l'ordre des opérations sur la mémoire
Instruction Decoding and Retirement	5	instructions de décodage et annulables
Interrupts	3	nombre d'interruptions
Branches	8	nombre des instructions de branchement
Stalls	2	nombre de stalls
Segment Register Loads	1	nombre de chargements du registre de segments
Clocks	1	nombre de cycles d'horloge
MMX Unit	7	nombre d'instructions MMX
Segment Register Renaming	3	nombre de renommage de registre

TAB. 2.2 *Nom du groupe des événements disponibles sur les architectures de la famille Intel* Pentium 6*.*

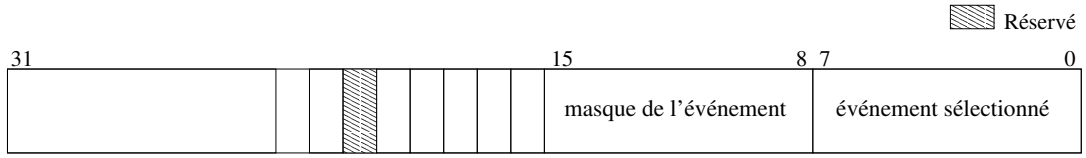

Figure 2.7 *Les registres* PerfEvtSel0 *et* PerfEvtSel1 *ont le même format. Les* 8 *bits de poids faible représentent l'événement choisi et les* 8 *suivants le masque de ce même événement.*

Les événements sont divisés par groupe selon les activités qu'ils observent. Le tableau 2.2 présente la liste des groupes d'événements de la famille *Pentium 6*. L'observation de l'activité mémoire est possible grâce aux événements appartenant à trois groupes, *Data Cache Unit (DCU)*, *External Bus Logic (ELB)* et *L2 Cache*. Le nombre d'événements de ces trois groupes est d'environ 30. La compréhension du fonctionnement et la

configuration de chaque événement exigent une bonne connaissance de l'architecture et ne sont pas triviales.

2.2.2 Famille Intel Pentium 4 et les processeurs Xeon (32 bits)

Les compteurs matériels de performances de tous les processeurs *Pentium 4* et de la dernière génération de Xeon ont les mêmes caractéristiques. Ces processeurs ont une architecture générale du type *Netburst* [4]. Ils disposent de 18 compteurs matériels de performance. Chaque compteur est un registre de $40bits$ qui est configuré grâce à deux autres registres : ESCR (*Event Selection Control Register*) et CCCR (*Counter Configuration Control Register*). Le registre de contrôle de sélection d'événement (figure 2.8), ESCR, contient, entre autres, $16bits$ pour le masque, 6 pour l'événement sélectionné et 35 réservés parmi ces $64bits$. Le registre de contrôle de configuration du compteur (figure 2.9), CCCR, a $64bits$, sur lesquels $49bits$ sont ignorés. L'identificateur de l'événement choisi (ESCR) est mis sur la position 13 du registre CCCR.

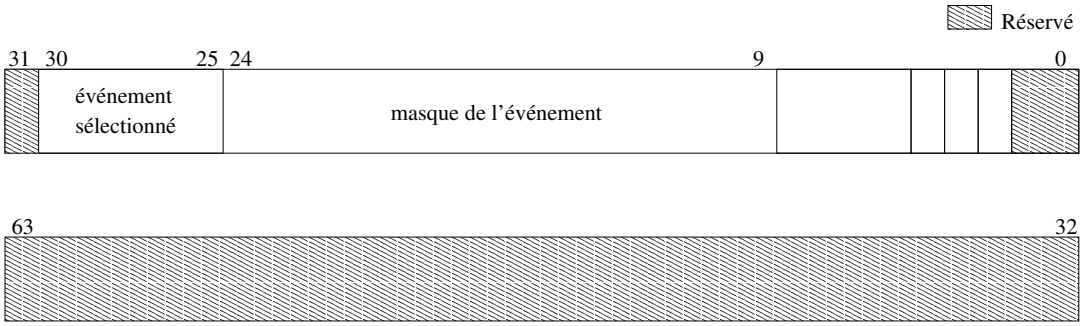

Figure 2.8 *ESCR : registre de contrôle de la sélection d'événement. Entre autres, nous mettons en évidence la configuration du masque de l'événement (du bit* 9 *au* 24*) et le champ de l'événement sélectionné (du bit* 25 *au* 30*).*

Les 18 compteurs matériels de performances sont organisés en 9 paires. Chaque paire de compteurs est liée à des sous groupes d'événements et à un ESCR. Ces mêmes 18 compteurs se divisent également en quatre groupes selon la fonction de l'unité du processeur à laquelle ils sont liés : *BPU*, *MS*, *FLAME* et *IQ*. Dans la figure 2.10, nous présentons un exemple avec l'unité fonctionnelle *IQ* (*Intruction Queen unit*) [7]. Les trois paires de compteurs, CCCR_Compteur0/CCCR_Compteur1, CCCR_Compteur2/CCCR_Compteur3 et CCCR_Compteur4/CCCR_Compteur5 permettent d'observer les activités des mêmes 4 unités [104]. Le choix de l'unité se fait par l'ESCR, situé en bas de chaque unité dans la figure 2.10. Les événements disponibles pour l'unité ALF, par exemple, peuvent être comptés par les compteurs CCCR_Compteur0, CCCR_Compteur1 ou CCCR_Compteur4. Selon cette logique, l'annexe B contient la liste des ESCR disponibles par compteur.

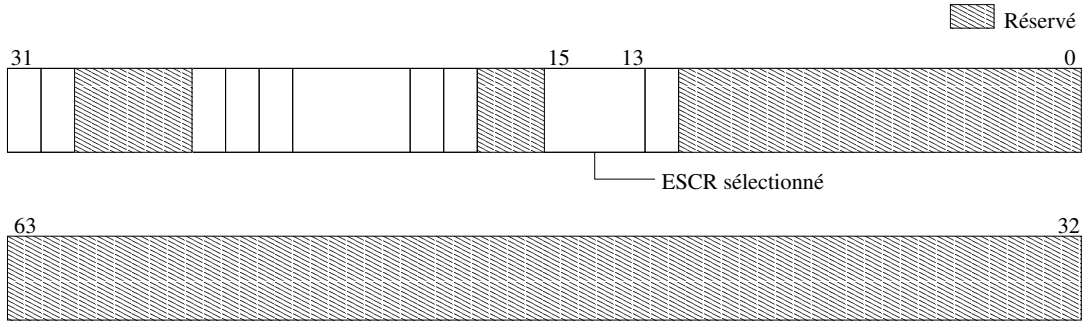

Figure 2.9 *CCCR : registre de contrôle de configuration du compteur. Le champ ESCR sélectionné fait le lien entre les configurations du registre et les paramètres de l'événement.*

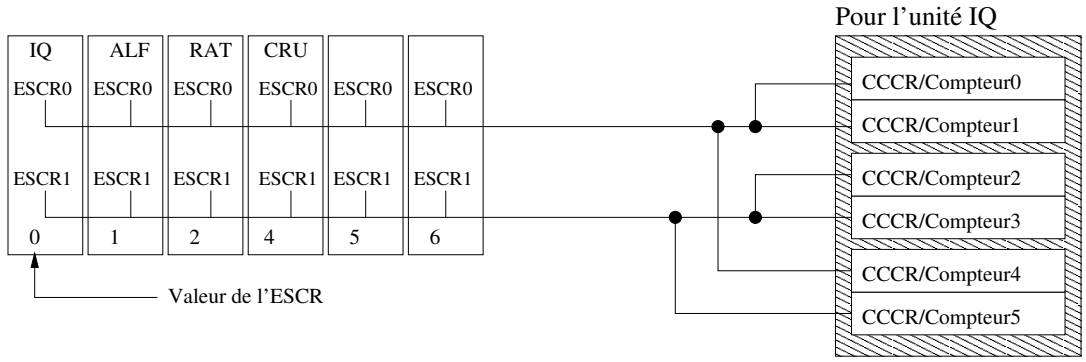

Figure 2.10 *Liens entre les compteurs matériels de performances et les unités fonctionnelles des processeurs* Pentium 4 *et Xeon. Les unités* IQ, ALF, RAT *et* CRU *sont liées aux compteurs* 0, 1 *et* 4 *par l'ESCR0, et aux compteurs* 2, 3 *et* 5 *par l'ESCR1.*

Maintenant, parmi les événements définis dans le guide Intel [7], prenons le *x87_assist* qui compte les instructions annulées. L'unité fonctionnelle capable de compter cet événement est la CRU. Les deux ESCR qui font le lien entre l'unité du processeur et les compteurs sont *MSR_CRU_ESCR2* et *MSR_CRU_ESCR3* (voir tableau de l'annexe B). Le numéro de l'événement est le $03H$, son masque peut être entre 0 et 4 et son CCCR est le $05H$.

La figure 2.11 montre la configuration des deux registres ESCR et CCCR à partir des informations de l'événement. Les compteurs liés à l'unité CRU sont ceux du groupe IQ (figure 2.10). Selon le tableau de l'annexe B, les 6 derniers compteurs ont les numéros 12 à 17. Sur le registre CCCR, nous changeons les bits 13 à 15 ($5 << 13$) selon les paramètres qui nous ont été donnés. Les deux autres paramètres sont sur le registre ESCR, le numéro d'événement ($03H$) sur les bits 25 à 30 ($3 << 25$) et le masque sur les bits 9 à 24. Dans notre exemple, nous avons décidé de compter 2 :POAO (*Handle x87 output overflow*), donc, le masque est $2 << 9$.

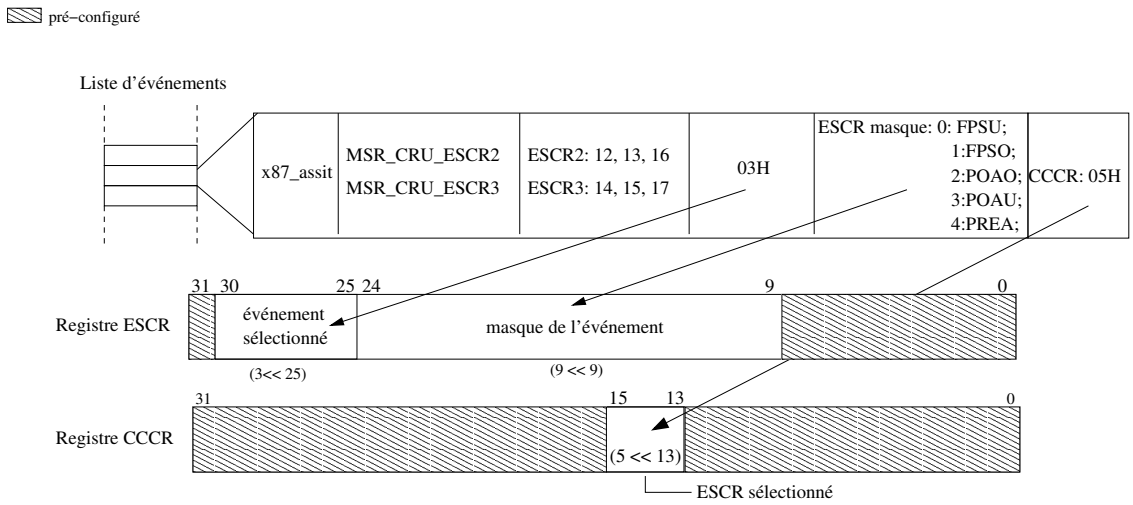

Figure 2.11 *La configuration des événements est faite à travers deux registres, l'ESCR et le CCCR. Les informations obtenues dans la liste d'événements, en haut, permettent la configuration des deux registres, en bas. Les flèches indiquent les valeurs à positionner dans les deux registres.*

La prochaine étape est la localisation de l'événement adapté à nos besoins. Si la tâche de configuration est difficile, la recherche et la compréhension de l'événement adapté à une étude ne le sont pas moins. La documentation Intel [7] sur les compteurs utilise un vocabulaire technique particulier et spécifique à l'architecture *Pentium 4* et Xeon. Le nombre total d'événements est de 48, cependant, la plupart des événements permettent d'utiliser un masque pour définir le comptage de sous événements. Chaque événement peut avoir de 0 à 14 sous événements, ce qui augmente le nombre de 48 à plus de 200.

Parmi les événements principaux, ceux qui semblent permettre l'observation des activités mémoire sont : *FSB_data_activity*, *IOQ_allocation*, *BSQ_cache_reference*, *memory_cancel*, *memory_complete* et *MOB_load_replay*. La configuration de l'événement et des sous événements et le choix du compteur exigent un haut niveau de connaissance du processeur.

2.2.3 Famille Itanium (64 bits)

Les architectures de la famille *Itanium* ont quatre compteurs matériels de performances, chacun de $32bits$. Ces compteurs sont organisés par paires, PMC / PMD(4,5) (*PMC : counter configuration register* et *PMD : performance counter data register*) et PMC / PMD(6,7) [9]. La seule différence entre ces deux paires est la taille du champs seuil, 2 bits sur PMC(4,5) et 3 bits sur PMC(6,7), qui contrôle le dépassement.

Dans la figure 2.12, nous présentons le format des compteurs matériels de perfor-

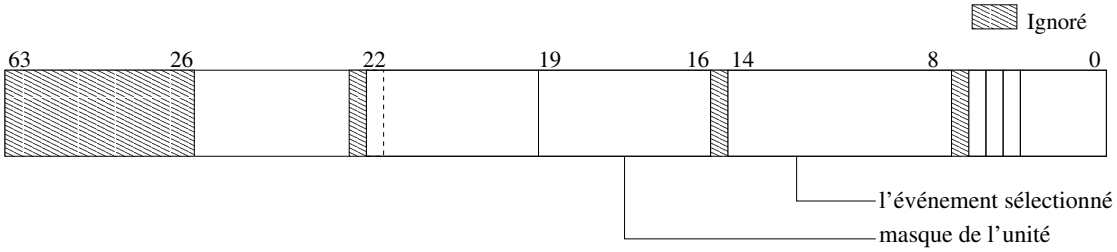

Figure 2.12 *Les registres, PMC(4,5,6,7), décrivent les configurations pour les compteurs de l'architecture* Itanium. *Ils sont organisés par paires, PMC(4,5) et puis PMC(6,7), qui a comme seule différence la taille du champ* threshold. *Une paire possède* 2 *bits (*20 *et* 21*) pour ce champ et l'autre* 3 *bits (bit* 22 *en supplément).*

mances. Le numéro de l'événement choisi se trouve dans les bits 8 à 14 et le masque dans les bits 16 à 19. La lecture d'un événement configuré par le registre PMC4 est disponible dans le registre PMD4.

Les événements sur l'architecture *Itanium* sont divisés en 6 catégories selon la ressource matérielle ou l'unité fonctionnelle qu'ils observent [9]. La catégorie **hiérarchie mémoire** regroupe les événements sur les trois niveaux de caches (*L1*, *L2* et *L3*) et sur le bus mémoire (*Front Side Bus*). L'ensemble des événements de cette catégorie compte un peu moins de 100 événements.

Ainsi, les deux compteurs du processeur *Itanium 2* mettent à disposition un grand nombre d'événements matériels. De plus, ils sont simples à configurer. Cela étant, ils présentent des limitations pour l'observation de plusieurs événements simultanément.

2.2.4 Famille AMD Athlon (32bits)

Les architectures de la famille AMD *Athlon* ont quatre compteurs matériels de performances, chacun de $48bits$. Les bits 33 à 48 sont ignorés. La configuration des événements sur ces architectures est faite à travers quatre registres, de **PerfEvtSel0** à **PerfEvtSel3**. La lecture du nombre d'événements sélectionnés par ces registres est disponible sur les registres **PerfCtr0** à **PerfCtr3** [8].

Les registres de configuration des événements sur les architectures *Athlon* ont le même format que ceux de la famille Intel *Pentium 6* représentée dans la figure 2.7. Cependant, le nombre d'événements est plus réduit et aucun numéro de masque ou d'identification (événement sélectionné) ne correspond.

Les événements sont divisés en quatre groupes selon l'unité de comptage du processeur. Le nombre total d'événement est 23, parmi eux 12 caractérisent la hiérarchie de la mémoire. La quantité réduite d'événements rend le choix d'événement simple, par contre,

l'observation des activités est limitée. Par exemple, il n'y a pas d'événements liés à l'activité du bus système.

2.2.5 Famille AMD (64 bits) :

La famille AMD $64bits$ est composée des architectures AMD *Athlon* $64bits$ et des architectures AMD *Opteron*. Le nombre et le format des compteurs sont les mêmes que pour la génération de processeurs précédents, les AMD Athon $32bits$ (figure 2.7). Par contre, le nombre d'événements passe de 23 à environ 80 [11].

Il y a sept unités fonctionnelles observables sur ces architectures, parmi elles, cinq concernant les activités de la mémoire. Ces cinq unités sont :

- *Data Cache Unit - DC*, qui permet l'observation de l'activité du cache donnée et de la table TLB (*Translation Lookaside Buffer*).
- *Instruction-Cache Unit - IC*, qui contient les structures des instructions de la mémoire cache et de la table TLB.
- *Bus Unit - BU*, qui contient le cache L2 et les caches externes.
- *Northbrigde - NB*, qui contient le bus système et le contrôleur DRAM.
- *Load/store Unit - LS* qui gère le chargement et l'enregistrement des instructions [10].

Le nombre d'événements mémoire disponibles sur cette architecture est de plus de 150. Finalement, l'architecture processeur AMD $64bits$ présente une évolution dans le nombre des événements par rapport à l'architecture précédente, même si ni le format ni le nombre de compteurs n'ont changé.

2.2.6 Synthèse

Le tableau 2.3 présente une comparaison des compteurs matériels de performance d'un large ensemble d'architectures processeurs. La plupart des architectures présentées possèdent entre 2 et 4 compteurs, exception faites pour les processeurs *Pentium 4*, Xeon et Power3. Chaque compteur ne permet de compter qu'un seul événement à la fois.

La liste d'événements dépend des unités fonctionnelles disponibles sur chaque architecture. Normalement, cette liste est divisée selon ces unités d'observation. Le nombre d'événements et le format varient d'une architecture à une autre et la complexité d'utilisation dépend du format des registres et de leur relation avec les registres de sélection.

Certaines architectures ont un grand nombre d'événements observables mais un nombre réduit de compteurs. Prenons les processeurs de la famille *Pentium 6*, où le nombre d'événements dépasse largement les 50 mais le nombre de compteurs n'est que deux. Une autre restriction vient de la conception physique des processeurs. Les événements sont liés aux unités fonctionnelles du processeur dont la lecture est limitée à une par compteur. Par

Architecture processeur	Nombre de compteurs	Nombre d'événements	Nombre d'événements mémoire
Famille Pentium 6	2	~ 75	~ 30
Famille Pentium 4 et Xeon	18	$+200$	~ 100
Famille Itanium	4	$+150$	~ 100
Athlon ($32bits$)	4	~ 20	~ 10
Athlon ($64bits$) et Opteron	4	$+150$	~ 80
MIPS 10000	2	~ 30	~ 15
MIPS 12000	4	~ 30	~ 15
Sun UltraSparc I/II/III	2	~ 25	~ 15
PowerPC 604	2	~ 25	~ 10
PowerPC 604e	4	~ 100	~ 25
Power3/II	8	$+100$	~ 50
DEC Alpha 21264	2	~ 10	aucun
DEC Alpha 21164	3	~ 49	~ 30

TAB. 2.3 *Ensemble d'architectures de processeurs et leurs compteurs matériels de performances. La taille de ces compteurs et le nombre des événements dépendent de l'architecture processeur.*

exemple, il y a des processeurs qui ne permettent pas l'observation de *cache miss* et de *cache hit* en même temps.

Une solution pour ces problèmes est le partage de compteur. Des bibliothèques de multiplexage comme PMX [80] proposent des techniques d'utilisation partagée des compteurs matériels. Ces bibliothèques font l'observation d'un événement à chaque nouvel intervalle de temps. L'intervalle de temps d'observation est défini au préalable. Le problème de cette technique est la perte de précision provoquée par l'échantillonnage en cas de forte variation d'occurrence des événements observés.

Enfin, parmi les compteurs étudiés, les architectures *Pentium 4* et Xeon sont les plus complexes au niveau de la configuration des registres (2 pour la sélection et 1 pour le compteur) et de choix d'événement. Les architectures AMD *Athlon* ($32bits$ et $64bits$), l'*Opteron* et la famille *Pentium 6* ont un seul registre de configuration par compteur, et sa configuration est moins complexe que celle des architectures *Pentium 4* et Xeon. Le problème avec l'architecture *Athlon* ($32bits$) est le nombre réduit d'événements. L'utilisation des compteurs exige un compromis entre la puissance (nombre d'événements disponibles) et la complexité de la configuration des registres.

2.3 Bilan

Le niveau de performance d'un multiprocesseur dépend, entre autres, de l'interconnexion de la mémoire et des processeurs au niveau matériel, du modèle de programmation et de l'ordonnancement au niveau logiciel.

Dans les premières sections de ce chapitre, nous avons présenté les architectures multiprocesseurs à mémoire partagée les plus courantes. Le principal problème de ces architectures est la hiérarchie mémoire. En effet le partage du bus système ou simplement de la puce de contrôle (*chipset*) peut provoquer une contention sur la hiérarchie mémoire, et par conséquent, une perte de performances. Les solutions matérielles sont très chères, dans une certaine mesure l'autre alternative est la voie logicielle avec, par exemple, les ordonnanceurs.

Le premier défi d'un ordonnanceur sur une machine multiprocesseur est le choix de la structure de la queue d'exécution, qui peut être locale ou globale. Puis, c'est la politique d'ordonnancement qui prend en compte la préemption ou la non-préemption. Enfin, le problème posé par les applications parallèles, lesquelles exigent généralement l'exécution de l'ensemble de leurs processus au même temps.

Réaliser un ordonnancement en se basant sur la disponibilité des ressources nécessite l'observation des ressources. Il y a plusieurs niveaux d'observation possible, dans l'application, dans le système d'exploitation ou directement auprès du matériel. L'observation précise est seulement possible à travers l'observation du matériel. L'apparition de compteurs matériels de performances permet cela.

La complexité d'utilisation et les incompatibilités de ces compteurs ont poussé l'apparition de bibliothèques qui définissent des interfaces standard d'accès à une ou plusieurs architectures processeurs. Des outils d'analyse et de profilage basés sur les compteurs sont ensuite apparus. Dans le prochain chapitre, nous présentons une classification de quelques-uns de ces outils.

3

Bibliothèques, outils d'analyse et systèmes adaptables

Dans le chapitre précédent nous avons présenté le classement des architectures multiprocesseurs. Nous avons montré que la conception de l'architecture de ces machines à mémoire partagée, malgré l'évolution des technologies, a un point faible au niveau de la hiérarchie mémoire, sur laquelle une demande intensive peut provoquer une contention sur le bus système ou sur la puce de contrôle. L'observation des activités matérielles, et de l'activité de la mémoire en particulier, est donc essentielle pour la compréhension du problème.

L'apparition des compteurs matériels de performance a rendu possible l'observation de l'activité au niveau matériel avec un grain fin. Malheureusement, l'exploitation de ces compteurs est complexe. En effet, le choix des événements exige de l'utilisateur la compréhension détaillée de l'architecture du processeur, d'autant plus que le nombre et le type d'événement varient selon le modèle de processeur. Qui plus est, une fois les événements intéressants sélectionnés, leur configuration n'est pas triviale non plus. Dans ce chapitre, nous commençons donc par présenter les bibliothèques d'utilisations de compteurs matériels. Ensuite, nous abordons les outils d'analyse de performance et d'observation de l'activité matérielle. Enfin, nous terminons avec la présentation des systèmes de contrôle d'exécution à travers l'observation de ressources.

3.1 Bibliothèques d'utilisation de compteurs matériels de performances

Les architectures des processeurs, les systèmes d'exploitation et les applications deviennent, au cours des années, de plus en plus complexes. Or l'optimisation des performances d'une application dépend de l'intégration parfaite de ces trois éléments. L'observation de l'exécution des applications aide à comprendre les problèmes d'intégration. Par contre, suivre l'application en cours d'exécution et extraire les interactions avec les trois niveaux (matériel, système et applicatif) est une tâche non triviale.

Depuis longtemps, il existe des outils d'observation au niveau des applications et du système d'exploitation. Cependant, les activités au niveau matériel sont restées méconnues jusqu'à l'apparition des compteurs matériels. Les premiers projets de bibliothèques d'accès aux compteurs datent du début des années 90. Ces bibliothèques avaient pour but de rendre l'utilisation des compteurs et l'analyse des performances simples.

3.1.1 HPM - *Hardware Performance Monitor*

Les architectures des processeurs CRAY, DEC Alpha, Power2 et de la famille PowerPC ont été les premières à supporter les compteurs matériels de performances. Aussi le projet HPM (*Hardware Performance Monitor*) [77] [76] [114] a fourni un outil d'accès et d'analyse de performances sur ces processeurs. Cet outil est composé d'un démon et de quelques outils de surveillance. Le démon est responsable de la prise de mesures et du multiplexage des événements, lequel permet l'utilisation alternée d'un compteur pour plusieurs événements. Les outils de surveillance permettent l'observation des événements disponibles. L'observation des activités matérielles peut être faite de deux façons : par ressource ou par processus.

L'évolution des architectures processeurs a apporté des changements sur les formats et le nombre de compteurs. À l'arrivée des architectures Power3 et Power4, le nombre des compteurs est monté à 8 et le nombre des événements à plus de 100. Si cette nouvelle génération de processeurs a rendu possible l'observation encore plus détaillée du matériel, elle a également complexifié l'analyse de performances et le choix des événements.

Le système *HPM Toolkit* [32] quant à lui définit des fonctions d'observation et une interface d'analyse de performance. L'objectif est de réduire la complexité du choix des événements et de l'analyse de performance. Une fonction d'observation est basée sur un ou plusieurs événements. Par exemple, la fonction **débit d'échec de la mémoire cache L1** est obtenue par $100 \cdot (1 - \frac{(L1_{\text{load misses}} + L1_{\text{store misses}})}{Total_{\text{load/store}}})$, où $L1_{\text{load misses}}$ et $L1_{\text{store misses}}$ sont le nombre d'opérations de lecture et d'écriture réussies sur la mémoire cache $L1$ et $Total_{\text{load/store}}$ est la somme de toutes les opérations de lecture et écriture pendant la même période. L'interface d'analyse de performance fait le lien entre le code source et les données observées.

En bref, les quatre éléments du système *HPM Toolkit* sont :
– l'outil *hpmcounter* qui permet d'accéder aux informations des compteurs matériels de performances.
– la bibliothèque *libhpm* qui fournit des fonctions Fortran, C et C++ pour des programmes séquentiels et parallèles (MPI, processus légers et hybrides).
– l'interface graphique *hpmviz* permet la visualisation des mesures prises par *libhpm*.
– le *hpmstat* qui rend possible l'observation des utilisations de ressources au niveau système.

Nous pouvons classer ces quatre éléments en deux groupes, un premier qui accèdent aux événements et les observent (*hpmcounter, libhpm* et *hpmstat*), et un second dédié à la visualisation (*hpmviz*). Les trois premiers éléments sont dépendants de l'architecture, contrairement au dernier. L'interface graphique *hpmviz*, par exemple, fonctionne sur les architectures Intel *Pentium* au dessous de PAPI (voir section 3.1.4).

3.1.2 Perfex et SpeedShop

Dans la section précédente nous avons présenté le système d'accès aux compteurs matériels de performances et d'analyse de performances HPM. Ce système ne supporte pas les architectures des processeurs du type MIPS. Pourtant les architectures des processeurs MIPS R10000 et R12000 ont, respectivement, 2 et 4 compteurs. La bibliothèque *Perfex* [118] fournit l'accès à ces compteurs.

Perfex est la couche de base pour l'utilisation des compteurs sur les architectures MIPS. Son interface ressemble à celle de l'outil HPM [62] sur l'architecture CRAY. L'outil Perfex permet la lecture et la configuration des registres et le contrôle du multiplexage des événements. La bibliothèque Perfex rend disponible aux utilisateurs 32 compteurs matériels de performances virtuels. Pour l'architecture MIPS R10000 qui contient 2 compteurs matériels réels, nous avons donc 16 compteurs virtuels par compteur réel. Les événements sont comptés par période de $\frac{1}{n}$, où n représente le nombre total d'événements. Le partage des compteurs réduit d'autant la précision des mesures.

L'interface graphique, *SpeedShop* [118], fait la correspondance entre les activités résultant du décompte des événements et le code source. SpeedShop permet le profilage à grain fin en identifiant même le nombre de transactions d'un événement pour chaque ligne du code source.

3.1.3 Vtune - *VTune Performance Analyzer*

Les premiers processeurs Intel à supporter les compteurs matériels de performances ont été ceux de la famille *Pentium 6* (*Pentium Pro*, *Pentium MMX*, *Pentium II* et *Pentium III*). Ces architectures ont 2 compteurs matériels et plus de 50 événements observables

(voir la section 2.2.1). La première bibliothèque d'accès aux compteurs sur ces processeurs, *PCT - Performance Counter Tool*, a servi de base du projet Vtune [6].

Le projet Vtune, *VTune Performance Analyzer*, supporte actuellement toutes les architectures processeurs Intel 32 et $64bits$. L'ensemble des outils Vtune est constitué de trois éléments : une bibliothèque d'accès aux compteurs et deux outils d'analyse de performances.

La bibliothèque de base de Vtune, *Counter Monitor*, permet la sélection et l'observation des événements. L'observation est faite selon l'utilisation de la ressource choisie au cours du temps d'exécution. L'outil d'analyse de performances, *Call Graph Profiling*, présente l'utilisation des ressources selon chaque fonction du programme représentée sur le graphe. Puis, le *Time-and-Event-Based* trace l'utilisation des ressources par fonction au cours de l'exécution.

3.1.4 PAPI - *Portable Programming Interface for Performance*

Les bibliothèques et les outils d'analyse de performances présentés dans les sections précédentes ne font le support que de quelques architectures de processeurs, généralement, d'un même fabricant. La principale raison de ces restrictions est l'incompatibilité des formats des compteurs, lesquels changent pour chaque architecture (voir section 2.2).

Dans ce contexte, le projet PAPI *Portable Programming Interface for Performance* [73] [22] propose une interface de programmation indépendante de l'architecture d'utilisation de compteurs matériels de performances. L'objectif principal du projet est de rendre transparente l'utilisation des compteurs, c'est à dire la configuration, le choix d'événement et la lecture, sur le plus grand nombre d'architectures possible.

La portabilité de PAPI s'appuie sur la définition d'une couche de programmation de haut niveau et d'une autre de traitement des compteurs (figure 3.1). La première, la couche portable (*Portable Region*), fournit l'interface de programmation des compteurs aux utilisateurs. La gestion de la mémoire, le multiplexage de compteurs, le support de traçage des processus légers, la manipulation des structures de données, le contrôle du dépassement de valeur (*counters overflow*) et le traitement de la réentrance de code (*thread safety*) sont, eux aussi, de la responsabilité de cette couche [23].

La deuxième couche est dépendante de la plate-forme. Elle est spécifique au format de chaque compteur défini sur chaque architecture. Par exemple, elle utilise *Perfctr* [90] sur les architectures Intel $32bits$ et *Perform* [3] sur les architectures Intel $64bits$.

Enfin, la bibliothèque PAPI dispose d'une liste d'événements prédéfinis composée des événements les plus couramment disponibles sur l'ensemble des architectures supportées. Cela étant, la configuration des événements spécifiques, natifs à une architecture n'est pas mise à l'écart mais elle exige une forte connaissance de l'architecture en question.

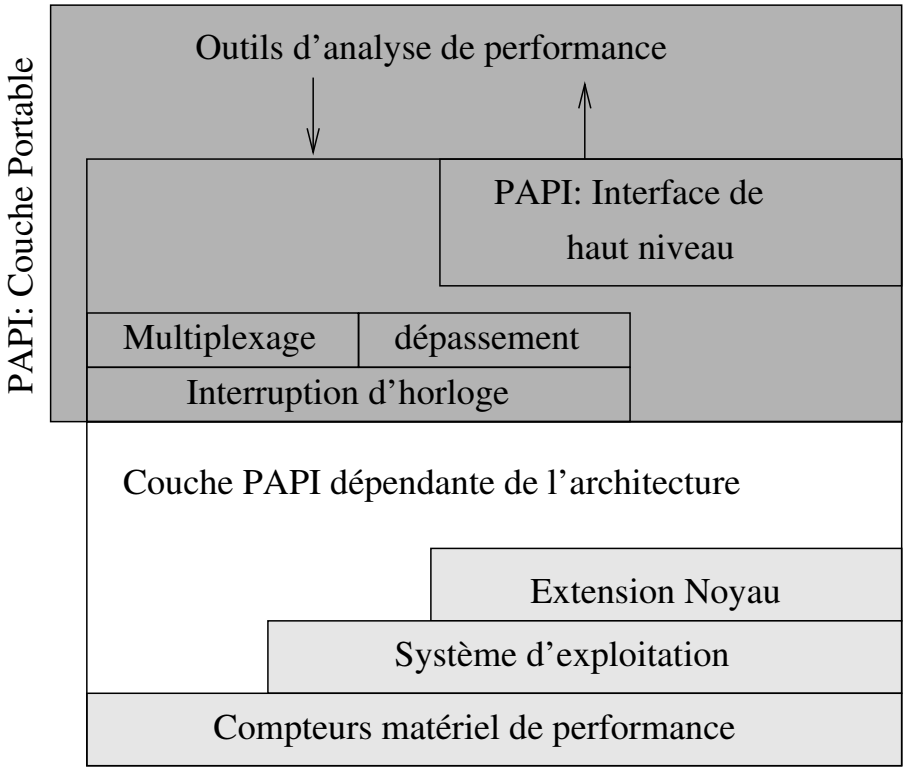

Figure 3.1 *L'architecture de la bibliothèque PAPI est divisée en deux parties : la première partie, la couche portable, fait le contrôle des structures de données et permet la configuration du système. La deuxième, la couche dépendante, permet l'accès aux compteurs selon la plate-forme d'exécution.*

Les événements natifs doivent être formatés directement par l'utilisateur alors que les événements prédéfinis sont automatiquement adaptés par la bibliothèque. Lorsque les programmes n'utilisent que des événements prédéfinis, ils peuvent s'exécuter sur l'ensemble des architectures supportées sans modifications sur les définitions des événements.

3.1.5 PCL - *Performance Counter Library*

Le projet PCL (*Performance Counter Library*) [19] [18] se place dans le même contexte que PAPI (section 3.1.4). Il permet l'accès à un ensemble de compteurs matériels de performances sur différentes architectures, avec une API unique. Le système PCL est composé de deux couches lui aussi : une couche indépendante de l'architecture offrant l'interface de programmation de haut niveau, et une autre dépendante fournissant la bibliothèque d'accès aux compteurs.

L'interface de programmation dispose des fonctions d'arrêt, de démarrage et de lecture

des compteurs. Les langages de programmation supportés sont C, C++, Fortran et Java. La couche dépendante est juste une interface au pilote d'accès aux compteurs. Le pilote d'accès dépend de l'architecture, par exemple, il utilise Perfctr [90] pour les architectures Intel *Pentium* $32bits$ et la HPM [32] pour les architectures Power3 et Power4.

Les événements sont divisés en quatre catégories :
- *hiérarchie mémoire*, qui contient les événements liés aux registres, aux mémoire *caches* et à la mémoire principale.
- *instructions*, qui compte entre autre le nombre d'instructions entières, flottantes ou les cycles d'horloge.
- *état des unités fonctionnelles*, qui compte le nombre de cycles pendant lequel une unité est occupée.
- *débits*, qui compte le nombre d'opérations flottantes par seconde ou des caches miss entre autres.

Le principal objectif de PCL est la définition d'un groupe uniforme d'événements et des appels de fonctions pour tous les systèmes. Pour cette raison, aucun événement spécifique à une architecture n'est défini dans l'ensemble des événements PCL.

3.1.6 Bilan

L'apparition des compteurs matériels de performances a permis l'observation précise des activités des ressources des processeurs. Cependant, l'utilisation de ces compteurs s'est montrée complexe. Cette complexité vient de l'hétérogénéité des formats des compteurs et des types d'événements (voir section 2.2).

Les premiers travaux sur le sujet envisageaient de simplifier l'utilisation des compteurs sur un ensemble d'architectures, sans se préoccuper de la portabilité. Dans ce cadre, nous avons présenté le projet HPM, qui propose une bibliothèque pour les architectures processeurs CRAY, DEC Alpha, Power3 etc, Perfex, pour les architectures MIPS et Vtune, pour les architectures Intel (voir la tableau 3.1). Ces trois bibliothèques disposent d'une couche d'accès aux compteurs et des outils d'analyse de performances visuels, *hpmviz* sur HPM, *Speedshop* sur Perfex et *Call Graph Profiling* ou *Time-and-Event-Based, System-Wide Sampling* sur Vtune.

L'utilisation de ces outils d'accès aux compteurs et d'analyse de performances était restreinte à cause des choix de leurs développeurs, comme, par exemple, le système d'exploitation supporté. Malheureusement, ces choix n'allaient pas toujours dans le sens des besoins des utilisateurs. Par exemple, l'outil Vtune ne supportait que le système d'exploitation Windows. Des projets comme Lperfex [15] et Rabbit [58] ont donc proposé des bibliothèques plus flexibles que celles existantes. En effet, leurs codes sources est accessible, elles contiennent un petit nombre de fonctions simples à utiliser et il est même possible de les modifier. Les plates-formes supportées par ces deux projets sont les architectures Intel (*Pentium*) et AMD sur le système d'exploitation Linux. Par ailleurs, en ce

Projet	Plate-forme		Bibliothèque PMC
	Architecture	Système d'exploitation	
HPM	POWER4, POWER3, 604, 604e	AIX	
Perfex	MIPS R10K, R12K, R14K	IRIX	
Vtune	Intel Pentium	Linux/Windows	
Lperfex	Intel jusqu'au Pent. III/AMD (32bits)	Linux	
Rabbit	Intel jusqu'au Pentium III	Linux	
PCL	POWER4, POWER3, 604, 604e	AIX	HPM
	AMD Athlon/Opteron, Intel Pent.	Linux	PerfCtr
	UltraSparc I, II & III	Solaris	
	MIPS R10K, R12K, R14K	IRIX	
	Alpha	Linux	
PAPI	POWER4, POWER3, 604, 604e	AIX	HPM
	AMD Athlon/Opteron, Intel Pent.	Linux	PerfCtr
	Intel Itanium I & II	Linux	PerfMon
	Intel jusqu'au Pentium III	Windows	
	Cray T3E, Cray X1	Unicos	
	UltraSparc I, II & III	Solaris	
	MIPS R10K, R12K, R14K	IRIX	
	Alpha	Tru64 Unix/Linux	

TAB. 3.1 *Tableau de comparaison des plates-formes supportées par les bibliothèques d'accès aux compteurs matériels de performances. Pour chaque projet, nous avons l'architecture et les systèmes d'exploitation supportés. Quand une bibliothèque en utilise une autre pour l'accès aux compteurs, cette dernière est notée dans la colonne Bibliothèque PMC (*Performance Monitor Counter*).*

qui concerne le projet Lperfex, un des objectifs était de permettre la surveillance des ressources sur des grappes. Enfin, les deux projets fournissent des éléments pour l'extension et l'adaptation de leurs bibliothèques selon les besoins des utilisateurs.

Malgré les efforts de ces deux projets, les problèmes de portabilité et de complexité du choix des événements persistaient. La division des bibliothèques en couches, et la définition d'un ensemble d'événements génériques ont été les solutions proposées par deux autres projets, PAPI et PCL. Comme il a été présenté dans les section 3.1.4 et section 3.1.5 ces deux bibliothèques sont basées sur deux couches : une couche dépendante et l'autre indépendante de l'architecture. Les couches dépendantes s'occupent de l'accès spécifique aux compteurs de chaque architecture et les couches indépendantes des aspects génériques, du type liste d'événements ou structures de données. Le projet PAPI apporte plus de fonctionnalités que PCL, notamment le multiplexage [80] d'événements.

En réduisant la complexité d'utilisation des compteurs matériels de performances, les bibliothèques que nous venons de présenter ont permis d'augmenter le nombre d'étude

analysant finement le comportement des applications. Des outils d'analyse de performances, de profilage et même des systèmes adaptables ont vu le jour en se basant sur les bibliothèques présentées. Les prochaines sections seront consacrées à ces outils.

3.2 Outils d'analyse de performances

Les outils d'analyse de performances sont importants pour la mise au point de programmes et pour l'optimisation de l'utilisation des ressources. Il est essentiel d'avoir une idée précise de la manière dont un programme s'exécute pour identifier les problèmes et améliorer les performances. De la même façon, il est important de connaître précisément l'utilisation des ressources afin d'éviter sa surcharge, et par conséquent, une perte de performances de l'ensemble des applications.

Mieux comprendre un système c'est bien l'observer. L'observation d'un système consiste à faire des prises de mesures sur un ou plusieurs de ces composants. À un niveau d'observation élevé (gros grain), un composant peut être une ressource ou une application. À un niveau plus fin, le composant peut être une fonction d'une application ou une procédure d'accès à une ressource.

Prenons d'abord l'observation d'un programme. Dans des cas simples, il arrive que la seule observation des résultats d'un programme permette de déterminer l'origine de certaines erreurs ou d'identifier des problèmes d'inefficacité. Malheureusement, elle dépend des connaissances empiriques du programmeur et ce n'est ni généralisable, ni suffisamment précis pour être applicable à des applications plus complexes.

L'observation par ressource n'est pas différente. Les informations disponibles au niveau système peuvent être facilement récupérées, par contre au niveau matériel, cela exigent plus de connaissances spécifiques. Dans le cadre de ce travail, nous parlons du niveau applicatif pour les opérations faites par les applications, du niveau système pour celles faites par le noyau et du niveau matériel pour celles faites par les processeurs, mémoires, disques etc.

Techniques d'observation : Les techniques d'observation, de surveillance (*monitoring*) et de profilage (*profiling*) [54] permettent d'observer l'exécution des applications sans les arrêter. L'objectif principal de ces techniques est de tracer l'exécution de l'application avec coût additionnel minimum. La surveillance consiste à faire l'observation des activités sur les ressources. Cette observation se fait par des outils dédiés à cette tâche qui sont extérieures aux applications. Elle peut être effectuée au niveau matériel, au niveau système ou au niveau applicatif. Le profilage de l'exécution d'un programme consiste à estimer le temps passé dans chaque sous-partie d'un programme. Cela permet de dresser un *profil d'exécution* à partir duquel il est possible de déterminer les parties de code les plus utilisées, c'est-à-dire celles où le programme passe le plus de temps. Ce sont ces

zones que l'on cherchera à optimiser en priorité.

Enfin, la complexité des programmes et des architectures exige des outils d'analyses et des techniques d'observations afin de comprendre les problèmes d'inefficacités. L'évolution des technologies matérielles présentées dans la section 2.1.1 soulève des problèmes d'inefficacités venus de la conception de l'architecture des multiprocesseurs. Nous nous sommes basés sur la surveillance dans le but d'identifier ces problèmes et sur le profilage pour caractériser le comportement des applications. L'observation par surveillance peut être faite sur deux types d'architectures, les **architectures simulées** et les **architectures réelles**. Les prochaines sections sont consacrées à la présentation des outils sur ces deux types d'architectures.

3.2.1 Observation sur une architecture simulée

Les simulateurs reproduisent le comportement d'architectures réelles en utilisant une abstraction de leurs composants. Le simulateur est capable de contrôler toutes les opérations sur le système simulé en permettant à l'utilisateur de tester des actions et d'observer avec détails les réactions. Cependant, plus le niveau de détail de la simulation est important, plus le temps de réponse augmente et le niveau d'abstraction baisse.

Dans le chapitre 2 il est fait état de la complexité des architectures des machines moderne. Les simulateurs sont utilisés par les fabricants des processeurs pour tester les problèmes de conception processeurs. Ils sont aussi utilisés par les développeurs de logiciels. En effet, grâce aux simulateurs, ces derniers ont la possibilité de tester leurs applications sur plusieurs architectures même sans les avoir chacune à disposition. Dans le cadre de cette thèse, nous allons décrire plus spécifiquement les simulateurs prenant en compte la hiérarchie de la mémoire.

L'observation du comportement de la hiérarchie mémoire sur les multiprocesseurs est supportée par le simulateur SIMOS [59]. Celui-ci prend en compte la charge d'un système réel, il permet également le contrôle du niveau de détail de la simulation et enfin il possède différents modèles mémoire. L'utilisation de ce système met à disposition les tests de performances de programmes sur des architectures simulées en proposant différents rapports de vitesses entre la mémoire et les processeurs.

Deux autres simulateurs permettent l'analyse détaillée de la hiérarchie mémoire de différents points de vue : SIGMA [33] et SIP [17]. SIGMA permet l'observation des activités dans la TLB, des caches de données et fournit des mesures de performances et des statistiques sur l'utilisation mémoire. SIP [17], basé sur l'environnement Simics [75], a comme cible l'identification des mauvaises allocations des structures de données.

Enfin, des travaux comme MemSpy [79] et MTOOL [52] ont montré l'importance des méthodes de collecte de données pour l'identification de goulots d'étranglements. MemSpy fait la collecte orientée sur les données et sur le code source alors que MTOOL

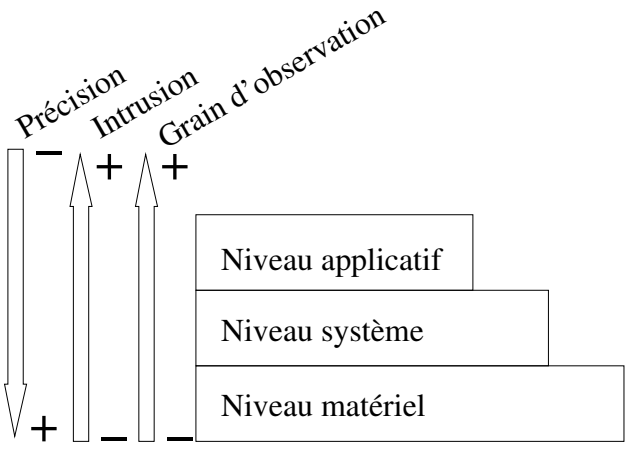

Figure 3.2 *Les trois caractéristiques, la précision des mesures, l'intrusion et la granularité de l'observation changent selon le niveau d'observation.*

marque juste les temps d'entrée et de la sortie de chaque fonction.

L'utilisation des simulateurs est très utile pour l'étude du problème de contention sur la hiérarchie mémoire. Cependant, les limitations liées au niveau d'abstraction offert ne permettent pas l'approfondissement des observations contrairement aux systèmes réels. Pour cette raison, nous avons étudié les outils d'analyse sur des architectures réelles.

3.2.2 Observation sur une architecture réelle

Dans les sections précédentes nous avons présenté les deux techniques d'observation, la surveillance et le profilage, ainsi que quelques environnements de simulation sur la hiérarchie mémoire. L'observation des systèmes réels permet d'analyser le comportement des programmes et des ressources sur une machine. Le principal problème de l'observation des applications ou de l'usage des ressources sur une architecture réelle est la modification de l'exécution provoquée par l'outil d'observation.

Le choix du niveau d'observation sur un système réel dépend d'un compromis entre le niveau de détail désiré par l'utilisateur et l'impact que l'instrumentation provoque sur le système. L'observation peut être instrumentée à trois différents niveaux (figure 3.2) : **applicatif**, **système** et **matériel**. Afin d'établir des comparaisons entre les niveaux d'observation, nous avons regardé trois paramètres : la précision de l'observation, l'intrusion et la granularité.

Le premier niveau, applicatif, facilite l'observation des applications en gros grains, il mesure les temps d'exécution de chaque fonction et peut compter le taux d'utilisation des structures de données. Néanmoins, la précision des mesures est plus faible. Enfin, le

surcoût de l'instrumentation des fonctions d'observation à ce niveau peut-être important, le niveau d'intrusion est donc élevé.

Le deuxième niveau dispose de plus d'informations, par exemple, l'allocation mémoire ou l'identification des appels système. Nous obtenons donc une plus petite granularité d'observation. La précision est ainsi plus fine par rapport au niveau applicatif, et le surcoût également car la prise de mesure se base sur des appels système.

Finalement, au niveau matériel, la granularité est la plus fine et l'intrusion la plus faible, puisque les mesures sont faites directement sur les instructions exécutées sur le processeur. La précision est donc la plus haute dans ce cas car il est possible d'utiliser l'ensemble des événements matériel disponible sur l'architecture.

Dans la section 2.2, nous avons présenté les compteurs matériels de performance. Ces compteurs matériels permettent des prises de mesures avec un coût négligeable. Dans cette section, nous allons donc présenter quelques outils d'observation pour des systèmes réels basés sur l'utilisation de ces compteurs matériels de performances.

L'observation d'un système réel dans ce travail de thèse a deux objectifs : la caractérisation d'une application et le contrôle de l'équilibre de l'utilisation des ressources matérielles de la machine. Nous pensons atteindre le premier objectif à travers le profilage et le deuxième par la surveillance.

3.2.2.1 Outils d'observation par profilage

Dans la section bibliothèque de compteurs matériels de performances (section 3.1), nous avons déjà présenté quelques outils de profilage basés sur les compteurs matériels. Nous les rappelons : dans le projet HPM Toolkit [32] l'outil *hpmviz*, dans le projet Perfex [118] l'outil *SpeedShop* et dans le projet Vtune [6] les outils *Call graph* et *Time and Event-Based*. Ces outils proposent simplement l'observation du comportement des fonctions du programme selon un ou plusieurs événements matériels choisis.

L'observation du comportement des applications permet l'identification des problèmes sur des régions du code source. Mais, le profilage sur les structures de données lié à l'utilisation des ressources permet en plus l'identification exacte de la source du problème. Le travail d'extension de l'outil d'analyse de performance *Sun ONE Studio* [63] a montré l'importance de l'analyse des structures de données pour l'identification des goulots d'étranglements mémoire. Par exemple, Il rend possible à l'utilisateur de connaître l'utilisation de la mémoire cache par les fonctions, par les instructions et finalement par les structures de données.

Ainsi, l'analyse d'un programme exige l'interaction d'outils de profilage avec l'application. Dans ce contexte, le projet SvPablo [35] propose un langage pour une analyse de performance indépendante de l'architecture, basé sur un modèle de fichier de visualisation Pablo [93] (SDDF - *Self-Defining Data Format*). Grâce à ce langage, SvPablo

supporte l'instrumentation automatique ou interactive. L'instrumentation automatique est faite par le compilateur, lequel ajoute des capteurs dans le code binaire de l'application. L'instrumentation interactive est classique, elle est effectuée par l'insertion de fonctions dans le code source de l'application manuellement.

Plus récemment, nous avons vu apparaître les outils CATCH [96] *Call-graph-based Automatic Tool for Capture of Hardware-performance-metrics* et SCALEA [107] *A Performance Analysis Tool for Distributed and Parallel Programs*. Ces deux outils proposent l'observation des applications puis une instrumentation automatique, pour les bibliothèques de programmation parallèle MPI et OpenMP. De plus, SCALE supporte les langages Fortran90 et HPF. CATCH, quant à lui, a été développé au dessous de la bibliothèque de classes C++ DPCL [34] et met à disposition les informations des trois niveaux, matériel, système et applicatif.

L'idée générale utilisée pour CATCH permet l'analyse globale du système. Par contre, l'optimisation de l'ensemble des instructions n'est possible que grâce à un profilage très fin. C'est dans ce contexte que DCPI [31] [30] propose un système de collecte et d'analyse de l'activité des processeurs, afin de permettre l'optimisation des applications.

Pour finir, nous présentons les travaux autour des outils de profilage basés sur la bibliothèque PAPI. Les deux premiers ont été PAPI Perfometer et PAPI Profometer [22] [72], qui présentaient respectivement une simple interface graphique de visualisation et un histogramme d'événements. Suite à ces outils est apparu Paraprof [16] qui propose un *framework* de composants pour l'analyse de performances basée sur le profileur TAU [14]. Plus récemment enfin, le projet KOJAK [115] [116] qui propose un outil d'analyse des programmes en OpenMP et MPI est né.

Les outils de profilage présentés permettent ainsi la caractérisation des applications sur plusieurs niveaux d'observations et l'identification de goulots d'étranglements.

3.2.2.2 Outils d'observation par surveillance

L'observation par profilage d'une application permet une analyse verticale de l'utilisation des ressources. Les ressources ainsi prises en comptes sont celles utilisées par une seule application. Or comme les applications s'exécutent indépendamment les unes des autres, ce type d'observation ne permet pas de caractériser l'utilisation globale d'une ressource. L'observation par surveillance, par contre, capture les données par ressource, ce qui permet l'analyse horizontale de l'utilisation des ressources (voir figure 3.3).

Paradyn [82] est un des projets qui utilisent l'observation par surveillance. Son architecture est composée de plusieurs modules, dont un de gestion des données (*Data Manager*), et un d'analyse automatique (*Performance Consultant*). Le fonctionnement est le suivant : Le premiers module permet la surveillance des activités de l'ensemble des applications contrôlées par le système. Puis le deuxième vérifie l'utilisation des ressources,

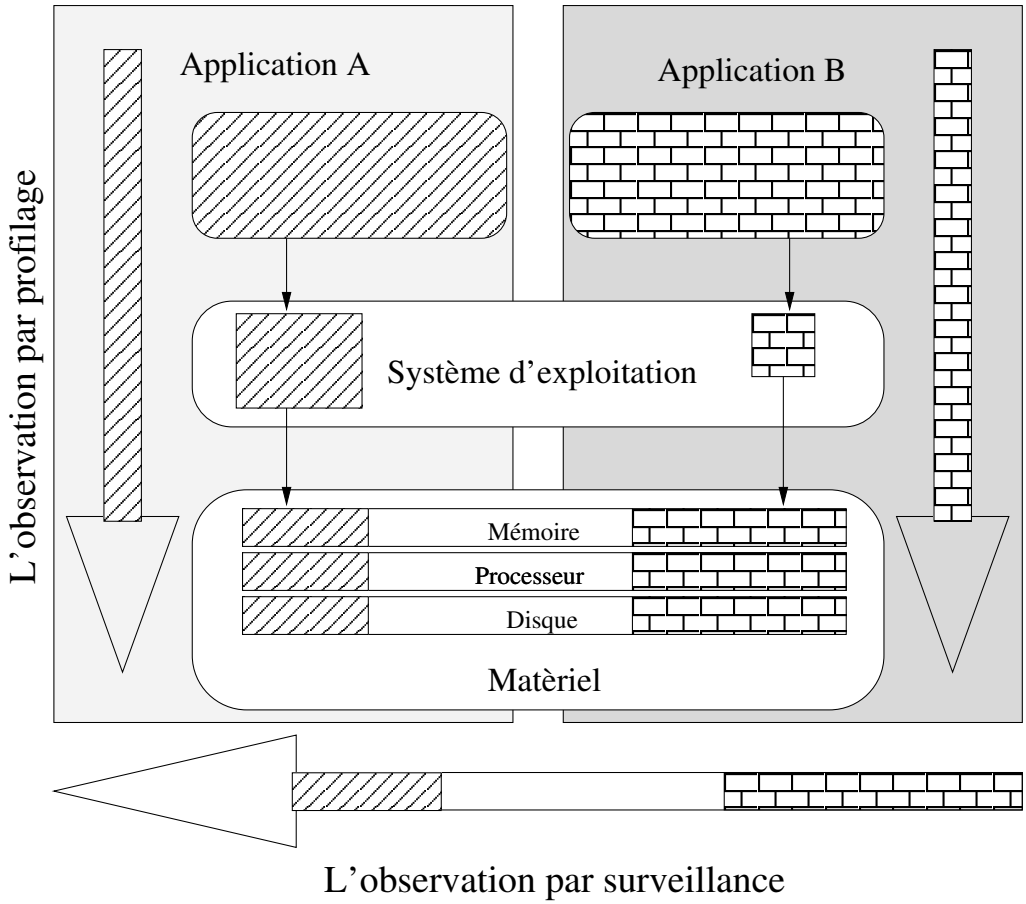

Figure 3.3 *Les façons de faire l'observation sont : par application, avec une vision verticale, et par ressource, avec une vision horizontale.*

comme les processeurs ou la mémoire, afin d'identifier des goulots d'étranglements en observant l'occupation de chaque ressource.

Le module d'analyse travaille sur des hypothèses : si le temps d'un appel bloquant est long, par exemple, il observe les requêtes en provenance du réseaux. Si le temps d'accès à la mémoire est long, il observe celles de la mémoires. Ainsi, le système change la cible d'observation automatiquement lors de la recherche du problème. Néanmoins, il existe toujours des cas où l'usage d'hypothèses ne suffit pas pour l'identification du problème. C'est dans ce contexte que le projet DeepStart [97] a proposé une extension du module d'analyse afin d'observer les applications à travers un graphe de fonctions. Cette technologie permet une identification plus rapide des goulots d'étranglements.

Même si le projet Paradyn fait l'observation de manière extérieure à l'application et s'il regarde un ensemble de ressources, son objectif reste l'amélioration des performances d'applications. Il ne cherche donc pas l'optimisation de l'utilisation globale des ressources. Or comme dans le domaine des grappes et des grilles la ressource réseau est

essentielle, il est apparu ces dernières années des applications de surveillance réseau.

Le projet MAGNET [49] ***M**onitoring **A**pparatus for **G**eneral ker**N**el-**E**vent **T**racing*, par exemple, propose une architecture pour la surveillance à grain fin des événements sur un ensemble de noeud d'une grappe ou d'une grille. Son but principal est de mettre à la disposition des utilisateurs des outils de prises de mesures sur les ressources, pour permettre le contrôle des applications en fonction de leur utilisation. Le système MUSE [50] *MAGNET User-Space Environment*, extension de MAGNET, offre un outil utilisateur pour aider l'analyse et le filtrage des données du démon MAGNET. Parmi les outils de surveillance plus généraux enfin, nous pouvons citer le projet Remo [39] qui propose un environnement de surveillance des réseaux. Une des plus importantes caractéristiques du moniteur du système Remo est l'intégration de la topologie du réseau et des prises de mesures.

La méthode d'observation par surveillance permet donc l'observation par ressources liées ou non à une application. Ce type de méthode se base normalement sur des capteurs, lesquels sont extérieurs à l'application. Les informations échantillonnées peuvent servir à des statistiques d'utilisations ou à l'adaptation de l'ensemble du système à la disponibilité des ressources.

3.2.3 Bilan

L'étude des problème des performances des applications repose sur l'identification des goulots d'étranglements, dont la formation est due aux déséquilibre entre les besoins des applications et la capacités des ressources. L'identification de ces goulots peut être faite à travers l'observation de l'exécution des application sur les architecture concernée. L'observation peut être envisagée dans deux contextes, tout d'abord sur une architecture réelle, ou bien sur une abstraction d'une architecture réelle, c'est-à-dire une architecture simulée. Au cours de ces dernières sections, nous avons présenté les principaux outils d'analyse de performances sur des architectures réelles ou simulées.

Les simulateurs étudiés permettent de tester plus facilement l'exécution d'une application sur différentes configurations matérielles. Cela permet de multiplier le nombre d'expériences. Ainsi, on peut trouver une architecture simulée plus adaptés pour une application donnée. Malheureusement, l'abstraction de l'environnement réel offert par les simulateurs peut cacher des éléments essentiels à la parfaite compréhension du comportement de l'application.

L'autre solution est l'observation des architectures réelles. Nous avons classé en deux groupes les types d'observations des architectures réelles : celles dédiées aux applications et celles dédiées aux ressources. L'observation par profilage fait l'analyse de l'exécution d'une application, donc, elle est dédiée aux applications. Les outils basés sur l'approche par profilage ne permettent l'étude des activités des ressources que pour l'application choisie. Ensuite, l'observation par surveillance est celle dédiée aux ressources. Elle fait

l'observation horizontale, c'est à dire par couche (niveau matériel, niveau système et niveau applicatif). Les outils basés sur cette approche permettent l'observation de l'ensemble du système. Ce genre d'outil est adapté à l'administration des machines ou à l'étude des statistiques d'accès.

L'observation des architectures réelles est de plus en plus utilisée pour le contrôle adaptatif des applications en cours d'exécution. L'apparition des outils de profilage et de surveillance basés sur des compteurs matériels de performances a facilité l'analyse et l'identification des problèmes au niveau matériel. La prochaine section est consacrée aux systèmes adaptables.

3.3 Systèmes adaptables

Un système est, d'une façon générale, un ensemble d'objets qui appartiennent à un même environnement. Les objets d'un système informatique sont : les ressources matérielles, les structures de données et les processus. Les processus allouent des structures de données et des ressources matérielles pour s'exécuter. Les ressources sont limitées, donc, les allocations le sont aussi. L'ensemble des processus en cours d'exécution dans un même environnement doit, de ce fait, adapter ses besoins aux ressources disponibles.

3.3.1 Définition d'un système adaptable

Un système adaptable est un système capable d'ajuster les conditions de son exécution en fonction de ses besoins ou de ceux d'un autre système [13]. L'adaptation d'un système informatique peut consister à optimiser l'utilisation de l'ensemble des ressources de la machine (le processeur, la mémoire, le disque et etc). Avoir un système adaptable est en fait souhaitable en vue de deux objectifs : **l'optimisation des performances d'une application** et **l'utilisation optimale de toutes les ressources**. Le premier est le souhait de l'utilisateur, lequel veut avoir le maximum de performance pour son application et le deuxième, celui de l'administrateur qui souhaite rentabiliser au mieux sa machine.

Ainsi, tant que l'utilisation des ressources est équilibrée et que les besoins des applications sont satisfaits, les visions de l'utilisateur et de l'administrateur sont respectées. Mais dès qu'une ou plusieurs ressources sont saturées, les performances des applications chutent, et par conséquent le rendement de la machine s'écroule [25]. A l'inverse, la sous utilisation des ressources déplaît à l'administrateur. Un des principaux objectifs de l'adaptation d'un système est donc de maintenir un équilibre dans l'utilisation des ressources.

Cette tâche n'est néanmoins pas triviale, car la demande et la disponibilité des ressources évoluent en permanence. Ainsi quand les besoins des applications dépassent largement la capacité d'une ressource, il apparaît les goulots d'étranglements. Il est alors

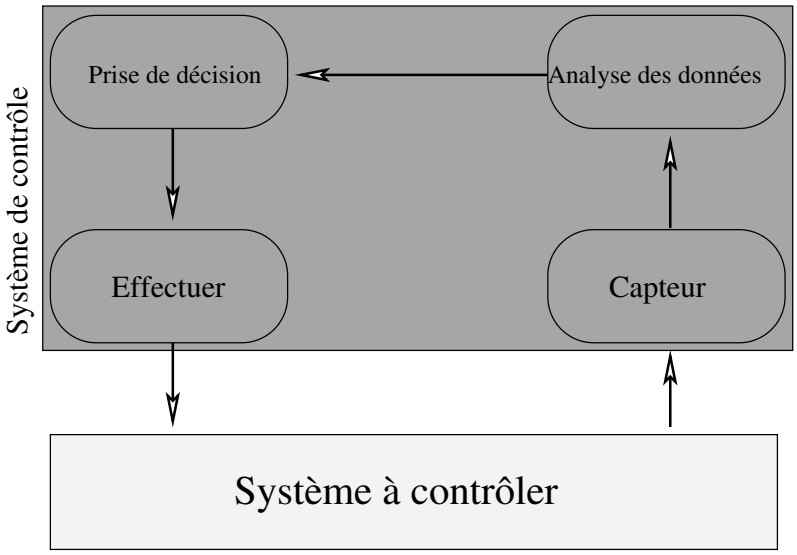

Figure 3.4 *L'architecture générique d'un système adaptable est basée sur un système de contrôle placé au dessus du système à contrôler. Le système de contrôle est composé de quatre modules réalisant un asservissement : capteur, analyse des données, prise de décision et effecteur.*

nécessaire de contrôler le système, c'est à dire contrôler l'allocation des ressources.

En fait, le contrôle de système peut être fait par des systèmes adaptables. L'architecture générique de ce type de système est présentée dans la figure 3.4. Il se divise en deux éléments, le *système de contrôle* et le *système à contrôler*. Le système de contrôle est composé par quatre modules :

- le module *capteur* qui surveille l'activité sur les ressources engendrée par les applications.
- le module d'*analyse des données* qui fait les traitements statistiques et, si nécessaire, le filtrage des données. C'est ce module qui réalise l'identification des goulots d'étranglements.
- le module *de prise décision* qui applique les règles de gestion du système. Selon la politique retenue il favorise l'optimisation des performances des applications ou le rendement de la machine.
- enfin, le module *effecteur* qui met en application les ordres du module de décision.

Le système de contrôle de systèmes adaptables que nous venons de présenter est toujours le même, cependant son positionnement par rapport au système à contrôler peut changer.

Dans un *système adaptable* simple (figure 3.5) le système de contrôle est un élément extérieur aux applications. Son capteur interagit directement sur le système d'exploitation et sur le matériel, et éventuellement, il prend des mesures sur les applications (non représenté dans la figure). L'analyse repose sur des données globales de l'utilisation des

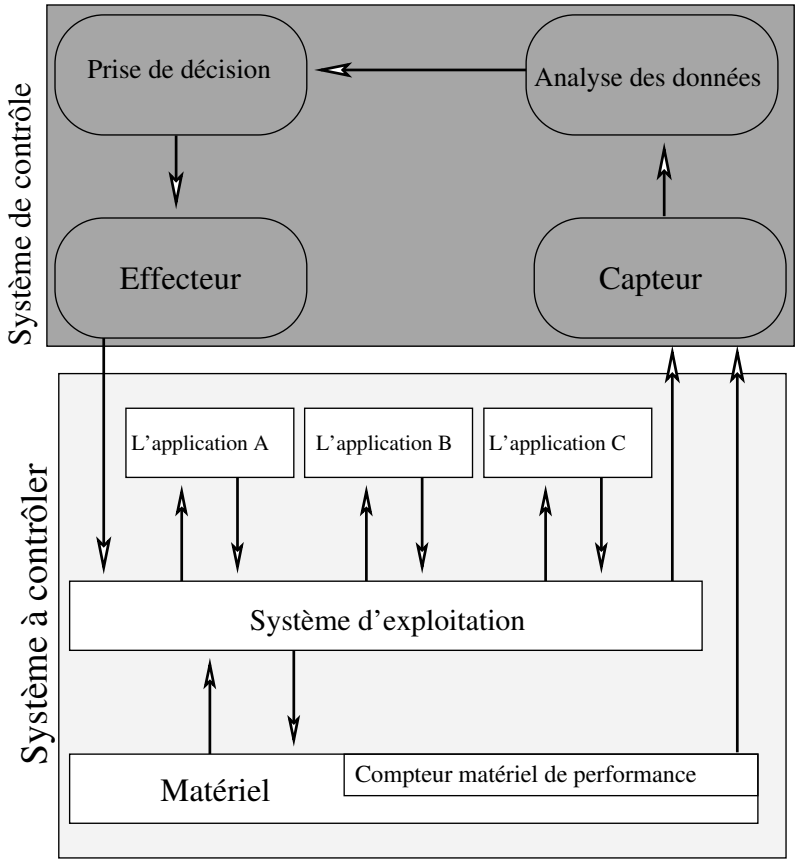

Figure 3.5 *Le système de contrôle reste celui de l'architecture générique. Par contre, les éléments de la couche système à contrôler montrent que l'observation est faite à l'extérieur des applications et que l'effectuer agit sur le système d'exploitation.*

ressources et sur les besoins spécifiques de chaque application. Son module de prise de décisions est basé sur les informations fournies par le module d'analyse de données et il fait le traitement des décisions selon les règles exigées. Les règles dépendent des objectifs et sont définies par l'administrateur du système. Enfin, le module effecteur met en application la politique (les règles définies) à travers des appels du système d'exploitation.

Une sous catégorie des systèmes adaptables est celles des *systèmes auto-adaptables* (figure 3.6). Dans le cas de ceux-ci, le système de contrôle fait partie de l'application et le système à contrôler est constitué de l'application elle-même uniquement. Ce genre de système n'agit donc que sur une application, lui-même, mais doit s'adapter aux ressources partagées. Son capteur peut obtenir des informations directement de l'application, du système d'exploitation et du matériel. Ses modules d'analyse et de prises de décisions sont adaptés aux besoins de l'application et la politique envisagée est la maximisation de la performance. Enfin, les décisions prises sont appliquées par l'effecteur de manière interne, directement sur le noyau de l'application.

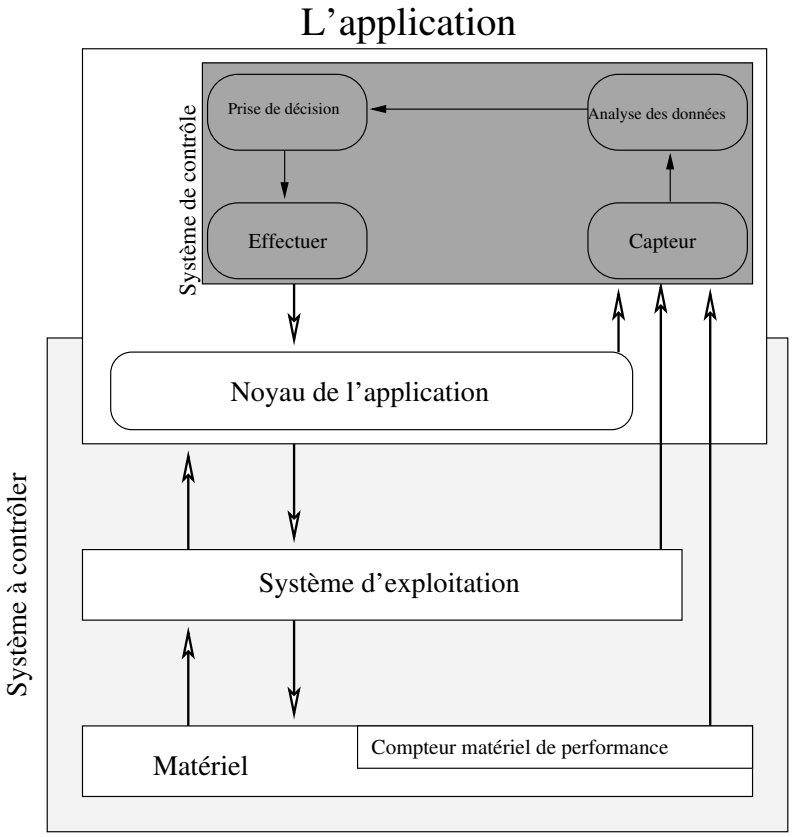

Figure 3.6 *La sous catégorie de systèmes adaptables, les systèmes auto-adaptables, contient le système de contrôle dans l'application et le système à contrôler vu pour cette application n'a que lui-même à observer. L'observation est faite sur l'application, sur le système d'exploitation et sur le matériel. Ainsi, l'effectuer intervient directement sur l'application.*

Dans la prochaine section nous présentons quelques systèmes adaptables.

3.3.2 Exemple de systèmes adaptables

Dans un grand nombre de domaines, l'adaptation peut servir à améliorer la qualité des services. Par exemple, dans les environnements distribués et parallèles, le nombre des processus des applications peut être très important et donc la gestion des ressources devient très complexe. Dans le cas des grilles notamment, le simple choix d'exécuter une tâche doit tenir compte de la capacité et de la disponibilité du réseau, des processeurs, de la mémoire ou même du disque. Celles-ci varient au cours du temps.

C'est dans ce contexte que le projet CODE [100] (*Control and Observation in Distributed Environments*) propose un système adaptable de gestion des ressources, des ser-

vices et des applications. L'architecture de ce système est composée de trois éléments : l'effecteur, le capteur et le gestionnaire, qui reçoivent respectivement les noms d'*Actor*, d'*Observer* et de *Manager*. L'effecteur et le capteur ont les caractéristiques standards des système adaptables définies dans la section précédente, hormis que l'on a en fait affaire à un groupe d'effecteurs et à un groupe de capteurs. L'élément gestionnaire, par contre, regroupe les deux fonctionnalités du module d'analyse des données et des prises de décisions. Actuellement, le système CODE permet l'optimisation de l'utilisation des ressources et l'analyse de statistiques issues des outils GRAM (*Globus Resource Allocation Manager*) et GIS (*Grid Information Service*).

Dans ce premier exemple, la quantité des ressources est la principale difficulté pour optimiser le système. Néanmoins, les systèmes adaptables sont aussi importants pour des environnements avec un nombre de ressources plus réduit. Dans le domaine du commerce électronique, par exemple, un des plus grands besoin est l'optimisation du temps de réponse perçu par les clients et le contrôle du nombre de clients connectés sur un fournisseur.

Ces systèmes ne sont pas constitués d'un grand nombre de machines, par contre, la prise de décision doit être rapide. Un des travaux que l'on trouve dans ce domaine est une modification d'un serveur Web apache, afin qu'il soit capable de s'adapter en cours d'exécution [46]. Trois techniques ont été évaluées pour la résolution de la solution optimale d'une équation d'estimation du temps de réponse en fonction du nombre de clients : la méthode de Newton [91], la logique floue [37] et la détection de saturation par des heuristiques. La méthode heuristique envisage d'avoir une charge d'utilisation de 100% sur l'ensemble de ressources : processeurs, mémoire, etc. Pour cela, elle fait simplement varier le nombre des clients [36].

Aux vues des expériences réalisées avec la méthode classique de Newton, celle-ci ne donne pas des temps de réponses stables. La logique floue, elle, fournit des résultats plus robustes, mais par contre le temps nécessaire pour converger vers une solution est long. Enfin, la méthode heuristique exige des informations précises sur les goulots d'étranglement qui, malheureusement, ne sont pas facilement identifiables. Le choix de la technique de prise de décision peut être essentiel au fonctionnement correct d'un système adaptable.

Toutes les études sur les systèmes auto-adaptables présentées jusqu'ici sont spécifiques à un domaine applicatif. Cependant, la définition d'un ensemble de composants génériques permet d'envisager des systèmes auto-adaptables réutilisables. C'est l'objectif du projet Autopilot [99] [94]. Celui-ci propose une plate-forme de contrôle d'exécution adaptable pour les applications parallèles et distribuées. Son architecture suit le modèle générique présenté dans la section précédente, elle est composée d'effecteurs, de capteurs, de modules d'analyse et prise de décision.

En fait, Autopilot est l'agrégation des systèmes auto-adaptables parallèles et distribués qui s'adaptent a ses environnements locaux, en même temps. Ils coopèrent afin d'optimiser l'exécution de l'ensemble des tâches. Chaque tâche contient un capteur, un effecteur

et peut prendre des décisions concernant son exécution sur un nœud en fonctions de l'utilisation mémoire ou processeur. Cependant, le module de prise de décision global, basé sur la technique de la logique floue, intervient à l'exécution de l'ensemble des tâches afin d'améliorer l'intégration entre les nœud.

Pour finir, le dernier exemple de système adaptable que nous présentons est celui de l'ordonnanceur [92] [85]. Cet ordonnanceur traite le problème de contention mémoire sur les multiprocesseurs à mémoire partagée [86]. Le principal but de ce système est de minimiser les échanges de données entre la mémoire principale et le disque. En évitant l'utilisation du disque pour le stockage de données temporaires (*swapping*), l'ordonnanceur augmente en effet les performances de la machine.

Cet ordonnanceur a été conçu suivant deux principes : la détection de la surcharge mémoire qui permet l'identification du problème et l'estimation de la mémoire utilisée des processus par le module de prise de décision. Le premier est possible grâce aux informations sur les pages mémoires allouées qui sont disponibles dans le noyau, et le deuxième par l'interception des requêtes d'allocation mémoire faites par les processus. Ainsi, la détection de la surcharge mémoire est faite à travers l'analyse des informations collectées par le capteur qui est situé au niveau système. Le mécanisme de prise de décision se base sur cette charge et fait ou non la demande d'un nouvel ordonnancement. Enfin, l'effecteur exécute les ordres du module de prise de décision.

La règle appliquée pour le module de prise de décision se base sur une limite inférieure et une limite supérieure pour la charge de la mémoire, et les processus en exécution sont classés selon leurs charges mémoires estimées. Une fois les limites établies et les charges des processus identifiées, il suffit à l'ordonnanceur de faire en sorte que la somme des charges des processus actifs reste dans les limites.

3.3.3 Synthèse

Au cours de cette section, nous avons présenté les définitions, les objectifs et quelques exemples de systèmes adaptables. Le fonctionnement général de ces systèmes est basé sur l'observation des applications ou des ressources. Puis, selon une politique d'utilisation, le système s'adapte ou fait l'adaptation des applications à la politique de contrôle retenue.

Généralement, ces politiques optimisent soit l'utilisation des ressources soit les performances des applications. Dans le premier cas, c'est le système global qui est privilégié (augmentation du rendement, vision administrateur), dans le second ce sont les utilisateurs des applications contrôlées. Même si ces objectifs semblent contradictoires, ils peuvent se rejoindre. Par exemple, si le système adaptable envisage d'optimiser l'utilisation des ressources en évitant la formation des goulots d'étranglement, il maintient un niveau de performance équilibré (pas d'écroulement de performance). Par conséquent, des applications peuvent voir leurs performances augmentées. Malheureusement, il y a des cas où le système d'adaptation est obligé de suspendre des applications provoquant l'allongement

de leur temps d'exécution total.

Le choix des technologies de prise de décision est essentiel pour assurer le bon fonctionnement d'un système. Parfois, des technologies robustes, telle que logique floue, ne sont pas efficaces, comme nous l'avons vu avec le serveur web adaptable. Par contre, des techniques simples, comme celle présentée pour l'ordonnanceur adaptable, peuvent se révéler être bien adaptées. Le choix n'est donc pas trivial.

3.4 Bilan

Dans ce chapitre nous avons présenté les bibliothèques d'accès aux compteurs matériels de performances et les outils d'analyse de performances des applications. L'utilisation des compteurs permet l'observation à grain fin des activités matérielles. Cependant, comme nous avons vu dans le chapitre 2, la configuration et le choix des événements matériels demandent des connaissances techniques spécifiques pour chaque architecture.

Nous avons donc présenté au cours de ce chapitre un ensemble de bibliothèques d'accès aux compteurs matériels de performances qui facilitent les travaux. Toutes ces bibliothèques ont des interfaces d'utilisation des compteurs pour plusieurs architectures processeurs. Des projets ambitieux, tels que PAPI et PCL, proposent des interfaces génériques pour un grand ensemble d'architectures. Pour cela, ils divisent leurs systèmes en deux couches, une responsable de l'accès aux compteurs spécifiques sur chaque architecture, et l'autre, proposant une interface portable. Dans la couche portable, sont définis des événements matérielles commun à la plupart des architectures.

L'apparition des compteurs matériels et des bibliothèques d'accès à ces compteurs permet l'identification des goulots d'étranglements au niveau matériel. Les outils d'analyse de performances, qui ne faisaient jusqu'alors qu'une analyse au niveaux système et applicatif, ont intégré l'utilisation des compteurs des plates-formes pour permettre l'analyse du niveau matériel. Ces outils ont été classés en deux catégories selon la technique d'observation utilisée. Dans la première catégorie, nous avons les outils utilisant la technique de profilage, lesquels font l'observation sur les applications. Dans la seconde, nous avons ceux utilisant la technique par surveillance, lesquels basent l'observation sur les ressources. Le profilage permet la caractérisation des applications et l'identification des goulots d'étranglements. La surveillance permet la construction des systèmes de contrôle adaptables.

Les systèmes adaptables ont comme but l'utilisation optimale des ressources et l'optimisation des performances des applications en cours d'exécution. Il n'est cependant pas toujours possible de satisfaire les deux objectifs en même temps. Les systèmes qui observent de l'extérieur des applications et qui sont basés sur l'utilisation de l'ensemble des ressources sont simplement classés comme systèmes adaptables. Les systèmes, qui ne prennent en compte que leurs besoins afin de s'adapter eux-mêmes aux ressources dispo-

nibles de la machine, sont classés comme systèmes auto-adaptables. En effet, l'amélioration du rendement de la machine ou des performances des applications ne dépend pas du type de système adaptable, mais de la politique de prise de décision utilisée.

Au cours du chapitre 2, nous avons vu que les technologies de la mémoire et du processeur ne suivent pas la même évolution. Puisque les compteurs matériels nous permettent l'observation des activités matérielles à coût faible et que la conception de l'architecture des multiprocesseurs soulève un problème sur l'interconnexion mémoire, nous avons étudié les activités mémoires sur les multiprocesseurs. Les résultats de ce travail d'observation sont décrits dans le prochain chapitre.

4

Observation de l'utilisation mémoire sur les machines multiprocesseurs

Un des goulots d'étranglement bien connu des machines multiprocesseurs à mémoire partagée est celui apparaissant dans la partie supérieure de la hiérarchie mémoire. On parle aussi du problème de la contention mémoire. Généralement, il est occasionné par des applications ayant des faibles localités spatiale et temporelle. Il peut résulter d'une seule application s'exécutant sur plusieurs processeurs simultanément, ou de plusieurs applications fonctionnant en concurrence sur différents processeurs. Les périphériques d'entrées/sorties peuvent aussi jouer un rôle dans ce problème.

La première étape pour analyser et par là traiter un goulot d'étranglement est de l'identifier. Cette identification peut être réalisée par l'observation des activités de la hiérarchie mémoire. Cette observation peut être effectuée par l'intermédiaire des compteurs matériels de performances (section 2.2) et les bibliothèques d'accès associées (section 3.1).

Le problème de contention mémoire peut être traité suivant plusieurs approches. Premièrement par l'augmentation des voies de communication entre la mémoire et les processeurs. Cette approche nécessite généralement de changer le matériel (changement de carte mère ou ajout de bancs mémoire dans le cas d'une architecture par entrelacement d'accès). Ceci peut cependant être très coûteux, voire impossible. La deuxième approche est la restructuration des codes applicatifs, cela nécessite une analyse fine du code afin de le modifier pour obtenir le maximum de performance en sollicitant un minimum les niveaux les plus lents de la hiérarchie mémoire. La troisième approche cherche à équilibrer l'utilisation des ressources en modifiant dynamiquement l'ordonnancement des applications,

afin d'éviter ou de limiter l'apparition de contention mémoire. Pour ce faire, il est nécessaire de connaître le comportement des applications du point vue des niveaux supérieurs de la hiérarchie mémoire. Dans ce chapitre, nous présentons des évaluations montrant les effets de la contention mémoire, ainsi que le moyen de l'identifier en la liant à l'observation de l'activité du bus mémoire. Puis nous présentons l'activité mémoire engendrée par un certain nombre d'applications durant leur exécution. Finalement nous terminons par une série de tests évaluant l'impact des activités d'entrées/sorties (réseaux et accès disques) comme tâches de fond sur la hiérarchie mémoire.

4.1 Introduction aux évaluations

Les premiers résultats d'évaluations présentés dans ce chapitre (4.3) sont des résultats de performances concernant la capacité de machines multiprocesseurs et notamment les limites imposés par leur hiérarchie mémoire. L'accent est mis sur les différents niveaux d'accélérations atteints. Ce paramètre permet de considéré le rendement obtenu sur les machines.

L'objectif étant d'obtenir les moyens d'un contrôle d'exécution, il est nécessaire de pouvoir évaluer le rendement en cours d'exécutions. Pour cela, nous essayons de déterminer une relation liant une activité matérielle via les compteurs matériels et le niveau d'accélération obtenu. Le coût négligeable des mesures par compteurs matériels rend possible l'observation en cours d'exécution et par là l'estimation du rendement nécessaire au système de contrôle.

Un autre élément important que nous étudions dans ce chapitre est la constance du comportement des applications du point de vue de leur utilisation de la hiérarchie mémoire en terme de débit. En effet une condition de second ordre pour la faisabilité d'un système de contrôle est de disposer d'applications ayant une certaine régularité de comportement. Si tel n'est pas le cas, les décisions de contrôle risquent d'être remises en cause trop fréquemment et rendraient le système au mieux inopérant et au pire contre-productif.

Avant de présenter les différentes évaluations, nous décrivons les différents programmes de tests employés, ainsi que les différentes machines.

4.2 Le choix des programmes de tests

Nous avons considéré plusieurs catégories de programmes de tests pour nos évaluations : les programmes synthétiques, principalement de STREAM ([81]), les programmes séquentiels de SPEC2000 [102], les programmes parallèles numériques de NPB NAS [44] [43] et des programmes de tests de disque et de réseau.

Parmi ces programmes, exception faite de ceux pour les tests disques et réseau, nous avons cherché à étudier l'impact des programmes couvrant le spectre des deux grandes familles que sont les programmes de type *CPU Bound* et *Memory Bound*. Comme leur nom l'indique, les programmes *CPU Bound* sollicitent principalement les unités fonctionnelles arithmétiques et/ou flottantes des processeurs, et moins la hiérarchie mémoire. Les programmes *Memory Bound* quant à eux, imposent un stress important sur l'ensemble ou sur un sous-ensembles des niveaux mémoires.

Les programmes synthétiques vont permettre, d'une part de tester les limites des performances de la hiérarchie mémoire des machines, et d'autre part d'étudier finement la relation entre l'accélération constatée et le débit des événements observés. Il y a 5 programmes dans cette catégories dont 4 sont issus de STREAM [81]. Ils sont tous construits sur le même modèle qui consiste en une boucle parcourant linéairement un ou plusieurs tableaux et effectuant une opération entre les éléments de ces derniers. Les élément des tableaux sont des flottants sur 64bit (*double*). La taille des tableaux est paramétrables. Les opérations sont les suivantes (a,b,c correspondent aux tableaux et i est la variable d'itération) :

- **copy** : $a[i] = b[i]$,
- **scale** : $a[i] = q * b[i]$,
- **sum** : $a[i] = b[i] + c[i]$,
- **triad** : $a[i] = b[i] + q * c[i]$,
- **indirect** : $a[i] = b[d[i]]$, (d[i] est un tableau de valeurs entières tirées aléatoirement entre 0 et $taille_{tableau} - 1$, cette opération ne fait pas partie des STREAM)

Ces opérations se retrouvent souvent dans les noyaux de calculs, et ils occasionnent une forte charge sur la hiérarchie mémoire. De plus, leur simplicité et le fait que chaque itération est indépendante rendent triviale la parallélisation en mémoire partagée avec des directives OpenMP.

L'ensemble des programmes séquentiels retenus dans SPEC2000 [102], est très souvent utilisé pour évaluer les machines et les processeurs. Ces programmes sont jugés représentatifs des applications pouvant être exécutées sur des stations de travail ou sur des serveurs. Ils ont été développés sur la base d'algorithmes d'applications réelles et classés en deux groupes de tests. Le premier groupe, CINT2000, est composé de 12 applications qui exécutent majoritairement des opérations sur des données entières. Le deuxième groupe, CFP2000, comprend 14 applications dont les noyaux traitent des opérations sur des données flottantes. Parmi ces 26 programmes, nous trouvons, par exemple, des applications comme ART2 (*The Adaptive Resonance Theory 2*) et EON (*Ray tracer*). L'application ART2 [40] est un programme de reconnaissance de formes utilisant un algorithme de réseau de neurones. EON [66] est un programme de lancer de rayons probabiliste pour la construction d'image en trois dimensions. Le premier programme est catalogué comme étant de type $MemoryBound$ et le deuxième est un exemple de la catégorie $CPUBound$.

L'ensemble suivant de programmes de tests choisis est celui de l'ensemble NAS ver-

sion 3 Benchmark [44] [43]. Les programmes de tests NAS ont été développés au sein d'un laboratoire de la NASA et sont composés de 8 noyaux de calcul dont 5 sont dérivés des applications de calcul en dynamique des fluides et les 3 autres de pseudo-applications. Parmi les programmes NAS, nous trouvons des applications qui requièrent une utilisation importante de la hiérarchie mémoire ($MemoryBound$), par exemple CG (*Conjugate Gradient Method*), et avec une forte charges processeurs ($CPUBound$) comme EP (*Embarrassingly Parallel benchmark*). Enfin, l'utilisation des algorithmes NAS est très courante dans leur version parallélisée suivant le paradigme par passage de message en utilisant le standard MPI [112]. Pour nos études, nous avons retenu la version parallélisés en mémoire partagée avec le standard OpenMP [64]. Ces programmes présentent une grand diversité de comportement et sont représentatifs des applications réelles du domaine du calcul numérique.

Pour finir sur l'ensemble des programmes de tests, nous avons sélectionnés des programmes d'évaluation de performance de disque de stockage, Bonnie [21], et de performance réseau, Netperf [60]. Notre objectif est d'observer via ces programmes, l'influence de l'utilisation de ces deux ressources sur la hiérarchie mémoire en concurrence avec des applications de calcul. Le programme de test Bonnie dispose de trois tests : l'écriture séquentielle de données (*putc*), la lecture séquentielle de données (*getc*) et la lecture aléatoire des données dans un fichier par plusieurs processus en concurrence. Le programme de test Netperf est très simple, il effectue des transferts de données entre sockets via les protocoles IP TCP ou UDP.

4.3 Conditions d'expérimentation

Avant d'aborder les différentes évaluations, nous présentons les machines utilisées et détaillons quelques informations liées aux conditions d'expérimentations.

	proc. number	proc. freq. (MHz)	taille des adresses	Distribution Linux / Version du noyaux	Bibliothèque d'accès aux compteurs matériels
Pentium Pro	4	200	32 bits	Mandrake / 2.4.13	perfctr 2.3.xx
Pentium II	2	450	32 bits	Debian / 2.4.20	perfctr 2.3.xx / 2.6.xx
Pentium III	4	550	32 bits	RedHat / 2.4.18	perfctr 2.3.xx
Pentium 4	2	1700	32 bits	Mandrake / 2.4.18	perfctr 2.3.xx / 2.6.xx
Intel Itanium2	2	900	64 bits	RedHat / 2.4.18	perfmon
AMD Athlon	2	1200	32 bits	Debian / 2.4.22	perfctr 2.6.xx
AMD Opteron	2	1400	64 bits	Fedora / 2.4.20	non installé

TAB. 4.1 *Liste des machines utilisées pour les différents tests avec leurs caractéristiques*

Le tableau 4.1 regroupe les machines utilisées pour les différents tests. Parmi ces machines il y a 5 types d'architectures différentes. Les trois premières machines font partie

de la famille Intel *Pentium 6* (*Pentium Pro*, *Pentium II* et *Pentium III*) parmi laquelle nous avons un biprocesseur et deux quadriprocesseurs. La machine suivante est un biprocesseur, doté de l'architecture processeur *Netburst* [4], avec des processeurs *Pentium 4*. Ensuite nous avons une architecture $64bits$ avec une machine biprocesseur dotés de processeurs Intel Itanium 2. Finalement, nous avons 2 machines biprocesseurs avec des processeurs AMD d'architectures différentes, une à base de processeurs *Athlon* $32bits$ et l'autre avec des processeurs AMD *Opteron* $64bits$.

Dans le tableau 4.1 il est aussi reporté les différentes distributions Linux présentes sur chaque machine ainsi que la bibliothèque d'accès aux compteurs utilisées. Plusieurs compilateurs ont été utilisés. Dont celui du groupe *The Portland Group (PGI)* version 3.2 avec l'option d'optimisation *-fast* pour les architectures Intel *Pentium 6* et *Pentium 4*. Le compilateur Intel version 8.0 avec l'option d'optimisation *-O3* pour la machine Itanium 2. Et finalement le compilateur GNU version 3.3 avec l'option *-O3* pour la machine *Opteron*.

Tous les tests n'ont pas été reproduits sur l'ensemble des machines et cela pour plusieurs raisons. En effet nous ne possédions pas directement toutes ces machines ce qui a ralenti les tests utilisant les bibliothèques d'accès aux compteurs (problème de mise en place et de stabilité). Ainsi pour la machine quadriprocesseur *Pentium III* nous n'avons eu qu'un accès très limité dans le temps pour réaliser nos expérimentations. Pour la machine *Opteron*, les tests sont limités car le compilateur GNU ne supporte pas les directives de parallélisation OpenMP. Pour conclure sur ce point, les résultats sur les machines où n'avons pas pu faire l'ensemble des tests sont tout de même présentés afin d'étayer et de compléter le reste des résultats.

Les résultats présentés par la suite et sauf mention spéciale, ont été obtenus en réalisant une moyenne sur au moins 30 essais. Les écart-types sont faibles et ne sont pas présentés, exception faite des résultats de la section 4.3.2.

4.3.1 Analyse de performances des multiprocesseurs

De nombreuses études sur les machines multiprocesseurs indiquent que la saturation de la hiérarchie mémoire limite fortement les performances [65] [25]. Dans cette section, nous effectuons des tests généraux qui corroborent ce résultat sur les machines utilisées.

La première évaluation présentée porte sur l'ensemble de programmes STREAM. Ces programmes ont été parallélisées avec OpenMP sur la seule boucle qu'ils contiennent. Il n'y a pas de dépendance entre les boucles. Le seul paramètre de ces programmes est la taille des tableaux de calcul qui a été fixée aux alentours d'environ 15% de la totalité de la mémoire vive de chaque machine. Cette limite correspond à la taille minimale au-delà de laquelle les résultats n'évoluent plus à cause des différents caches. Avec ce paramétrage les programmes stressent énormément la hiérarchie mémoire. Les machines disponibles pour ces tests ont été le quadriprocesseur *Pentium Pro* et les biprocesseurs *Pentium II*,

Pentium 4 ainsi que la machine *Athlon*. Le compilateur utilisé sur ces tests est PGI. Le tableau 4.2 présente les accélérations obtenues avec les différents processeurs. On peut observer que les valeurs atteintes (de 1.01 à 1.3) sont très loin des valeurs théoriques maximales (ici 2). Cela indique clairement une contention au niveau supérieur de la hiérarchie mémoire. Cela suggère aussi que sur des applications très stressantes les performances peuvent, dans le pire des cas, atteindre ces limites. Enfin, ces résultats montrent aussi que sur l'ensemble des machines biprocesseurs de générations et d'architectures différentes, la hiérarchie mémoire restent un facteur potentiellement très limitant.

	Pentium Pro	Pentium II	Pentium 4	Athlon ($32bits$)
STREAM Copy	1.06	1.17	1.18	1.17
STREAM Triad	1.02	1.24	1.30	1.15
STREAM Add	1.01	1.22	1.25	1.29
STREAM Scale	1.04	1.29	1.22	1.23

TAB. 4.2 *Dans ce tableau nous montrons les résultats des programmes de tests STREAM sur 4 machines de génération et d'architecture différentes. Les accélérations obtenues sont loin du maximum théorique (ici 2).*

La deuxième série d'évaluations a été réalisée avec les programmes NAS dans leur version parallèle OpenMP. Cette évaluation est intéressante puisque, pour les applications parallèles en mémoire partagées, plusieurs facteurs peuvent jouer sur l'accélération finale : en premier lieu, le rapport entre la partie parallèle et la partie séquentielle, ensuite la qualité de la parallélisation et les problèmes afférents comme le nombre des synchronisations, les prises de verrous ou la présence de faux partage de données. Finalement, se pose ensuite le problème de la qualité de la hiérarchie mémoire. Dans le cas des programmes NAS, le comportement global des applications est bien connu. Elles ont un rapport liant la partie parallèle et la partie séquentielle très important lorsque qu'il y a peu de processeurs, ce qui est notre cas. Nous avons aussi vérifié qu'il n'y a pas problèmes de faux partage notable en utilisant les compteurs matériels (cf. section 4.3.2).

Dans l'ensemble NAS il y a un paramètre, appelé classe, qui fixe la taille des données utilisées pour chaque application. Ce paramètre peut prendre trois valeurs qui sont, par ordre croissant de taille, A, B et C. Dans les tests réalisés, ce paramètre peut évoluer pour une même application suivant la capacité mémoire des machines. Ainsi les machines *Pentium 4* et *Itanium 2* ont exécuté les programmes NAS de la classe "B", exception faite pour l'algorithme FT sur le *Pentium 4*, qui a été réalisé avec la classe "A". La classe "A" a été utilisée pour la machine *Pentium II* comme paramètre à tous les programmes.

Le tableau 4.3 rassemble les accélérations obtenues. Les résultats montrent qu'avec l'application EP la valeur de l'accélération atteint le maximum pour toutes les machines. Cette application est connue comme faisant partie des applications *CPU Bound*. Les résultats sur les autres applications sont plus variés suivant les machines. La machine *Pentium II* se comporte raisonnablement sur l'ensemble des applications. Il est important de

	Pentium II	Pentium 4	Itanium 2 ($64bits$)	Opteron ($64bits$)
NAS CG	1.8	1.0	–	1.5
NAS BT	1.8	1.7	1.5	1.9
NAS EP	2	2	2	2
NAS FT	1.7	1.1	1.5	–
NAS MG	1.6	1.5	1.5	1.7
NAS SP	1.7	1.5	1.5	1.8
NAS LU	–	–	1.5	1.8

TAB. 4.3 *Accélérations obtenues avec les programmes NAS sur* 3 *machines multiprocesseurs différentes. La machine* Pentium II *utilise la classe de paramètres "A" (moins exigeant en taille mémoire) et les trois autres utilisent la classe "B" sauf pour l'algorithme FT pour la machine* Pentium 4 *(classe "A").*

rappeler que pour cette machine la classe utilisée est d'une taille inférieure par rapport aux autres machines, par conséquent la charge sur la hiérarchie mémoire est globalement plus faible. Par contre, la machine *Pentium 4* possède des performances plutôt faible voire très faible. Elle est clairement limitée par sa hiérarchie mémoire. Pour le reste des valeurs, il est a priori difficile d'affirmer quel est le facteur prépondérant qui limite l'accélération. Néanmoins, en analysant les codes applications et en nous référant à certaines d'études [69], [95], nous pouvons extraire les informations suivantes : l'application CG dispose d'une boucle représentant la majorité du code exécuté, et cette partie est particulièrement stressante du point de vue de la hiérarchie mémoire. Pour les applications BT, FT, MG, SP et LU, leur structure est plus complexe et il est difficile d'identifier le facteur prépondérant.

Dans le tableau 4.4 nous présentons les résultats de l'ensemble des applications SPEC2000 lancées en concurrence. Ces applications n'ont pas des données partagées puisqu'elles sont séquentielles, et donc, nous observons des performances proches de l'optimum (ici 2). Pourtant, dans les deux cas, nous avons aussi des performances lointaines des optimums. Pour le *Pentium II*, les applications SPEC2000 SWIM, ART et MCF ont des performances proches à 1.50, et pour le *Pentium 4*, quelques-unes sont même à 1, SPEC2000 ART par exemple. Les comportements confirment alors que la hiérarchie mémoire est également un facteur limitant les performances pour les applications séquentielles qui s'exécutent en concurrence.

4.3.2 Estimation de l'accélération en fonction de l'activité mémoire

Il y a plusieurs conditions nécessaires pour réaliser un mécanisme de contrôle de rendement basé sur l'observation de la hiérarchie mémoire. Deux conditions voisines sont : la possibilité d'identifier la présence d'un goulot d'étranglement sur la hiérarchie mémoire et

	Pentium II	Pentium 4
Wupwise	1.87	1.94
Swim	1.46	1.28
Mgrid	1.73	1.77
Applu	1.82	1.84
Mesa	1.99	1.98
Art	1.54	1.04
Equake	1.67	1.54
Ammp	1.95	1.75
Apsi	1.88	1.85
Gzip	1.96	1.90
Vpr	1.91	-
Gcc	1.83	-
Mcf	1.55	1.33
Crafty	1.99	1.97
Parser	1.94	1.87
Eon	1.99	1.99
Gap	1.82	1.89
Vortex	1.97	1.84
Bzip2	1.86	1.83
Twolf	1.88	1.70

TAB. 4.4 *Dans ce tableau nous montrons les résultats des programmes de test SPEC2000 sur* 2 *machines multiprocesseurs différentes. Ces tests présentent les résultats des exécutions de ces applications en concurrences.*

la capacité de mesurer son impact sur le rendement global. La première condition signale que la machine est dans une zone de saturation à faible rendement, la seconde permet de quantifier cette baisse. Dit autrement, il s'agit de déterminer une relation entre l'activité mémoire et le niveau de performance atteint par les applications.

Dans l'étude [25], les auteurs ont montrés que les performances obtenues sur des régions parallèles des applications NAS sont liées aux débits d'événements mémoires observés sur le bus système. La parallélisation était réalisée par le biais de directives OpenMP.

Nous reprenons ici le même principe pour étudier la relation entre l'activité mémoire et le rendement de la machine. Nous allons considérer que le rendement peut être caractérisé par l'accélération. Une accélération maximale signifie en effet que le rendement est aussi à son niveau maximum. L'activité mémoire est observée à travers les compteurs matériels. Nous considérons plusieurs architectures de types et de générations différents, il s'agit pour chaque cas de déterminer l'existence d'un relation entre l'accélération (le rendement) et l'activité mémoire (débit d'événement). Pour cela il faut identifier les évé-

nements matériels qui rendent compte au mieux de l'activité mémoire. Nous nous sommes donc concentrés sur l'activité autour du bus mémoire, le niveau supérieur de la hiérarchie où siègent d'éventuels goulots d'étranglements.

La méthodologie utilisée pour établir la relation entre l'accélération et l'activité mémoire est la suivante. Tout d'abord nous avons utilisé les programmes de tests STREAM parallélisés avec OpenMP. Nous avons montré dans la section précédente qu'ils portent à saturation la hiérarchie mémoire et font s'écrouler l'accélération. Pour étudier ce phénomène, il est nécessaire de maîtriser le niveau d'activité mémoire. Pour cela la méthode la plus simple est d'insérer des instructions *nop* dans le corps des boucles. Plus le nombre d'instruction *nop* est élevé, plus la charge sur la hiérarchie sera faible, ce qui par conséquent fait décroître l'activité mémoire. Ainsi, nous obtenons une accélération en fonction de l'activité mémoire qui est représentée par le débit de l'événement matériel sélectionné. Le débit est égal à $\frac{n_{evenements}}{t_{parallele}}$ où $n_{evenements}$ est le nombre de événements relevé sur le compteur et $t_{parallele}$ le temps mesuré lors de l'exécution en parallèle. Le calcul de l'accélération équivaut naturellement à $\frac{temps_{sequentiel}}{temps_{parallele}}$. Trouver l'événement qui représente le mieux l'activité mémoire n'est pas une chose évidente. Dans la mesure du possible nous avons adopté une approche exhaustive en ayant au préalable sélectionner les événements relatifs à la hiérarchie mémoire.

Nous allons définir trois zones de comportements des applications à l'aide de la figure 4.1 qui présente le type de résultat attendu. Sur cette figure nous avons représenté les zones types où doit évoluer la relation entre l'activité mémoire et l'accélération. Nous définissons grossièrement trois zones comportementales. La première nommée *zone non-saturée* correspond à la zone où le rendement est à son maximum et où l'activité mémoire est faible. La deuxième appelée *zone intermédiaire* est un zone de transition où le rendement chute ou est très instable avec un niveau d'activité mémoire qui est supérieur à la zone précédent. Finalement la troisième, la *zone saturée*, correspond à une accélération faible est donc un rendement minimal pour une activité mémoire élevée.

Nous avons également effectué des mesures avec les programmes NAS version OpenMP, ainsi que SPEC2000. Pour ces derniers, il n'y a pas de version parallèle, dès lors nous avons choisi une exécution en concurrence du même programme.

Lors de tests préliminaires, nous avons observé une grande variabilité des valeurs mesurées à la fois sur les durées et sur les valeurs des compteurs. De plus cette variabilité ne s'estompe pas en multipliant les mesures pour obtenir une valeur moyenne ou en employant la médiane. Les figures qui vont être présentées par la suite contiennent seulement les moyennes sur 30 mesures effectuées par point. Généralement les moyennes et les médianes se confondent. Les figures avec écarts types sont présentées en annexe A. Nous utilisons les événements sur lesquels la montée du débit provoque la décroissance des performances.

Enfin, le temps d'une prise de mesure du nombre d'événements matériels et la prise de date sont négligeables par rapport au temps d'exécution ($7\mu s$ par prise de mesure sur

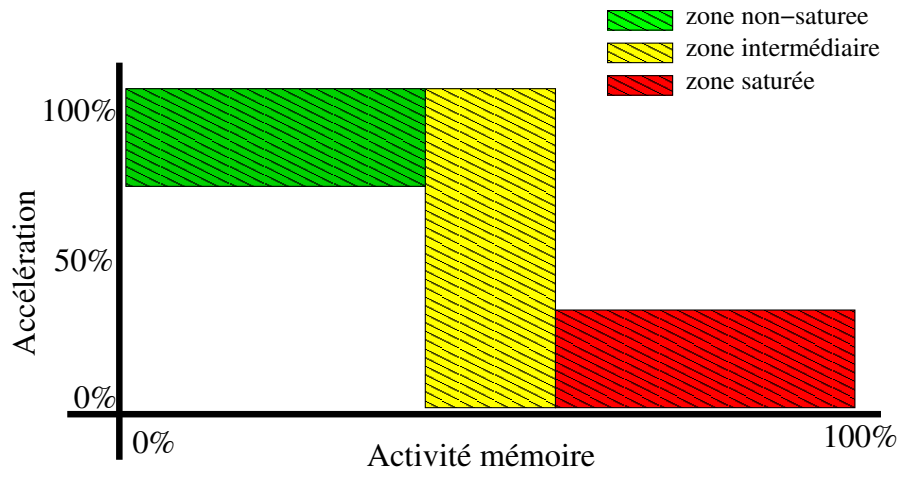

Figure 4.1 *Les applications sont classées en trois zones :* non-saturée, intermédiaire *et* saturée. *La définition de ces trois zones est basée sur l'accélération observée par les applications et le niveau d'activité mémoire (mesurée par les compteurs matériels)*

la machine *Pentium 4*).

4.3.2.1 Architecture processeur de la famille Intel Pentium 6

Nous commençons nos tests sur la relation entre l'activité mémoire et l'accélération avec la famille Intel *Pentium 6*, pour laquelle nous avons le plus grand nombre de machines. Les trois machines de cette famille que nous avons testées sont : deux machines quadriprocesseurs *Pentium Pro* et *Pentium III* et une machine biprocesseur *Pentium II* (voir le tableau 4.1).

Nous rappelons que l'élément de la hiérarchie mémoire que nous envisageons d'observer est le bus mémoire. Cela étant, la première étape de nos expérimentations est le choix de(s) l'événement(s) matériel(s) capable(s) de représenter le niveau d'activité de cette ressource.

Ces événements ainsi que les compteurs matériels de performances ont été présentés dans la section 2.2. Le nombre d'événements identifiés sur la hiérarchie mémoire est d'environ 30 sur cette famille de processeur. Nous avons utilisé la bibliothèques PAPI (voir section 3.1) pour accéder aux compteurs, cela du fait de sa simplicité d'utilisation et de la disponibilité d'événements prédéfinis.

Parmi les événements prédéfinis par l'API de PAPI, nous avons essayé tous ceux de la hiérarchie mémoire (15 événements) et de la cohérence de cache (6 événements). Malheureusement, aucun de ces événements ne permet d'établir une relation directe entre le débit d'événement et l'accélération. Ce résultat était prévisible puisque la majorité de ces événements est liée à des mouvements de données qui se déroulent exclusivement à

l'intérieur des processeurs.

Nous avons ensuite évalué l'ensemble des événements liés au bus mémoire de l'architecture étudiée. Cet ensemble est composé du groupe *EBL - External Bus Logic* [7], lequel contient environ 20 événements. Nous avons fait des tests exhaustifs sur l'ensemble des événements. L'accès à ces événements ce fait à travers de la bibliothèque *Perfctr* [90].

Les premiers tests avec les boucles sur des tableaux (STREAM et indirect) nous ont montré que l'événement **BUS_TRAN_MEM** représente bien le rapport entre l'accélération et l'activité mémoire. Cet événement semble être le plus adapté à nos besoins puisqu'il compte toutes les transactions sur le bus mémoire.

Néanmoins, il ne permet pas d'identifier l'origine (le processeur) de ces transactions mémoires. Savoir de quel processeur provient le plus grand nombre de transactions est essentiel pour l'observation fine de l'utilisation mémoire de la machine et la réalisation d'un système contrôle.

Nous avons élargi nos expérimentations sur les activités des caches. Finalement, nous avons trouvé l'événement **L2_DBUS_BUSY** qui compte le nombre de cycles où le bus de données du cache L2 est occupé, lequel nous permet d'identifier et de quantifier le flot de transaction par processeur.

Les résultats obtenus sur les différentes machines d'architecture *Pentium 6*, présentés dans la suite de cette section, sont obtenus avec cet événement.

Pentium Pro : Les tests obtenus sur cette machine quadriprocesseurs de génération ancienne sont présentés à titre d'exemples. De plus, la capacité mémoire de cette machine est très limitée (256 Mo), ce qui diminue d'autant le nombre d'expérimentations envisageables. Nous nous sommes donc restreint au lancement des programmes de tests STREAM et indirect.

Les courbes de la figure 4.2 montrent qu'au-delà d'un certain niveau d'utilisation de la hiérarchie mémoire, on observe une chute brutale des accélérations. Les programmes de tests modifiés par l'insertion de *nop* montrent bien l'évolution de l'accélération en fonction de l'utilisation du bus mémoire. Le comportement des programmes peut être décomposés en deux zones. Une première, la plus large sur le graphique, correspond à une zone *non-saturée* où l'accélération est proche du maximum atteignable. La seconde correspond à une zone *intermédiaire* et à un zone *saturée* quasiment indistinguable. La difficulté ici est qu'il est impossible de connaître le niveau précis de l'accélération en observant le débit d'événements. La chute rapide de l'accélération montre que cette machine est très sensible au problème du goulot d'étranglement.

Pentium II : Dans les expériences sur l'architecture *Pentium II*, nous avons élargi les tests afin de caractériser l'utilisation du bus mémoire sur un plus grand ensemble de pro-

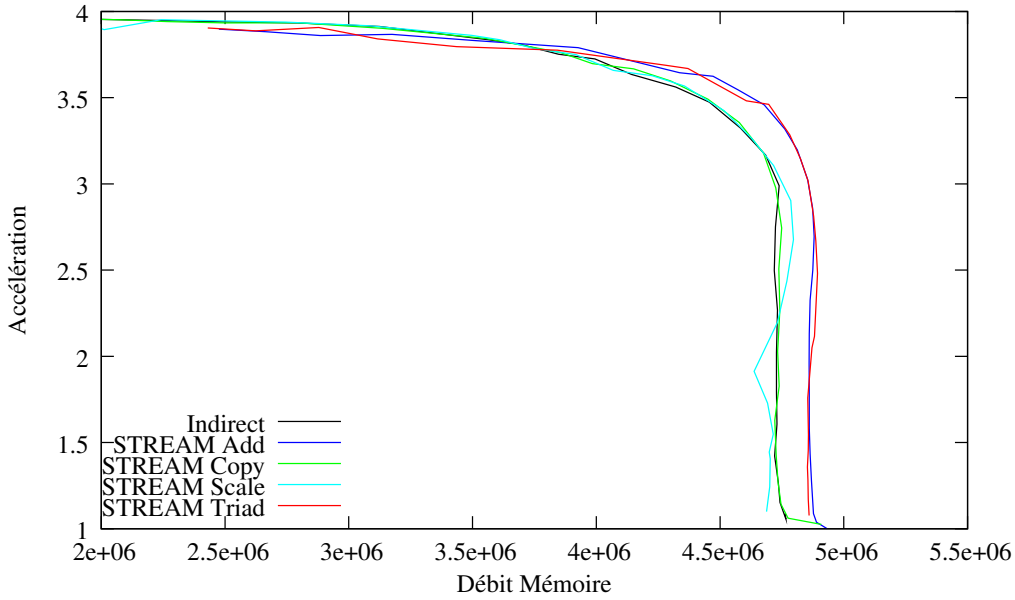

Figure 4.2 ***Quadriprocesseur Intel Pentium Pro :*** *Les résultats sur les programmes de tests STREAM et indirect montrent l'évolution de l'accélération en fonction de l'utilisation mémoire pour un quadriprocesseur* Pentium Pro. *Ces programmes de tests présentent des comportements caractéristiques des deux zones : une* non-saturée *et une* intermédiaire.

grammes.

Nous avons réalisé les tests avec les programmes STREAM (Triad, Copy, Add et Scale) et indirect modifiés pour constater l'évolution de l'accélération en fonction de l'utilisation du bus mémoire. De plus, nous avons ajouté les programmes de tests NAS et SPEC2000.

Les algorithmes NAS utilisés sont EP, BT, MG, SP, CG et FT dans leurs versions parallélisées avec OpenMP. Chacun de ces algorithmes est un point dans la courbe (figure 4.3). Parmi l'ensemble SPEC2000, nous avons évalué les programmes : *wupwise*, *swim*, *mgrid*, *applu*, *mesa*, *art*, *equake*, *ammp* et *apsi* issus du sous-ensemble CINT2000, ainsi que *gzip*, *vpr*, *gcc*, *mcf*, *crafty*, *parser*, *gap*, *vortex*, *bzip2* et *twolf* issus de CFP2000. Pour chaque expérience, deux programmes identiques sont exécutés en concurrence (machine biprocesseurs). Les résultats des programmes SPEC2000 sont représentés par une croix sur le graphique et ils sont indifférenciés dans la légende par soucis de lisibilité.

Pour les programmes NAS et SPEC2000, les mesures sont réalisées au début et à la fin de chaque programme. Ces mesures prennent donc en compte les parties des programmes qui n'ont pas été parallélisées. Nous restons ici dans une approche à granularité élavée. Dans la section 4.3.3, nous montrons l'évolution de l'utilisation mémoire en fonction du temps des programmes NAS et SPEC2000.

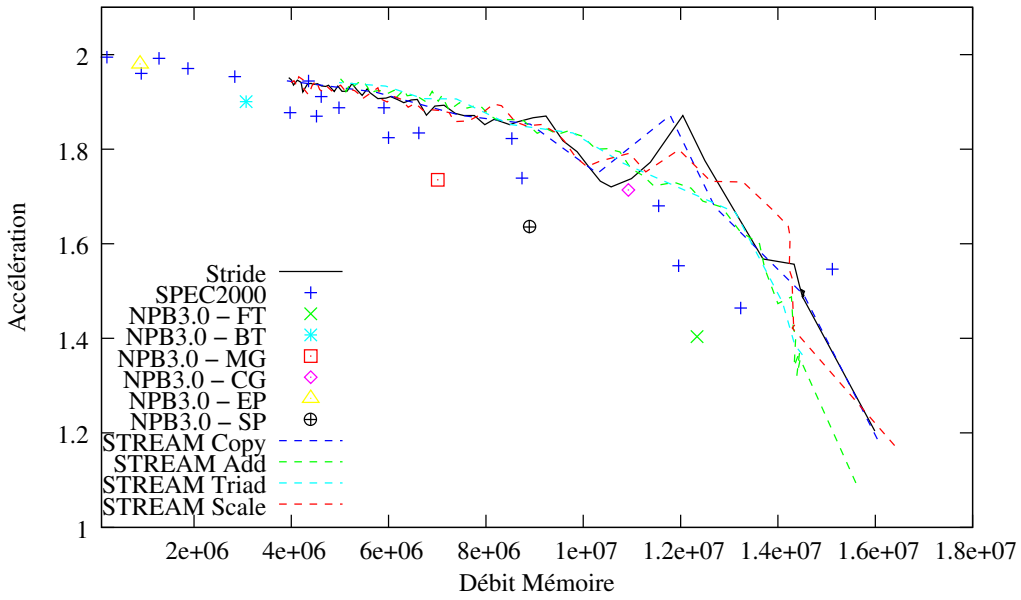

Figure 4.3 ***Biprocesseur Intel Pentium II :*** *Les programmes NAS sont représentés par des points et n'ont pas été modifiés. Pour les autres programmes, nous avons ajouté des instructions assembler (*nop*) dans les boucles principales pour soulager l'utilisation mémoire. Nous observons que les programmes qui ont des accélérations proches de l'optimum, ont un faible débit mémoire.*

Les courbes et les différents points de la figure 4.3 présentent un large spectre de comportement. Nous avons des applications réelles, telles que celles des SPEC2000, avec des accélérations proches du maximum et des débits très faibles, ainsi que d'autres avec de mauvaises accélérations et des débits très élevés. Le comportement de l'ensemble des applications peut être classé suivant les 3 zones, définies précédemment : *non-saturée*, *intermédiaire* et *saturée*. Nous observons que les accélérations chutent moins brutalement que dans le cas précédent. La détermination de l'accélération suivant le niveau de débit d'événement en est facilitée.

Pentium III : La dernière machine de la famille *Pentium 6* testée a été un autre quadri-processeur, cette fois composé par 4 processeurs de l'architecture *Pentium III*. Les caractéristiques physiques permettent l'exécution de l'ensemble des tests STREAM, indirect, NAS et SPEC2000. Malheureusement, notre accès à cette machine pour réaliser les tests a été très limité.

Nous n'avons réalisé que 2 campagnes de tests : STREAM Copy et indirect. L'objectif de ces tests était uniquement de faire une validation minimale de la relation entre les accélérations et l'utilisation mémoire sur une cette architecture.

Les courbes de la figure 4.4 montrent le comportement des trois programmes de tests. On peut distinguer les trois zones prédéfinies : *non-saturée*, *intermédiaire* et *saturée*. Ces

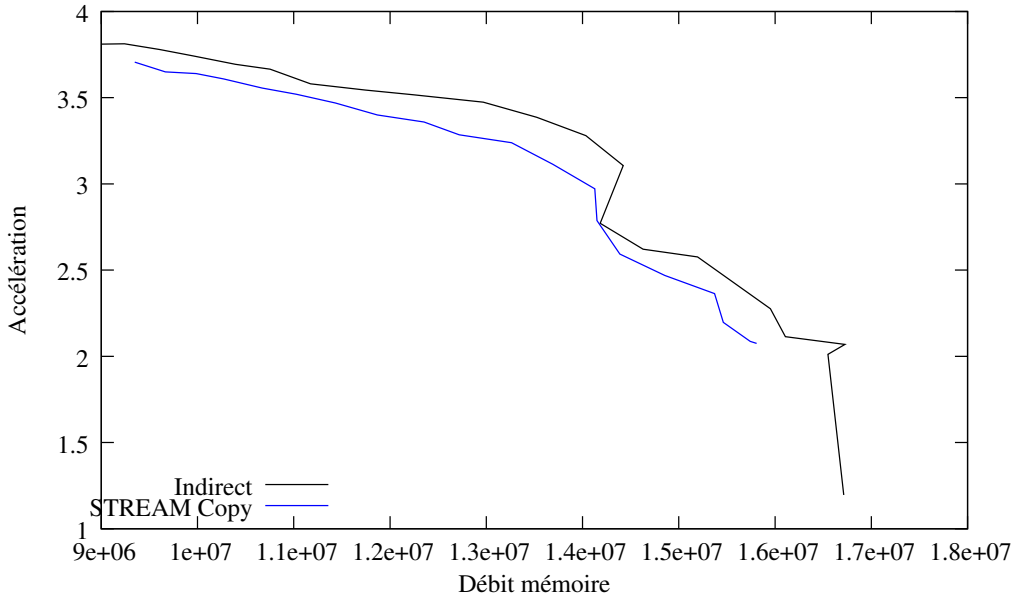

Figure 4.4 ***Quadriprocesseur Intel Pentium III :*** *Les résultats préliminaires présentés sur la machine quadriprocesseur* Pentium III *montrent aussi le rapport entre la chute de l'accélération et le niveau d'utilisation du bus mémoire. L'évolution de l'accélération en fonction de l'utilisation du bus se révèle plus douce que celle sur les deux autres machines testées.*

résultats montrent que l'évolution de l'accélération est relativement douce et similaire au cas précédent. La contention sur le bus mémoire est encore évidente et la relation entre l'accélération et l'utilisation du bus mémoire est confirmée.

4.3.2.2 Architecture processeur de la famille Intel Pentium 4

Dans cette section, nous présentons les expériences menées sur l'architecture Intel *Pentium 4*. La machine que nous avons utilisée est la machine *Pentium 4* présentée dans le tableau 4.1.

L'architecture de processeur de type *Netburst* a un nombre d'événements et de compteurs matériels de performances plus élevé que les architectures étudiées dans la section précédente (voir section 2.2). Avec l'augmentation du nombre de compteurs les formats des registres de configuration des compteurs matériels ont été modifiés. Le nouveau format des registres rend les manipulations plus complexes. De ce fait, le portage de la bibliothèque d'accès aux compteurs matériels *Perfctr* a été lent et les versions disponibles ont souvent été instables. De même, l'évolution des noyaux Linux a largement entravé la stabilisation de cette bibliothèque.

Le grand nombre d'événements et de sous-événements de cette famille rend la mé-

thodologie par les tests exhaustifs employée sur la famille *Pentium 6* difficilement envisageable. Pour cette famille, nous avons donc approfondi l'étude sur les événements matériels ainsi que sur le fonctionnement de l'architecture afin de réduire le nombre de tests à réaliser. Dans notre premier tri, nous n'avons gardé que les événements associés aux unités fonctionnelles liées à l'ensemble de la hiérarchie mémoire. Malgré tout, la quantité d'événements à tester restent encore élevée (environ 100). Nous avons donc suivi un classement des ensembles d'événements par groupe selon les unités fonctionnelles (plus de détails sur la section 2.2.2), pour structurer notre campagne de tests.

Après un test ciblé d'événements par groupe, nous avons détaillé les tests sur quelques groupes. Les groupes retenus sont ceux qui ont montré des résultats stables et une évolution de l'accélération en fonction du débit d'événements observés. De cette façon, le nombre d'événements a été réduit d'environ 80%. Enfin, nous nous sommes concentrés sur les événements du groupe *Front Side Bus*. Parmi les événements de ce groupe, celui qui a présenté le comportement le plus stable sur les tests préliminaires a été l'événement *Data Ready*. Cet événement compte les cycles sur lesquels il y a des transfers sur le bus provoqués par un processeur ou par le *chipset* .

Pentium 4 : Le biprocesseur *Pentium 4* utilisé est plus récent que les trois autres machines. La taille de la mémoire principale et la vitesse des processeurs sont largement supérieures à celles des machines utilisées dans la section précédente. Les programmes utilisés lors des tests étaient l'ensemble des programmes STREAM, indirect, NAS et SPEC2000.

Les résultats présentés sur la courbe de la figure 4.5 montre un comportement stable sur la grande majorité des programmes testés.

Les programmes de tests STREAM et indirect sont représentés par des lignes. Les autres programmes sont indiqués par des points. De nouveau les résultats de chaque programme SPEC2000 sont indifférenciés.

La courbe (figure 4.5) présente des programmes dont les résultats en terme d'accélération sont proche du maximum, tels que NAS EP et trois programmes SPEC2000 (*crafty*, *mesa* et *eon*). Ils ont pour caractéristique commune d'être du *CPU Bound*. Part ailleurs certains programmes SPEC2000 ont des performance en dessus de 1.4.

Parmi tous nos programmes de tests, seul l'algorithme NAS CG présente des résultats inattendus. Jusqu'à maintenant nous n'avons pas réussi à expliquer ce comportement. Un certain nombre d'investigations ont été menées mais n'ont pas apportées de réponse. Nous avons notamment isolé et examiné la boucle parallèle qui représente la majeure partie du temps d'exécution, mais le résultat est sensiblement le même. Nous avons vérifié l'impact des verrous en les inhibant et en comparant cette version à celle d'origine, le débit mémoire s'en trouve réduit et l'accélération augmentée mais là aussi de manière peu significative. Nous avons aussi vérifié l'absence de trafic de cohérence de cache lié à un éventuel problème de faux partage. Il apparaît donc que pour un certain schéma

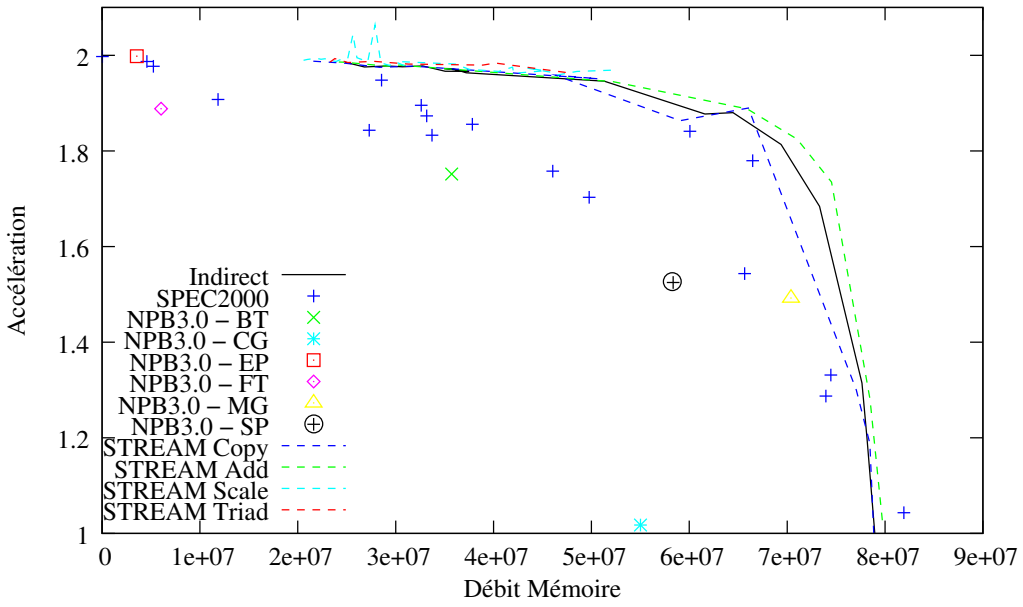

Figure 4.5 ***Biprocesseur ntel Pentium 4 :*** *Les résultats de cette courbe permettent d'établir un rapport entre l'accélération et l'utilisation mémoire des applications. Les applications peuvent être classées en deux zones :* non-saturée *et* intermédiaire. *Malheureusement, le comportement des applications ne permet pas une analyse fine de l'évolution de l'accélération selon le débit mémoire.*

d'accès à la mémoire, le meilleur événement trouvé pour représenter l'activité mémoire peut être un mauvais indicateur.

Néanmoins, il y a assez de résultats concluant identifiables par l'allure générale des courbes et la position des applications pour établir un rapport entre l'utilisation mémoire et l'accélération. On peut définir deux zones : une *non-saturée* et l'autre *intermédiaire/saturée*. Le problème de l'algorithme NAS CG peut être traité en déplacement la limite entre les zones d'un peu plus de 6.10^7 à un peu plus de 5.10^7. Cela aura pour effet de provoquer de fausses situations de saturation qui seront à prendre en compte dans l'évaluation finale du système de contrôle proposé.

4.3.2.3 Architecture processeur AMD 32bits

Dans cette section, nous présentons les évaluations menées sur une machine biprocesseurs composée par des processeurs AMD *Athlon 32bits* (voir tableau 4.1). Cette architecture utilise une approche point à point (voir section 2.1.1) pour la liaison entre les processeurs et le *chipset* donnant accès à la mémoire principale.

Cette structure répercute principalement le problème de la contention mémoire sur le *chipset*. La section 4.3.1 a montré que le problème de contention était bien présent et les

performances loin de 2 en terme d'accélération. C'est dans ce contexte que nous avons mené l'étude sur cette machine.

Les deux compteurs matériels de performance disponibles sur cette architecture ont le même format que ceux de la famille Intel *Pentium 6*. Cependant, leurs paramètres ne sont pas identiques et le nombre d'événements à compter est bien plus réduit. Dans la section 2.2, nous avons présenté ces compteurs et identifié l'ensemble des événements mémoire disponibles. Le nombre d'événements est très réduit, il est de 10 et portent principalement sur les requêtes liées aux mémoires *caches*. Vu le petit nombre d'événements, nous en avons fait le test exhaustif.

AMD Athlon : Parmi les programmes de test, nous avons débuté par ceux qui font des opérations sur des tableaux (STREAM et indirect). Le compilateur utilisé est celui du groupe *The Portland Group* avec l'option d'optimisation *"-fast"*. Après les tests de la totalité des événements liées à la hiérarchie mémoire, aucun événement capable de représenter la relation entre activité mémoire et accélération n'a été identifié.

La trop faible quantité d'événements à notre disposition empêche d'avoir une vision assez fine de l'activité de cette famille de processeurs. Cela ne nous a pas permis l'identifier la relation souhaité entre l'accélération et l'utilisation mémoire. Sur cette famille de processeur on ne peut donc pas proposé de mécanisme de contrôle d'exécution basé sur l'observation de l'activité mémoire.

4.3.2.4 Bilan

Tout au long de la section 4.3.2 nous avons présenté les résultats des observations sur le bus mémoire afin d'établir une relation entre l'accélération et le niveau d'activité mémoire. Grâce aux événements matériels, nous avons réussi à établir ce rapport souhaité sur 4 des 5 machines testées.

La difficulté de l'observation sur la machine AMD *Athlon* ($32bits$) ne vient pas ni l'utilisation des compteurs matériels de performances ni de la méthode de connexion des processeurs à la mémoire, mais du manque des événements liées à l'ensemble de la hiérarchie mémoire. Les événements mémoires de cette architecture n'ont comme cible principale que les mémoires *caches*.

Le rapport entre l'accélération et l'utilisation de la hiérarchie mémoire nous permet d'envisager le contrôle d'exécution. Le classement en 3 zones permet d'identifier de façon robuste et simple l'état de la hiérarchie mémoire et cela avec seulement 2 valeurs de débits d'événements.

Les résultats précédents permettent d'obtenir un mécanisme d'estimation de l'accélération en cours d'exécution nécessaire pour l'élaboration d'un mécanisme de contrôle

d'exécution. Ce mécanisme d'estimation va fonctionner en relevant périodiquement la valeur des compteurs matériels. Cet échantillonnage pose le problème de la stabilité de la grandeur observée, en effet tout système de contrôle ne peut fonctionner que si une forme de stabilité est présente. Des périodes d'activité moyenne stable sont beaucoup plus simples à gérer. Nous étudions le profil de l'activité mémoire dans la prochaine section.

4.3.3 Profil de l'activité mémoire des applications

Dans cette section nous étudions le profil de l'activité mémoire au cours du temps sur un ensemble d'applications.

Les applications retenues pour ces évaluations sont celles des SPEC2000 et des NAS. Les prises de mesures pour le calcul du débit mémoire ont été faites en relevant à chaque seconde la valeur du compteur matériel. L'événement retenu est celui qui a été sélectionné pour représenter l'activité du bus mémoire (*cf.* section 4.3.2).

Pour chaque courbes, les sous-sections suivantes sont obtenues de la manière suivante. Pour chaque application nous avons fait 10 exécutions. Puis, nous avons calculé la moyenne sur chaque mesure à l'échelle de la seconde, la variance à cette échelle est faible. Enfin, nous avons effectué un moyenne sur une fenêtre glissante de taille 5. Cette mesure reprend le principe que nous utiliserons ensuite dans le système de contrôle pour observer l'activité mémoire tout en effectuant un minimum de lissage.

Pour l'interprétation des résultats nous utilisons la notion de zone définie en section 4.3.2, qui classe le niveau de charge en 3 catégories : *non-saturée*, *intermédiaire* et *saturée*. Les frontières sont différentes pour chaque architecture et chaque machine, elles sont déterminées par l'observation des courbes de la section précédentes.

Architecture processeur de la famille Intel Pentium 6 : Dans la famille Intel *Pentium 6*, nous avons effectué les évaluations sur la machine *Pentium II* (*cf.* tableau 4.1). Le compilateur utilisé est *GNU 3.3* avec l'option d'optimisation *"-O3"*. Nous avons observé l'ensemble des programmes de tests SPEC2000.

Les frontière pour les zones précitée sont : 6.10^7 entre la zone *non-saturée* et la zone *intermédiaire* et puis $1,3.10^7$ entre la zone *intermédiaire* et la zone *saturée*.

Nous avons extrait de l'ensemble des résultats regroupés en annexe A quatre applications (ART, EON, APSI et MCF) dont le comportement temporel est significatif de l'ensemble des profils temporels observés. Le profil de ces applications est montré dans la figure 4.6. Au vu des allures des courbe obtenues, pour simplifier la description et adopter une approche similaire à celle utilisée dans le section 4.3.2, nous définissons trois types de comportement temporel : *constant*, *irrégulier* et *bruité*. La définition du premier comportement est triviale, est exprime bien les comportements temporels des applications

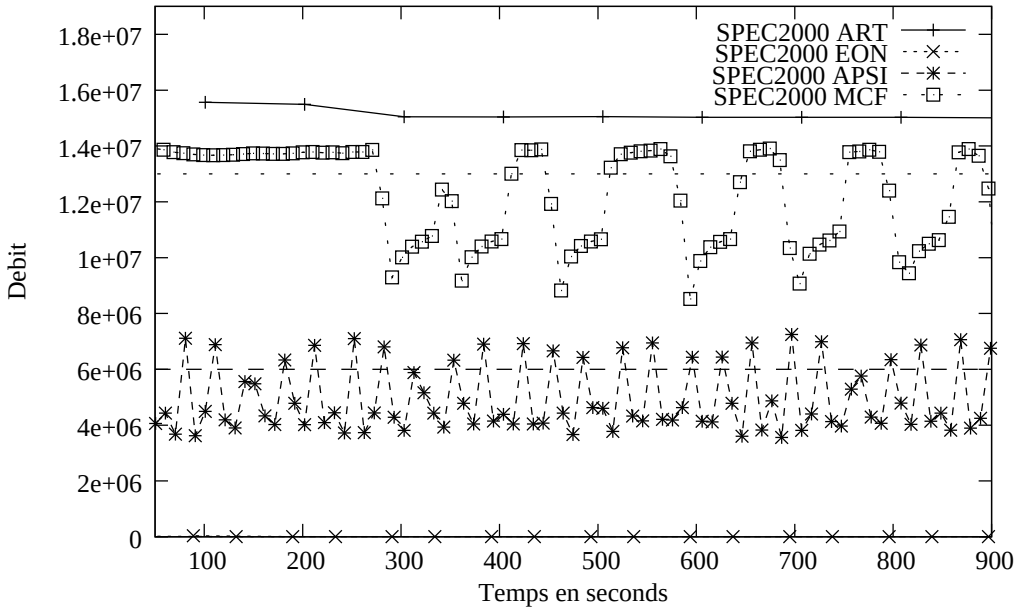

Figure 4.6 ***Biprocesseur Intel Pentium II :*** *Cette courbe présente les comportements de l'utilisation mémoire en fonction du temps. Nous avons pris* 4 *exemples, les deux premiers (ART et EON) sont classés comme ayant un comportement temporel* constant, *fonctionnant pour le premier dans une zone* saturée *et l'autre dans une zone* non-saturée. *Le troisième, l'application APSI est possède un comportement temporel irrégulier puisqu'elle change de zone fréquemment. Enfin, le dernier (MCF) est classé comme ayant un comportement temporel bruité car il change fréquentement de zone.*

ART et *EON*. A noter que ces deux applications ont un régime de fonctionnent différent : la premièrer fonctionne dans une zone *saturée* et la seconde dans une zone *non-saturée*.

Ensuite nous avons des applications comme *APSI* dont le comportement temporel peut être qualifié d'*irrégulier*. En effet cette application change fréquemment de zone, passant de la zone *saturée* à la zone *intermédiaire*.

Finalement l'application *MCF* présente un tout autre comportement que nous avons qualifié de *bruité*. Effectivement à l'observation de la courbe nous pouvons noter que le comportement temporel change très fréquemment mais avec une assez faible variabilité. Majoritairement cette application reste dans la zone *non-saturée*.

Il faut noter que les comportements temporels peuvent se succéder dans un même application. Par exemple le profil de l'application *MCF* débute par un comportement *constant* dans une zone *saturée* et puis elle est devenue *irrégulière*.

Pour finir, nous avons observé que 17 programmes SPEC2000 n'ont aucun changement de zone parmi la totalité des 20 applications (annexe A) et sont donc à comportement temporel *constant*.

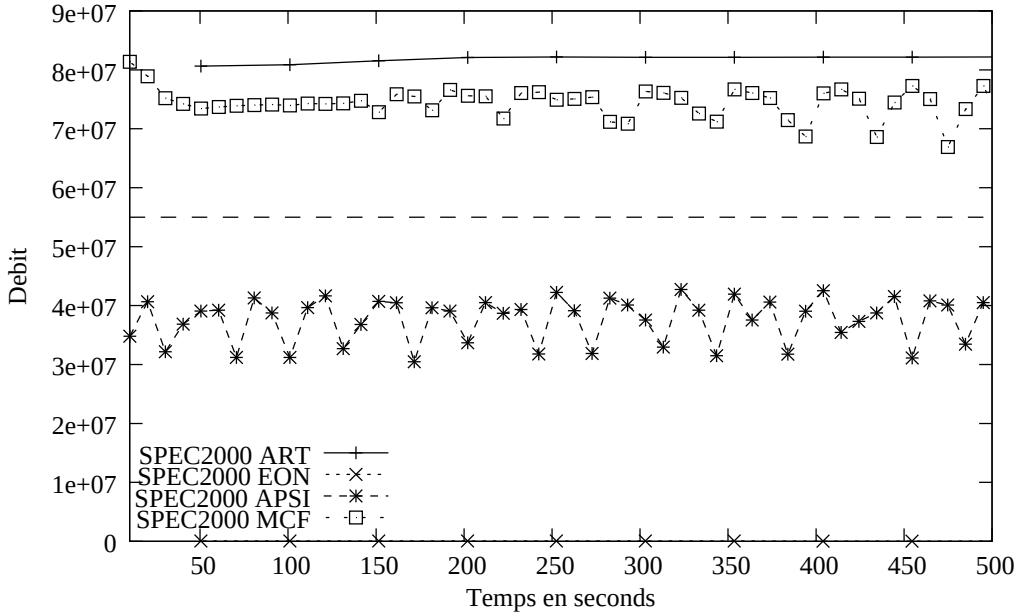

Figure 4.7 ***Biprocesseur Intel Pentium 4 :*** *Dans cette figure, nous présentons le comportement de l'utilisation mémoire en fonction du temps sur la machine biprocesseur* Pentium 4*. Les programmes sont les mêmes que ceux présentés pour le biprocesseur* Pentium II*. Ces quatre programmes sont classés comme ayant un comportement* constants*. Deux d'entre eux sont dans une zone* non-saturée *et les deux autres dans une zone* intermédiaire/saturée*. Il est essentiel de rappeler que l'événement matériel choisi sur cette machine n'a identifié que ces deux zones.*

Architecture processeur de la famille Intel Pentium 4 : La deuxième famille processeur testée a été l'Intel *Pentium 4*. Nous avons utilisé la machine biprocesseur *Pentium 4* présentée dans le tableau 4.1 et le compilateur *GNU 3.0* avec l'option d'optimisation *"-O3"*. Comme pour les tests précédents, nous nous sommes basés sur l'étude de la section 4.3.2 pour la définition des limites, plus précisément, sur les résultats des tests de cette même machine de la section 4.3.2.2. Ces résultats permettent l'identification de deux zones, une *non-saturée* et l'autre *intermédiaire/saturée*, avec une seule limite de débit d'événement mémoire $5.5.10^7$.

Comme nous l'avons déjà vu, le classement temporel du comportement des applications est basé sur le pourcentage de changement de zones en fonctions du temps d'exécution. L'événement matériel de l'architecture *Pentium 4* oblige une analyse plus grossière du comportement des applications.

La courbe de la figure 4.7 montre le comportement des quatre programmes SPEC2000 (ART, EON, APSI et MCF). Nous pouvons tous les classés comme ayant un comportement temporel *constant*, même s'ils ont quelques caractéristiques différentes. Par exemple, pour les applications ART et EON, le premier se situe dans une zone *intermédiaire/saturé* avec un niveau d'utilisation mémoire proche du maximum. Par contre, le deuxième ap-

partient à une zone *non-saturée* et a un niveau d'utilisation mémoire faible. Les deux autres applications, APSI et MCF présentent une plus grandes variabilités de comportement, mais cela ne les mène pas à faire des changements de zones. Enfin, le classement dynamique de ces applications montre un comportement *constant* pour tout l'ensemble des tests (voir annexe A).

Le classement fait dans cette section permet la caractérisation des comportements des applications au cours du temps. Dans ce travail, nous envisageons d'observer les applications à gros grain afin de rendre possible l'adaptation de l'exécution des applications avec des temps d'exécutions longs. Les évaluations faites constatent qu'il est possible de prévoir l'utilisation mémoire et par conséquent, les accélérations d'une période future en se basant sur un ensemble des mesures sur des périodes précédentes. Cependant, parfois les écarts entre les mesures rendent le mécanisme d'estimation de l'accélération impossible. Heureusement, le nombre des applications dans ce cas, classe irrégulière, est bien réduit.

4.3.4 Impact des activités d'entrées/sorties réseaux et disques sur la hiérarchie mémoire

La dernière étude présentée vise à observer l'impact des activités d'entrées/sorties réseaux et disques sur la hiérarchie mémoire.

Les applications retenues comme tâches de fond pour ces évaluations sont Netperf [60] avec les options "-l 100 -A 1048576", pour le réseau, et Bonnie [21] pour les accès au disque de stockage. Netperf est simple, il fait des transferts de données entre sockets via les protocoles IP TCP ou UDP. Pour Bonnie, nous avons gardé les deux premières phases d'exécution, celle qui fait l'écriture séquentielle de données et celle qui fait la lecture séquentielle de données. Les applications principales sont SPEC2000 ART avec un niveau fort d'activité mémoire et SPEC2000 EON avec un niveau faible. La méthodologie utilisée pour l'élaboration des courbes est la même que celle de la section précédente. Les courbes présentent alors les résultats des deux SPEC2000 ART en concurrence, plus chacune des tâche de fond, ensuite, deux SPEC2000 EON en concurrence plus chacune des tâche de fond. Elle est basée sur la moyenne d'une fenêtre glissante de taille 5.

L'analyse des résultats est faite selon les 3 zones définies dans la section 4.3.2 : *non-saturée*, *intermédiaire* et *saturée*.

Architecture processeur de la famille Intel Pentium 6 : Dans la famille Intel *Pentium 6*, nous avons effectué les évaluations sur la machine *Pentium II* (*cf.* tableau 4.1). Cette machine contient un disque de stockage avec un contrôleur *SCSI* (Small Computer System Interface) et une carte réseau standard ethernet 100Mbit/s. Le compilateur utilisé est $GNU3.3$ avec l'option d'optimisation "-O3".

Les deux applications utilisées, SPEC2000 ART et SPEC2000 EON, ont été classées

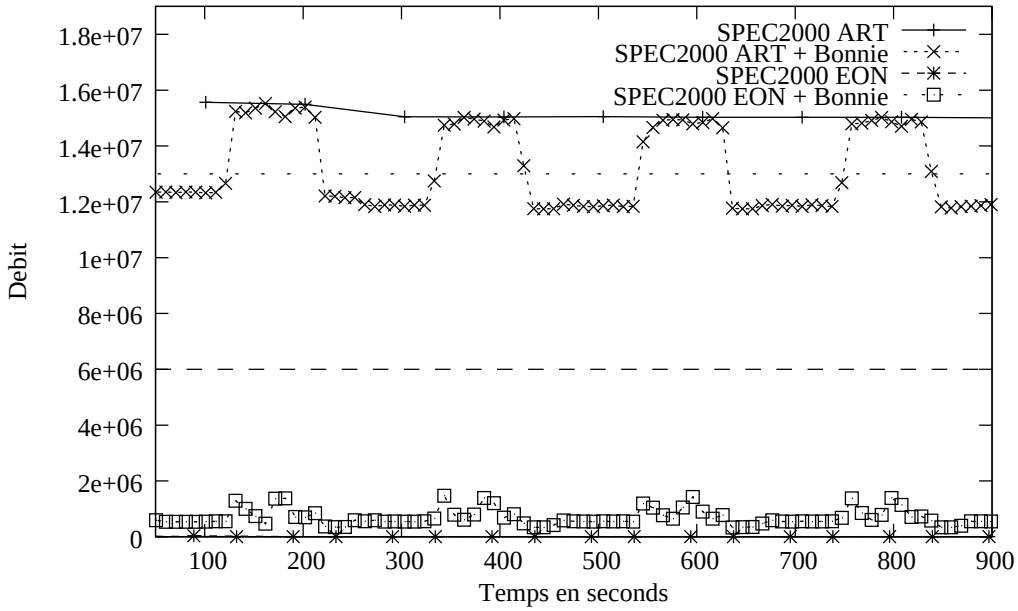

Figure 4.8 ***Biprocesseur Intel Pentium II :*** *Cette courbe présente les comportements de deux applications, SPEC2000 ART et SPEC2000 EON, avec et sans la présence d'une tâche de fond. Dans ce cas, la tâche de fond est le programme de test Bonnie. L'exécution de l'application SPEC2000 ART change de régime pendant la phase d'écriture de Bonnie et revient au même niveau qu'avant dans la phase de lecture. Pour l'application EON, on note une augmentation des activités mémoire, néanmoins, elle reste dans une zone* non-saturée.

selon leur comportement temporel (section 4.3.3) comme *constant*. Dans la figure 4.8, nous présentons les activités mémoire effectuées par ces applications en fonction du temps avec et sans la tâche de fond. Dans ce cas, la tâche de fond est le programme de test Bonnie. Pour l'application SPEC2000 ART, un changement de régime est remarqué à chaque phase d'écriture sur le disque par la tâche de fond. Elle passe d'une zone *saturée* à *intermédiaire*. Cependant, la lecture n'implique pas d'impact sur les activités mémoire. Pour l'application SPEC2000 EON, même si le niveau d'activité sur la hiérarchie mémoire augmente, il est resté encore très bas (*non-saturée*).

Dans la figure 4.9, nous avons les résultats avec la tâche de fond réseau, qui est obtenu avec le programme de test Netperf. L'application SPEC2000 ART comme l'application SPEC2000 EON ne change pas de régime. Néanmoins, dans le cas de l'ART, le niveau d'activité mémoire baisse un peu, et dans le cas de l'EON, le niveau augmente un peu.

Architecture processeur de la famille Intel Pentium 4 : La deuxième famille testée a été celle de l'Intel *Pentium 4*. Elle est représentée par la machine *Pentium 4* biprocesseur (*cf.* tableau 4.1). Cette machine a un disque de stockage avec un contrôleur *SCSI* et une carte réseau ethernet 100Mbit/s. Enfin, le compilateur utilisé est $GNU3.0$

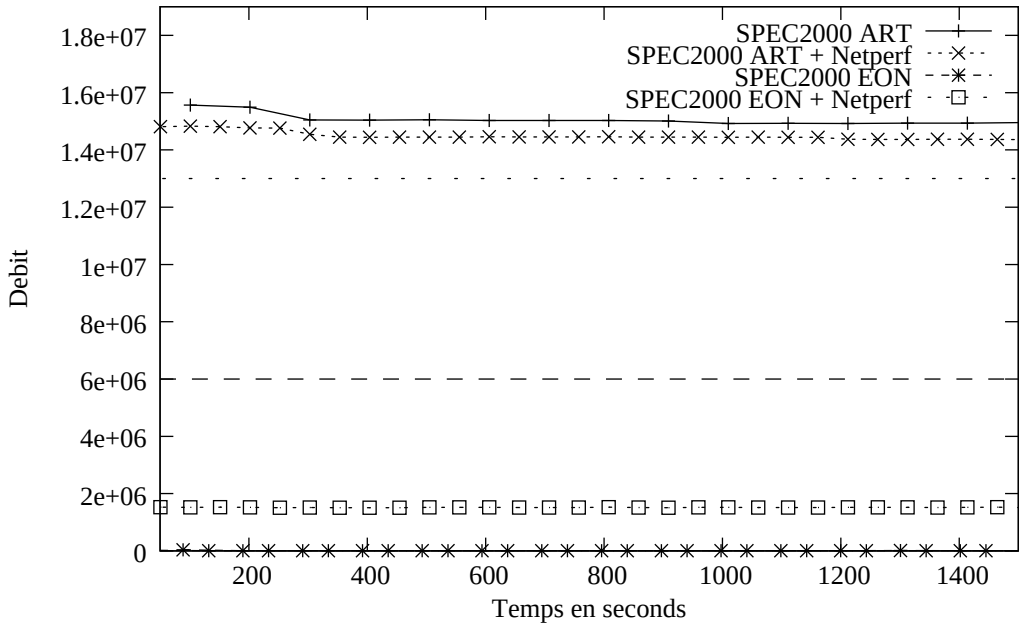

Figure 4.9 ***Biprocesseur Intel Pentium II :*** *Cette deuxième courbe présente les résultats avec la tâche de fond réseau. Le programme de test utilisé est Netperf. Pour l'application SPEC2000 ART, nous avons une légère réduction du niveau d'activité mémoire, et pour l'application SPEC2000 EON, nous observons une légère croissance de ce niveau. Néanmoins, aucune des deux applications ne manifeste des changements des régimes provoqués par cette tâche de fond.*

avec l'option d'optimisation "-O3".

Dans le premier test, nous avons utilisé les mêmes applications que précédemment : SPEC2000 ART et SPEC2000 EON. Dans cette architecture, nous n'avons que deux zones, une *non-saturée* et l'autre *intermédiaire/saturée*. Dans la figure 4.10, nous présentons les résultats pour la tâche de fond d'accès disque : le programme de test Bonnie. On note que le niveau d'activité mémoire s'affaiblit pour l'application SPEC2000 ART. Au contraire, l'application SPEC2000 EON présente une augmentation du niveau d'activité mémoire. Les applications restent dans les mêmes zones qu'avant, *non-saturée* pour SPEC2000 ART et *intermédiaire/saturée* pour SPEC2000 EON.

Pour le second test, nous lançons Netperf comme tâche de fond. Les applications, SPEC2000 ART et SPEC2000 EON, ne changent pas de régime, et ont des comportements semblant à ceux avec la tâche de fond d'entrée et de sortie. Nous avons remarqué une légère baisse du débit mémoire pour le SPEC2000 ART, et une légère croissance du débit pour le SPEC2000 EON.

Finalement, les activités du réseau ainsi que celles des accès disques provoquent de légers changements sur les débits mémoires des applications observées. Cela dit, il y a une seule situation de changement de régime, celle où l'application SPEC2000 ART

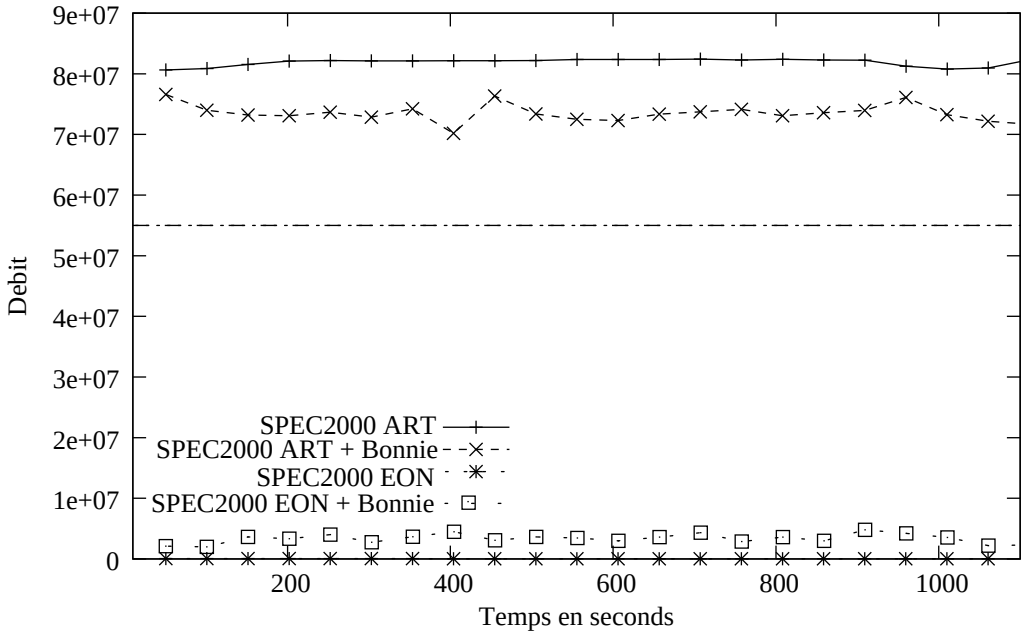

Figure 4.10 ***Biprocesseur Intel Pentium 4 :*** *Dans cette figure, nous présentons le niveau d'activité mémoire de deux applications, le SPEC2000 ART et SPEC2000 EON, avec une tâche de fond. Dans ce cas, le programme de tests utilisé est Bonnie. On note que l'exécution de la tâche de fond Bonnie en concurrence à deux autres instances de l'application SPEC2000 ART affaiblit le niveau d'activité mémoire. Pour l'application SPEC2000 EON, on note exactement le contraire, il augmente légèrement.*

s'exécute sur le biprocesseur *Pentium II* en concurrence avec la tâche de fond disque. Par conséquent, nous pouvons supposer, comme première approximation, que les activités en tâches de fond d'entrées/sorties n'ont pas un impact important sur le comportement des applications.

4.3.5 Synthèse

Nous avons présenté les résultats de l'évaluation des programmes de tests afin de vérifier l'impact de l'utilisation mémoire sur des machines multiprocesseurs. Nos premières évaluations ont montré que les algorithmes classés comme *Memory Bound* ont des performances loin des optimums, et ceux classés comme *CPU Bound* ont des performances proches des optimums.

Les analyses sur ces résultats préliminaires ont montré aussi que toutes les architectures $32bits$ testés ont des faiblesses au niveau de la hiérarchie mémoire. Les deux autres machines, l'Itanium 2 $64bits$ et l'*Opteron* $64bits$, utilisées sur les plateformes évaluées sont plus équilibrées au niveau du débit mémoire et fréquence processeur.

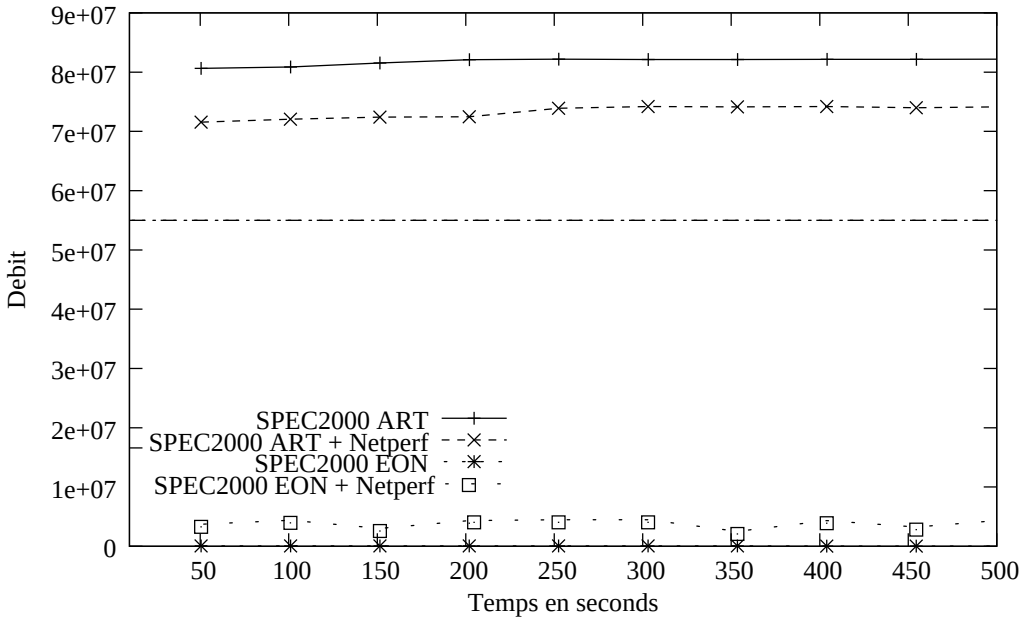

Figure 4.11 ***Biprocesseur Intel Pentium 4 :*** *Cette figure présente les résultats de deux applications : SPEC2000 ART et SPEC2000 EON. La première application baisse légèrement son débit dû l'influence de la tâche de fond. La deuxième application augmente légèrement son débit. Cela étant, ni l'une ni l'autre ont des changements de régimes.*

Nous avons aussi montré le mécanisme utilisé pour établir la relation entre l'accélération et l'activité mémoire. Nous avons constaté que, même avec certaines restrictions, les machines de la famille Intel *Pentium 6* et *Pentium 4* ont des événements matériels capables d'établir cette relation. Malheureusement, l'architecture des processeurs AMD *Athlon* $32bits$ ne permet pas la même chose. Nous obtenons le rendement en divisant l'accélération estimée par le nombre de processeur. Puisque nous pouvons obtenir l'estimation de l'accélération en cours d'exécution il en va de même avec le rendement.

Dans le cadre de ce travail, nous envisageons de prévoir les performances des applications. Cela dit, la prochaine étape a été consacrée à la vérification du comportement mémoire des applications en fonctions du temps. Les résultats sur les deux architectures évaluées ont montré que la grande majorité des applications possèdent un comportement constant. Puisque les applications ont des comportements mémoire constants nous pouvons espérer que l'accélération estimée d'une période soit la même que celle de la prochaine période. Enfin, notre mécanisme montre que nous sommes capables de prévoir les performances.

4.4 Bilan

Au cours de ce chapitre, nous avons présenté une étude de l’impact de l’utilisation de la mémoire sur quelques machines multiprocesseurs. Les évaluations sur un ensemble de programmes de tests avec différentes caractéristiques ont confirmé les problèmes de contention sur la hiérarchie mémoire sur ces machines. Une façon de traiter ce problème est de soulager un ou plusieurs composants de la hiérarchie mémoire. Cependant, l’identification d’un composant de la hiérarchie mémoire qui permet de représenter les performances d’une application au cours d’exécution est complexe.

Nous avons décrit une approche capable d’identifier les performances au cours d’exécution, laquelle permet d’envisager un contrôle d’exécution des applications. Cette approche est basée sur l’observation des activités sur le bus mémoire. Dans un premier temps, nous avons étudié les programmes de tests existants en essayant de rassembler un groupe avec les différentes caractéristiques au niveau de l’utilisation des ressources.

Nous achevons ce chapitre par des évaluations de l’impact de l’utilisation du réseau et du disque sur les applications et leurs accès sur le bus mémoire. Les impacts observées sont relativement limités ce qui nous permet de les ignorer en première approximation.

En conclusion, nous avons réuni les conditions principales (estimation du rendement en cours d’exécution et stabilité du comportement) nécessaires à réalisation d’un système de contrôle d’éxecution. Dans le prochain chapitre nous présentons les principe d’un tel système ainsi qu’un premier prototype.

5

DRAC : Système de contrôle d'exécution

Dans le cadre de ce travail de thèse, nous avons étudié l'impact de l'utilisation de la hiérarchie mémoire sur les performances des applications sur les multiprocesseurs à mémoire partagée. Dans un premier temps, nous avons présenté le classement des architectures processeurs utilisées sur les multiprocesseurs ainsi que quelques limitations conceptuelles ou technologiques qui peuvent être sources d'inefficacités.

La suite de cette étude a été consacrée à l'observation des activités des composants responsables des sources d'inefficacités. Pour cela, nous avons utilisé les compteurs matériels de performances (voir chapitre 2) afin de faire une observation à un grain très fin de ces composants de la hiérarchie mémoire, tel que le bus mémoire.

Les évaluations présentées dans le chapitre 4 ont permis de déterminer une relation liant l'utilisation du bus mémoire et les performances des applications. Cette relation permet ainsi d'estimer les performances des applications en cours d'exécution et donc du rendement à travers l'observation de l'utilisation du bus mémoire.

C'est dans ce contexte que nous proposons, dans ce chapitre, le système de contrôle DRAC. L'ordonnanceur de ce système est basé sur une politique de prise de décisions suivant l'utilisation du bus mémoire. Son principal objectif est donc d'ordonnancer les processus de telle façon que la hiérarchie mémoire ne soit pas saturée.

5.1 Introduction

Le système de contrôle de processus, DRAC (*Adaptive Control System with Hardware Performance Counters*), est basé sur l'architecture générale des systèmes adaptables. DRAC est un système extérieur aux applications et au système d'exploitation, qui n'intervient sur l'exécution de l'ensemble des applications contrôlées que dans le cas d'une mauvaise utilisation de la hiérarchie mémoire.

Une des sources d'inefficacité sur les machines multiprocesseurs à mémoire partagée est la contention sur la hiérarchie mémoire. Dans l'objectif de maximiser le rendement des multiprocesseurs (vision de l'administrateur), DRAC ordonnance les applications en fonction de leurs besoins en mémoire.

DRAC a aussi été conçu dans l'objectif d'intervenir sur des applications à temps d'exécution moyen et long, afin de limiter les perturbations générées par un nouvelle ordonnancement. De même les applications visée sont celles lancée en mode *batch*.

La première section de ce chapitre présente le principe du système en donnant une idée générale de l'ordonnanceur et du traitement de DRAC pour les problèmes de synchronisation. Ensuite, nous décrivons l'architecture du système et concluons avec une présentation du prototype.

5.2 Principe du système

Dans cette section, nous présentons les principales caractéristiques du mécanisme de contrôle du système DRAC. Ce mécanisme est activé seulement si l'environnement d'exécution dispose des conditions favorables à son fonctionnement. Les principales conditions sont qu'il existe des applications prêtes à s'exécuter et que ces applications aient des besoins en débit mémoire différents.

Dans ce contexte, nous visons à améliorer le temps total d'exécution d'un ensemble d'applications. Le point de départ est le problème de la contention mémoire qui provoque la chute des performances des applications et par conséquent celle du rendement de la machine. DRAC va jouer sur l'ordonnancement des applications en fonctions de leur utilisation mémoire afin de réduire le problème. Dans un premier temps, le système observe la charge courante sur le bus mémoire et les activités faites par chaque application en cours d'exécution. Si l'utilisation du bus mémoire est saturée ou sous-utilisée, le mécanisme de contrôle se déclenche en ordonnançant les processus disponibles afin d'empêcher ou de limiter cette situation. La politique mise en place évite alors, d'une part, le ralentissement de l'exécution des processus provoqué par la saturation de la ressource mémoire, et d'autre part, le gaspillage de cette même ressource.

Prenons l'exemple de la figure 5.1 sur lequel il y a quatre processus, deux avec un fort

besoin en débit mémoire, appelé *processus lourds*, et deux autres avec un faible besoin en débit mémoire, appelé par la suite *processus légers*. Le temps d'exécution de chacun de ces processus en séquentiel est de 5 unités de temps, cependant, lorsque des *processus lourds* s'exécutent ensemble leurs temps d'exécutions augmentent de 60% du fait de la contention mémoire (de 5 à 8 unités de temps).

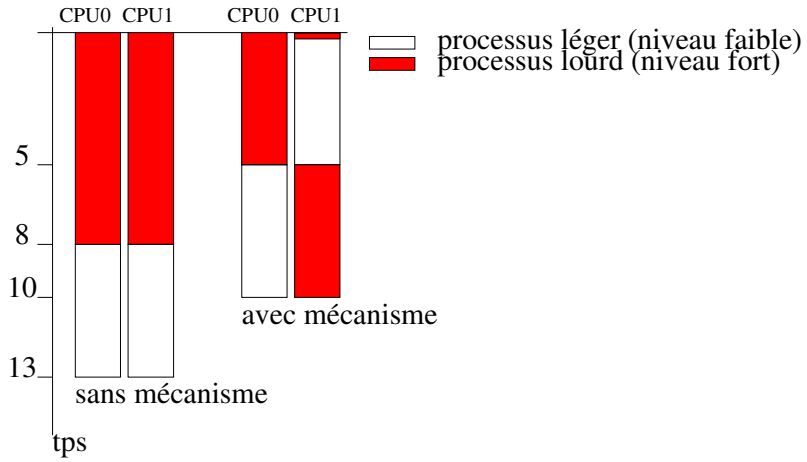

Figure 5.1 *Cet exemple contient deux processus lourds, c'est-à-dire des processus avec un fort niveau d'utilisation mémoire, et deux autres processus légers, c'est-à-dire des processus avec un faible niveau d'utilisation. Dans le premier cas, les deux processus lourds sont lancés et suivis par les deux processus légers. Dans le deuxième cas, DRAC ordonnance l'exécution d'un processus lourd avec un autre processus léger afin d'éviter la saturation de la ressource mémoire. Enfin, le temps d'exécution de l'ensemble est de* 13 *unités de temps dans le premier cas et d'environ* 10 *unités de temps dans le deuxième.*

Nous présentons deux situations pour l'exemple décrit, la première *sans le mécanisme* de contrôle de DRAC et la seconde avec ce *mécanisme*. L'exécution sans le mécanisme peut ordonnancer des applications du même type en concurrence, ce qui provoque le ralentissement de l'exécution du couple des *processus lourds*. Par conséquent, le temps total d'exécution est de 13 unités de temps. La deuxième situation avec le mécanisme DRAC identifie rapidement la saturation du bus mémoire quand le couple des *processus lourds* s'exécutent et remplace l'un des *processus lourds* par un autre *processus léger*. Dans ce cas, nous avons un temps total d'exécution d'environ 10 unités de temps.

L'exemple avec le mécanisme est une vue simplifiée de l'ordonnanceur du système DRAC. En effet, l'ordonnancement des processus exige plus que le simple changement de l'ordre d'exécution, puisque le système a besoin de prendre en compte la gestion de l'ensemble des processus qui s'exécute en concurrence. L'identification et le contrôle des processus concurrents, sans aucune dépendance, est trivial. Le système DRAC intercepte simplement les opérations de création et destruction de ces processus en les ajoutant et les effaçant de sa liste de processus à contrôler. Dans le cas de processus avec des dépendances, c'est-à-dire des programmes avec des points de synchronisations, le système

de contrôle doit faire en sorte de ne pas introduire de retard de traitement excessifs qui allongerai le temps de complétion des applications. Le système doit identifier ces points et modifier en conséquence son comportement. L'approche la plus simple pour faire face à des programmes avec synchronisation et de les traiter globalement, c'est-à-dire que les processus de ces applications sont ordonnancés toujours simultanément. Ainsi nous garantissons de ne par retarder l'exécution d'un processus par rapport à un autre processus de la même application en attente sur un point de synchronisation. L'inconvénient principale est de laisser de coté des situations où l'on pourrait augmenter le rendement en jouant sur l'ordonnancement des processus d'une même application. Par exemple, dans le cas d'une application parallélisée avec les directives OpenMP s'exécutant sur un biprocesseur, il peut se trouver des parties parallèles saturant la hiérarchie mémoire. Il peut être intéressant de modifier l'ordonnancement de cette application en faisant s'exécuter un *thread* après l'autre, conjointement avec un processus ne chargeant pas la hiérarchie mémoire.

Parmi toutes les situations nécessitant des synchronisations nous avons retenus deux type de situations regroupant une majorité de cas de figure. La première est celle nécessitant l'exécution d'une région (ou section) critique en *exclusion mutuelle*. C'est un cas de figure très courant visant à éviter l'occurrence de conditions de course (*race conditions*). La méthode de l'*exclusion mutuelle* consiste à n'autoriser qu'un processus à la fois dans une section critique. La figure 5.2 (A) présente un déroulement temporel d'un accès en *exclusion mutuelle* à une section critique. Il existe une grande variété d'algorithmes réalisant l'*exclusion mutuelle* [106] : les sémaphores, les moniteurs, le mutex, le fast mutex, le *spin lock*, la solution de Peterson ainsi que les primitives de *sleep* et *wake-up*.

Le seconde type de situation, courante et importante dans le domaine des applications du calcul scientifique, sont les fonctions de synchronisation que nous regroupons ici sous le terme de *barrières*. Une *barrière*, sa version la plus simple, peut être définie comme un point de synchronisation où un groupe de processus sont mis en attente jusqu'au moment tous les processus du groupe soient arrivés ce qui libèrent l'ensemble des processus. La figure 5.2 (B) présente le déroulement d'une telle *barrière*. En dénombre plusieurs variantes autour de la condition de libération et du nombre de processus libérés. Toutefois la *barrière* simple représente un majorité des cas de figure avec la fonction de synchronisation join utilisé pour attendre la terminaison d'un *thread*.

Nous l'avons déjà décrit, le mécanisme de contrôle DRAC, en cas de besoin, remplace de processus ce qui signifie la suspension l'exécution d'un processus. La synchronisation d'un ou d'un groupe de processus exigent donc d'être pris en compte par le système DRAC au moment de la suspension d'un processus.

Le traitement de l'*exclusion mutuelle* et de la *barrière* par DRAC peut être plus ou moins complexe selon la technique d'attente utilisée. En fait, il existe deux techniques pour gérer l'attente au point de synchronisation d'une région en *exclusion mutuelle* et d'une *barrière* : l'approche par **attente active** et l'approche par l'**attente passive**.

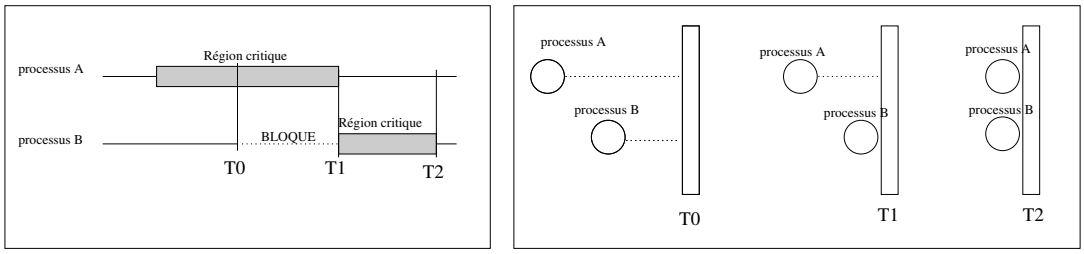

Figure 5.2 *À gauche, nous présentons un scénario d'interdiction d'accès à une région critique (*exclusion mutuelle*), et à droite, une scénario du contrôle d'exécution d'un groupe de processus (une* barrière*).*

attente active : Dans cette approche, les processus en attente restent actifs dans une boucle renouvelant le test d'accès à chaque itération. Cette approche est naturellement consommatrice de temps processeur, par contre, elle a l'avantage d'être la plus réactive.

attente passive : La deuxième approche est basée sur le principe de suspension de l'exécution des processus. Pour la méthode d'*exclusion mutuelle*, à l'entrée d'une région critique, si le processus est tout seul dans cette région alors il s'exécute sinon il est suspendu. À chaque fois qu'un processus sort de la région critique, un autre processus est réveillé s'il en existe encore. Pour la *barrière*, les processus s'exécutent normalement jusqu'au point de synchronisation. Ils sont alors suspendus jusqu'à l'arrivée du dernier processus du groupe. L'attente passive évite la consommation excessive de temps processeur, par contre, elle peut provoquer des changements de contexte non nécessaires et elle est aussi plus coûteuse en temps de traversée.

Enfin, il existe des algorithmes mélangeant les deux approches. Dans un premier temps, l'attente active est utilisé et après un certain temps le processus est suspendu (attente passive). Cette approche vise à trouver un bon compromis entre le gaspillage de temps du processeur et la réactivité.

Le système DRAC traite les cas d'*exclusion mutuelle* indépendemment de l'approche d'attente (passive ou active) utilisée. Lorsqu'un processus entre dans une région critique il n'est plus contrôlé par le système, c'est-à-dire le système de contrôle ne fera pas sa suspension. Une fois sorti de la région critique le processus devient de nouveau contrôlable.

Dans le cas de la *barrière*, le problème est d'éviter de prolonger l'attente des processus. L'intervention du mécanisme de contrôle peut provoquer cet allongement s'il décide de suspendre un processus attendu à une *barrière*. Dans le cas ou l'ordonnanceur de DRAC a décidé de réduire le nombre de processeurs allouer à l'application pour optimiser le rendement, l'approche proposée est la suivante. Au moment où un processus arrive sur la *barrière* il est suspendu et remplacé par un autre processus du même groupe et cela jusqu'au dernier. Tous les processus sont alors réveillés pour passer cette *barrière*. Cette

approche fonctionne avec les 2 types d'attente. Avec l'attente passive c'est la *barrière* qui suspend les processus. Il faut noter que si les exécutions de *barrière* sont très rapprochées les surcoûts d'intervention (suspension/réveil de processus) peuvent obliger à inhiber le système de contrôle pour des besoins de performance. Nous rappelons que le système DRAC est destiné aux applications ayant des cycles de comportement de durée moyenne à longue.

Pour mettre en pratique les approches précédent il est nécessaire de d'identifier les zones d'*exclusion mutuelle* et les opérations de *barrière*. L'identification peut-être faite à plusieurs niveaux. Ainsi pour assurer l'*exclusion mutuelle* on utilise généralement des opérations atomiques ou de verrouillages au niveau matériel qui dans certaines conditions sont détectable avec les compteur matériel. Un autre approche est l'analyse de code binaire avant le lancement de l'application pour détecter la présence d'opération atomique, dans ce cas il faut s'assurer qu'il ne peut pas y avoir de génération de code à l'exécution qui pourrait comportait des opération atomique.

Finalement, on constate que les applications nécessitant des synchronisations utilisent généralement de bibliothèques spécialisées, comme *LinuxThread [84], nptl [41], pth [42],* ou encore des environnement de programmation parallèle comme OpenMP [103]. Dans ce cas il est plus un instrumentation du code est plus indiqué. C'est le choix que nous avons retenu (voir section 5.3.4.2).

Pour finir sur le contrôle des programmes comportant des synchronisations, on peut noter qu'avec les environnements de programmation parallèle et sous certaines condition il y a des solutions relativement simple. Par exemple dans le cas d'OpenMP, une solution est de modifier au besoin (suivant l'ordonnanceur) le nombre de processus via les directives *omp_set_num_threads*. La condition principale nécessaire à l'emploi de cette technique est que l'application doit supporter le changement du nombre de processus en cours d'exécution.

5.3 Architecture DRAC

Dans cette section nous présentons l'architecture du système de contrôle d'exécution DRAC. Dans le chapitre 3 nous avons présenté l'architecture des systèmes adaptables. Le système DRAC a globalement la même architecture que celle présentée dans la figure 3.5.

Les modules d'un système adaptable sont : **le capteur**, **l'effecteur**, **l'analyse des données** et **la prise de décisions**. Les fonctionnalités de ces modules sont, respectivement : observer le comportement du système et appliquer les adaptations, faire le traitement statistique sur les données et déterminer un nouvel ordonnancement selon la politique utilisée.

En observant l'architecture DRAC présentée dans la figure 5.3, nous pouvons iden-

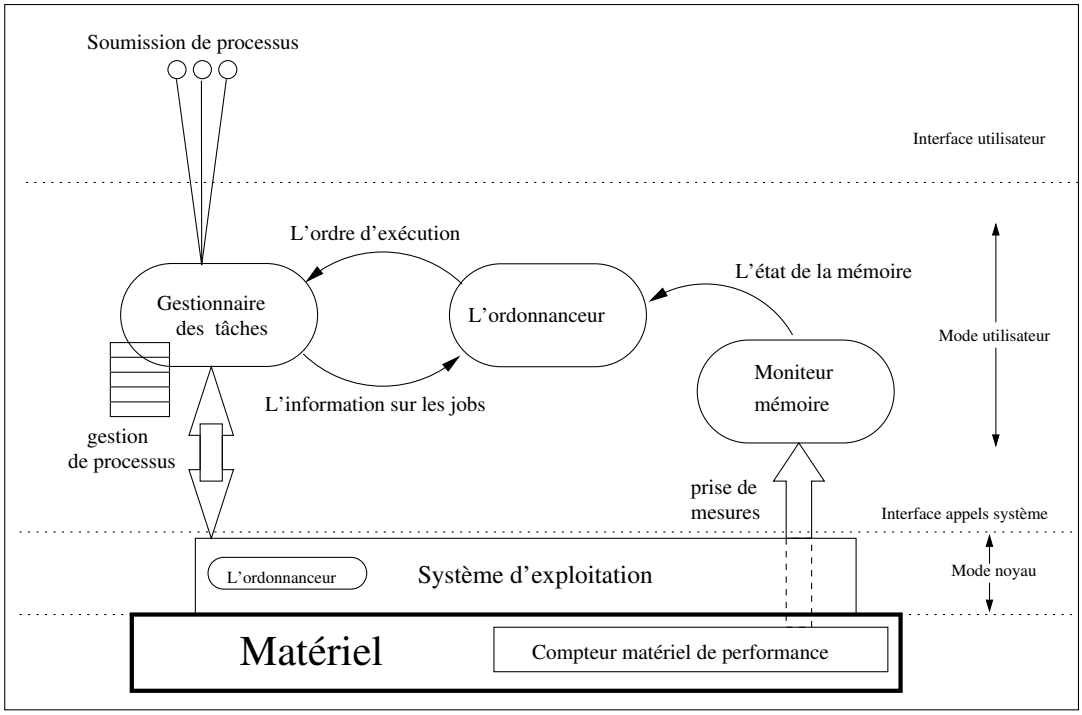

Figure 5.3 *L'architecture du système DRAC se place au niveau utilisateur et elle est composée essentiellement de trois modules :* le gestionnaire de tâches, l'ordonnanceur *et* le moniteur mémoire. *Le gestionnaire gérer la soumission, le retrait, la suspension et l'activation des processus. L'ordonnanceur fait l'analyse des données collectées par le moniteur mémoire et propose un nouvel ordonnancement. Enfin, le moniteur mémoire observe et traite statistiquement les activités sur le bus mémoire.*

tifier cet ensemble de fonctionnalités. Le premier module, gestionnaire de tâches (voir section 5.3.4), gère les processus. L'**ordonnanceur** (voir section 5.3.3) est le module qui réalise la prise de décision. Enfin, le **moniteur mémoire** (voir section 5.3.2) est le responsable de l'observation des activités sur le bus mémoire et de l'analyse statistique des données collectées.

L'ensemble des composants du système DRAC se trouve dans le niveau utilisateur. Cela s'explique essentiellement par la simplicité d'implantation d'un système de contrôle dans ce niveau. Ainsi son développement peut être fait avec des fonctions standard des bibliothèques de base ce qui permet d'avoir un système portable. Une autre raison vient de la simplicité de développement au niveau utilisateur par rapport à un système au niveau noyau. Cela étant, le temps de réaction de DRAC au niveau utilisateur est plus lent que s'il était dans le niveau noyau. Sachant que DRAC vise à contrôler des applications ayant un temps d'exécution plutôt long, son temps de réaction n'est pas prioritaire, excepté pour les fonctions de synchronisation (*c.f.* section 5.3.4.2).

5.3.1 Etat de processus dans le système DRAC

Dans la section 5.2 nous avons présenté le principe du système DRAC. Le contrôle d'exécution des processus de ce système revient essentiellement à changer l'ordre d'exécution des processus, suivant leurs besoins en mémoire. Parallèlement au état d'un processus appartenant au système d'exploitation, le système DRAC a besoin de maintenir un état par processus pour assurer leur contrôle.

Pour un système d'exploitation et dans le cas le plus simple on définit trois états pour un processus [106] : **en cours d'exécution** (c'est-à-dire utilisant le processeur à cet instant), **prêt** (exécutable, temporairement arrêté pour laisser s'exécuter un autre processus), et **bloqué** (ne pouvant pas s'exécuter tant qu'un événement externe ne se produit pas).

Cependant, dans les systèmes réels il est généralement nécessaire de préciser encore quelques situations. Par exemple, dans le système d'exploitation Linux un processus peut avoir cinq états :

- **prêt à s'exécuter** (*runnable*) : le processus est dans la queue d'exécution (*run-queue*).
- **endormi non-interruptible** (*uninterruptible sleep*) : il est endormi en attente non-interruptible. Cet état arrive généralement quand un processus fait des entrées et des sorties. Les processus qui sont dans cet état ne peuvent pas être suspendus.
- **endormi** (*sleeping*) : il est bloqué en attendant un événement externe.
- **stoppé** (*traced or stopped*) : il est simplement suspendu.
- **zombie** (*a defunct process*) : les processus qui sont terminés attendent que le père effectue l'appel système *wait*. De tels processus sont appelés des processus zombies.

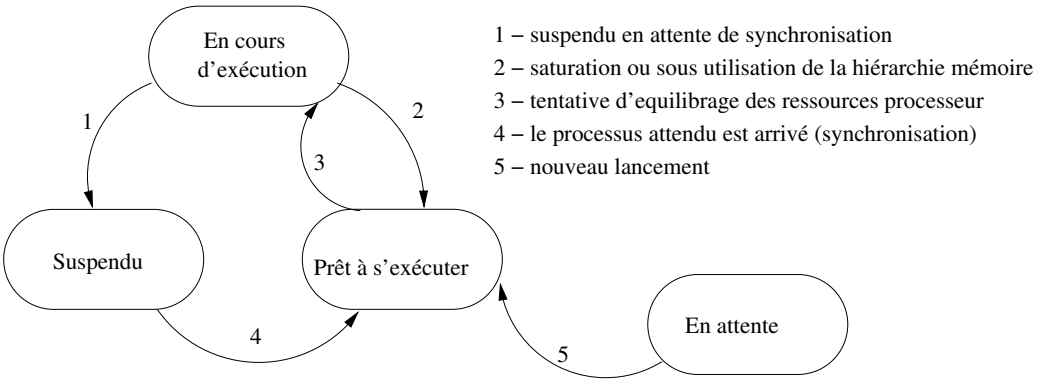

Figure 5.4 *Un processus DRAC peut avoir quatre états : en attente (soumis au système DRAC), suspendu (en attente de synchronisation ou de la libération d'une région critique), prêt à s'exécuter (temporairement suspendu pour laisser s'exécuter un autre processus) et en cours d'exécution (à la charge du système d'exploitation).*

Un processus contrôlé par le système DRAC contient des états qui sont parallèles à

ceux du système d'exploitation. Dans certains cas, le processus entre dans le système DRAC avant même d'être lancé par le système d'exploitation. De plus, il doit aussi prendre en compte la synchronisation entre les processus. Un processus vu par DRAC peut avoir quatre états (figure 5.4) :

- **en cours d'exécution :** DRAC laisse l'ordonnanceur du système d'exploitation s'en occuper.
- **prêt à s'exécuter :** exécutable, temporairement suspendu pour laisser s'exécuter un autre processus DRAC. L'état de ce processus vu par le système d'exploitation est donc stoppé.
- **suspendu :** suspendu, en attente de la libération d'une région critique ou d'une *barrière*. Pour le système d'exploitation le processus est simplement stoppé.
- **en attente :** soumis mais ne pas lancé, dû à la limitation en mémoire du système. Ce processus est inconnu par le système d'exploitation.

Dans la figure 5.4 nous présentons les états des processus DRAC et les événements qui provoquent leur changement. Un processus entre dans le système DRAC via une soumission et reste dans l'état *en attente* jusqu'à ce qu'il soit lancé. Une fois que le processus est lancé il passe à l'état *prêt à s'exécuter*, il devient donc candidat à l'exécution. Puis, un processus qui est dans l'état *en cours d'exécution* peut passer soit à état *suspendu* soit à *prêt à s'exécuter*. Il est soit dans l'état *suspendu*, s'il attend une *barrière*, par exemple, ou à l'entrée d'une région critique, soit dans l'état *prêt à s'exécuter*, si l'ordonnanceur DRAC essaie de faire un nouvel ordonnancement.

L'ensemble des informations des processus soumis à DRAC est sauvegardé dans une liste de processus, laquelle est partagée par quelques modules du système DRAC. Les prochaines sections sont consacrées aux modules du système.

5.3.2 Moniteur mémoire

Le premier composant du système que nous présentons est le moniteur mémoire qui fait essentiellement l'observation de l'utilisation de la hiérarchie mémoire. Il utilise l'approche d'observation par surveillance (voir chapitre 3.2) pour collecter les événements matériels autour du bus mémoire adapté à l'architecture processeur. Il est composé de trois modules (figure 5.5) : le capteur, le traitant et l'analyseur.

Le capteur est responsable de la collecte des activités du bus mémoire. Pour cela, il fait des prises de mesure à intervalle régulier (échantillonnage) des compteurs de performances directement au niveau noyau, à travers la bibliothèque d'accès aux compteurs. Ce module configure ainsi l'événement matériel adapté à l'architecture processeur utilisée.

Une fois les informations prises, le traitement des données s'effectue par l'intermédiaire du module traitant. Son activité est simple mais importante. Il prend les informations brutes venues des compteurs, tels que le nombre de transactions sur le bus mémoire

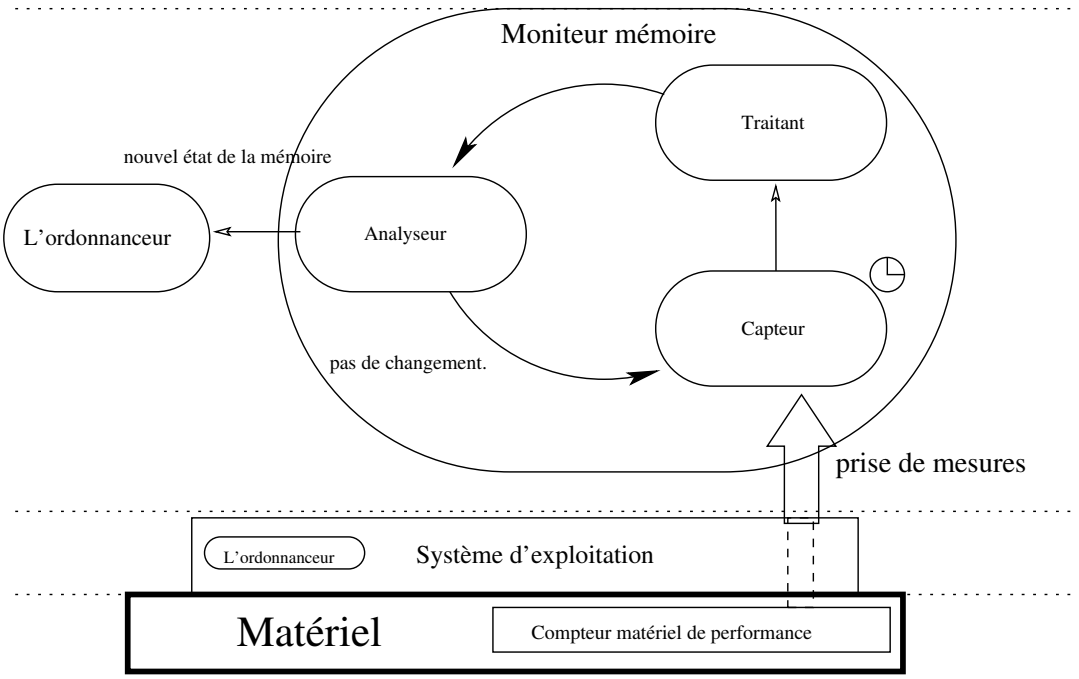

Figure 5.5 *Les trois composants du moniteur mémoire sont : le capteur, le traitant et l'analyseur. Le capteur fait la prise de mesure des compteurs de performances directement du noyau du système à travers la bibliothèque d'accès aux compteurs. Le traitant reçoit des informations brutes venues des compteurs et il les transforme en données, tel que le débit du bus mémoire. Enfin, l'analyseur filtre les données via des méthodes statistiques avant d'envoyer l'état du bus mémoire à l'ordonnanceur.*

et l'intervalle de temps des mesures, afin de faire le calcul du débit du bus mémoire par seconde. Il faut dire que, dans certaines architectures, telles que celles de la famille *Pentium 6*, le système utilise plus d'un événement.

Enfin, l'objectif du moniteur mémoire est de transmettre à l'ordonnanceur la charge du bus mémoire dans la période précédente. Donc, le dernier module fait l'analyse statistique, c'est-à-dire filtre les données et élimine les mesures bruitées. Ce module, ainsi que le module traitant, peuvent être facilement modifiés. Le cycle d'exécution entre ces trois modules se répète à chaque période d'une centaine de millisecondes pendant toute l'exécution du système. La définition de la taille de la période doit prendre en compte la granularité de l'observation des applications (gros grain) et le coût de la prise de mesure (plus petit que 10 microsecondes sur les machines testées).

5.3.3 L'ordonnanceur

Au sein de cette section, nous détaillons le fonctionnement du composant central du système DRAC : l'ordonnanceur. Ce dernier se situe aussi au niveau utilisateur, il communique avec le moniteur mémoire et le gestionnaire des tâches. En fait, c'est lui qui contrôle l'exécution des processus soumis à DRAC, sans même établir de liens directs avec l'ordonnanceur du système d'exploitation. Son but est d'appliquer la politique d'ordonnancement qui vise simplement à minimiser le temps durant lequel le bus mémoire est saturé et, par conséquent, à augmenter le rendement de la machine.

Pour cela, l'ordonnanceur utilise les résultats décrits dans le chapitre 4 pour estimer les performances des applications en fonction de l'activité du bus mémoire. Les débits mémoire transmis par le moniteur mémoire permettent à l'ordonnanceur de classer ses processus selon le niveau de charge mémoire. Sachant que l'ordonnanceur DRAC maintient un processus par processeur, il est donc possible de connaître la charge induite sur la machine.

Par exemple, nous présentons dans le tableau 5.1 l'exemple d'un biprocesseur. L'exécution de deux processus classés dans le niveau *fort* provoque forcément la saturation du bus mémoire. Au contraire, l'exécution des deux processus classés dans le niveau *faible* provoque la sous-utilisation du bus mémoire. L'ordonnanceur cherche à maximiser l'utilisation du bus mémoire sans le saturer, cela revient à exécuter en concurrence un processus classé *fort* et un processus classé *faible*. La machine se trouve alors équilibrée du point de vue de ces ressources de calcul et de sa hiérarchie mémoire.

Niveau de charge mémoire avec 2 processus	Charge Induite sur un biprocesseur		
	sous utilisation	équilibrée	saturée
fort + fort			X
faible + faible	X		
faible + fort		X	

TAB. 5.1 *Nous présentons les charges induites d'une machine biprocesseur. La politique de l'ordonnanceur DRAC équilibre la charge mémoire, donc, les situations recherchées sont celles qui contiennent un processus faible et l'autre fort.*

5.3.3.1 L'algorithme de contrôle

Afin d'obtenir l'équilibre d'utilisation de la hiérarchie mémoire et des processeurs, l'algorithme de contrôle utilise deux limites pour le niveau d'activité sur le bus mémoire. Une limite inférieure, qui caractérise la sous-utilisation du bus mémoire, et une limite supérieure caractérisant la saturation de ce même bus.

Nous avons vu précédemment qu'un processus peut être classé selon son débit mé-

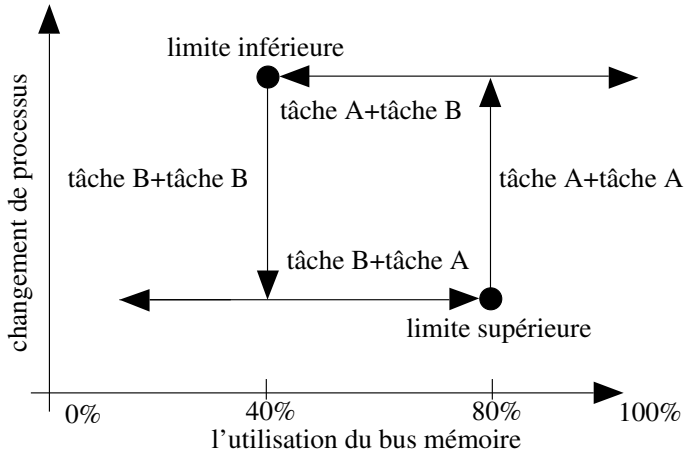

Figure 5.6 *L'algorithme de contrôle suit le principe de contrôle par hystérésis. Ce principe est basé sur la définition de limites. Tant que l'utilisation du bus mémoire est entre 40% et 80% l'ordonnanceur n'est pas appelé. Dès que l'utilisation se trouve à un niveau de sous-utilisation, ou à un niveau saturé, un nouvel ordonnancement est appliqué.*

moire, qui représente donc sa charge sur le bus mémoire. La charge sur le bus de la machine est la somme de l'ensemble des charges des processus en cours d'exécution dans le système. Par exemple, dans le tableau 5.1, les couples d'exécutions qui maintiennent équilibrée la charge du bus mémoire de la machine sont ceux qui ont un processus *faible* en concurrence avec un *fort*. Malheureusement, cette situation dépend de la charge des processus disponibles dans le système et de l'évolution de leurs charges.

Le mécanisme de contrôle de l'ordonnanceur DRAC utilise le principe d'hystérésis afin d'augmenter la robustesse de la prise de décisions. Prenons encore le cas du biprocesseur qui exécute des tâches d'un type A et B où A sont des processus avec un niveau de charge *fort* et B les processus avec un niveau de charge *faible*. La limite inférieure est fixée à 40% de l'utilisation du bus mémoire et la limite supérieure à 80%.

Le mécanisme (figure 5.6) considère que le système est équilibré si la charge de la machine est entre les deux limites. Aussitôt que la charge passe à un niveau sous-utilisé, ou saturé, un nouvel ordonnancement est appliqué. L'utilisation des deux limites rend la prise de décision plus robuste en évitant des changements d'ordonnancement trop fréquent.

Finalement, nous présentons dans la figure 5.7 un diagramme d'activité de l'ordonnanceur DRAC. La première chose à connaître est le nombre de processus en cours d'exécution. Si ce nombre dépasse le nombre de processeurs, l'ordonnanceur suspend un processus DRAC via le gestionnaire de tâches. Si ce nombre est inférieur au nombre de processeurs, l'ordonnanceur choisit un processus suspendu et le met en exécution.

Enfin, quand le nombre de processus en cours d'exécution est égal au nombre de processeurs, l'ordonnanceur déclenche la procédure de contrôle suivant l'utilisation du bus

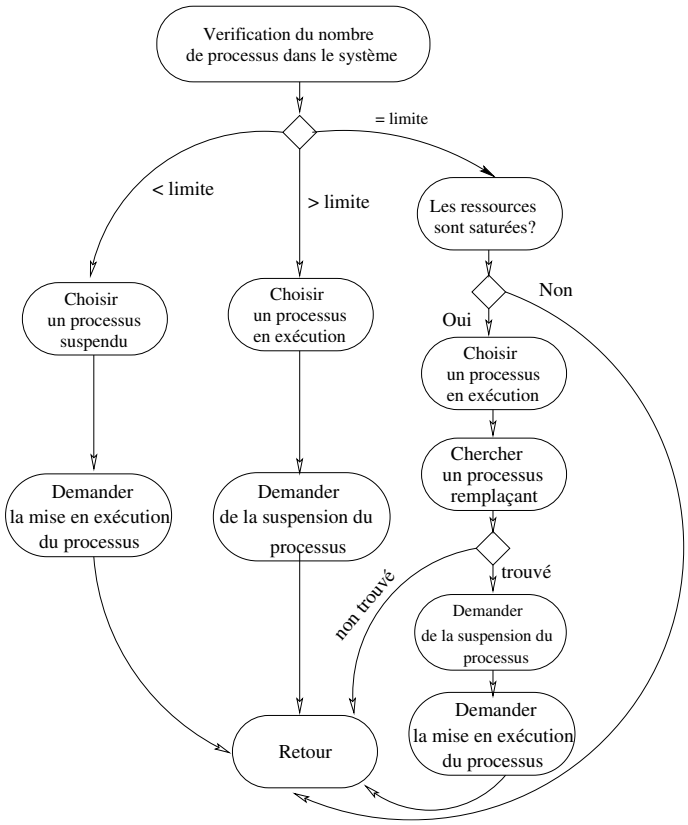

Figure 5.7 *L'ordonnanceur DRAC a deux objectifs : contrôler le nombre de processus en cours d'exécution dans le système et tenter d'éviter la saturation du bus mémoire. Le nombre de processus DRAC en cours d'exécution doit être égal au nombre de processeurs. Selon cette limite l'ordonnanceur suspend ou met en exécution des processus. Puis, selon la préscence d'une saturation ou une sous-utilisation du bus, l'ordonnanceur effectue un remplacement de processus.*

mémoire. L'algorithme de contrôle indique s'il faut effectuer un nouvel ordonnancement suivant le niveau de charge du bus, un processus *en cours d'exécution* est remplacé par un autre parmi les processus *suspendus*.

L'ordonnanceur fait la demande de remplacement d'un seul processus à la fois. Même si toutes les décisions sont prises par l'ordonnanceur, il ne réalise pas vraiment les changements de processus : c'est le gestionnaire de tâches qui effectue les ordres de l'ordonnanceur.

5.3.4 Gestionnaire de tâches

Ce module est responsable de la gestion effective des processus du système DRAC soit : la soumission, le retrait, le démarrage et la suspension des processus ainsi que le

contrôle de la synchronisation.

5.3.4.1 DRACsub : gestionnaire de soumission de processus

Le gestionnaire de soumission de processus, aussi appelé DRACSub, applique le contrôle d'admission aux processus dans le système. Ce module s'exécute dans un processus, séparément de l'ensemble du reste du système. Il reste la plupart du temps dans une boucle d'attente de soumission (figure 5.8).

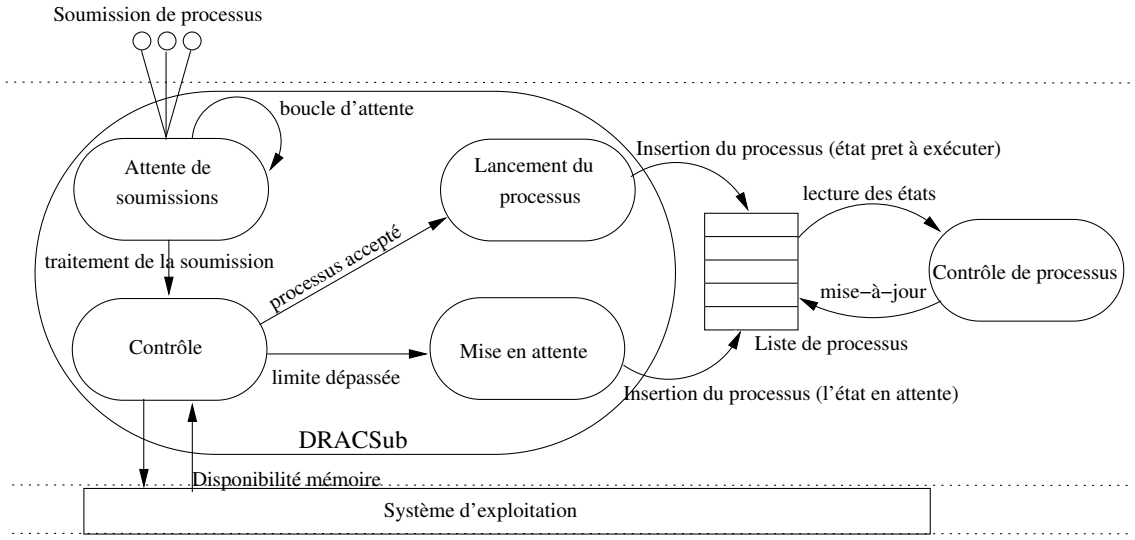

Figure 5.8 *Un des rôles du gestionnaire de tâches est le contrôle d'admission. Son objectif est de limiter le nombre de processus dans le système selon la disponibilité de la mémoire principale. Lorsqu'il reçoit une soumission il vérifie la quantité de mémoire disponible. Ensuite, il met le processus soit* en attente *soit dans l'état* prêt à s'exécuter.

Lorsqu'un processus est lancé il demande au système d'exploitation l'allocation d'un espace mémoire. Ce processus peut faire des nouvelles demandes de mémoire ou même les libérer durant son exécution. Cependant, la mémoire allouée à un processus reste indisponible indépendemment de son état. Un processus DRAC dans l'état *en attente* n'est pas encore connu par le système d'exploitation donc il n'a pas de mémoire allouée. Ainsi, le système DRAC peut accepter les soumissions indépendamment de la quantité de mémoire disponible, néanmoins, il doit contrôler le nombre de processus lancé dans le système, en effet il ne peut pas y avoir trop de processus *prêt à s'exécuter*.

Lorsqu'une nouvelle soumission arrive le module déclenche alors la procédure de contrôle d'admission (figure 5.8) qui vérifie la quantité de la mémoire principale disponible. Le nombre de processus DRAC en cours d'exécution est environ toujours le nombre de processeurs. Les processus suspendus gardent leurs allocations mémoire.

Le dépassement de l'allocation mémoire provoque l'utilisation de l'espace temporaire sur disque (*swap*) induisant potentiellement de forte baisse de performance. Afin de minimiser l'utilisation du disque, le système DRAC limite le nombre des processus lancés dans le système selon la disponibilité mémoire. Ces limites vont de 50% de la taille de la mémoire principale à 200%, configurables par l'administrateur. Cependant, il est recommandé que la limite acceptée ne dépasse pas la taille mémoire, soit 100%. Finalement, le processus qui vient d'être soumis est mis en attente si la limite est dépassée, dans le cas contraire, il est simplement lancé.

5.3.4.2 Contrôle de processus : retrait et synchronisation

Dans la section précédente nous avons présenté le traitement des soumissions des processus. Nous avons montré que l'ensemble des processus soumis à DRAC sont lancés via DRACsub, cela fait de DRAC leur processus père du point de vue du système d'exploitation. Nous avons donc utilisé la notification de fin d'exécution d'un processus fils à son père afin d'identifier sa terminaison et son retrait du système. L'opération de retrait de processus consiste simplement à effacer les données concernant un processus de la liste des processus.

L'objectif principal du gestionnaire de processus est de permettre la modification de l'exécution des processus par l'ordonnanceur DRAC. Grâce à ce gestionnaire, l'ordonnanceur peut connaître le nombre de processus, l'historique de leurs charges et leurs états. C'est l'ordonnanceur qui décide l'ordre d'exécution qui est appliqué par le gestionnaire de tâches, par contre, c'est le gestionnaire qui doit prendre en compte les dépendances (synchronisation) entre les processus.

Comme a été indiqué dans la section 5.2 nous traitons principalement deux cas de synchronisations : les méthodes d'*exclusion mutuelle* et les *barrières*. Pour chaque cas le traitement effectué par le système est différent. Pour l'*exclusion mutuelle* il s'agit de bloquer la suspension des processus dans une section critique, pour les *barrières* simples il s'agit de mettre en attente tous les processus jusqu'à l'arrivée du dernier processus. Dans la suite la section, nous détaillons les solutions proposées pour les deux situations

Exclusion mutuelle **:** Afin de garantir qu'un processus dans une région critique ne soit pas suspendu par l'ordonnanceur, il suffit qu'un processus inhibe l'action du système à l'entrée d'une région et le réactive à sa sortie (figure 5.9). La proposition retenue est celle de l'instrumentation des fonctions des bibliothèques de parallélisme responsables par l'entrée et la sortie d'une région critique. Le système DRAC est alors inhibé avant le verrouillage de la région critique et réactivé après le déverrouillage. Le contrôle de condition de courses n'est pas modifié, c'est-à-dire qu'il continue à être traiter entièrement par les fonctions de la bibliothèque utilisée et leur bon emploi dans l'application. L'inhibition est déclenché par le test de la présence de drapeaux positionnés lors de l'entrée dans les fonctions de verrouillage. Ces drapeaux sont levés lors du passage dans les fonctions

de déverrouillage correspondantes. Si un drapeau a été positionné, le système inhibe le contrôle d'exécution pour l'ensemble du groupe d'exécution.

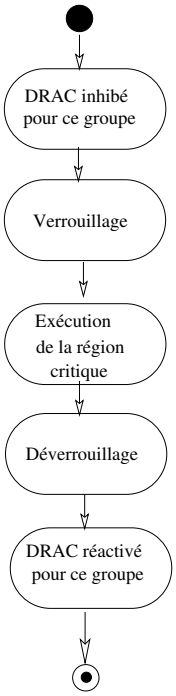

Figure 5.9 *L'action du système DRAC est inhibé à l'entrée d'une région critique et réactivé à la sortie.*

***Barrière* :** Comme indiqué précédemment, dans le cas général des synchronisations de type *barrière*, il s'agit d'éviter que l'ordonnanceur DRAC endorme un processus qui serait attendu pour libérer une *barrière*.

Nous présentons dans la figure 5.10 le traitement simplifié réalisé par le système pour une *barrière* comme celle que l'on peut trouver dans les applications utilisant les directives OpenMP. Ce traitement intervient via une instrumentation de la bibliothèque autour de la fonction *barrière*. Le traitement est exécuté par chaque processus. La première étape consiste à mesurer le temps écoulé depuis le dernier passage sur une *barrière*. Cette durée est ensuite comparée à une valeur au-dessous de laquelle il est préférable d'inhiber le processus de contrôle. En effet si les *barrières* successives sont trop rapprochées le surcoût de gestion serait trop important. Dans le cas contraire, et si le système de contrôle à décider de diminuer le nombre de processus actifs, un nouveau processus *suspendu* est activé et le processus *en cours d'exécution* se suspend. Si le processus était le dernier à arriver à la *barrière* il réactive tous les processus du groupe afin de leur permettre de traverser la *barrière*. A la sortie de la *barrière* on teste de nouveau si le système de contrôle à réduit le nombre de processus activés pour ce groupe. Si c'est le cas seul les premiers processus à concurrence du nombre fixé par le système de contrôle resteront activés, les autres se

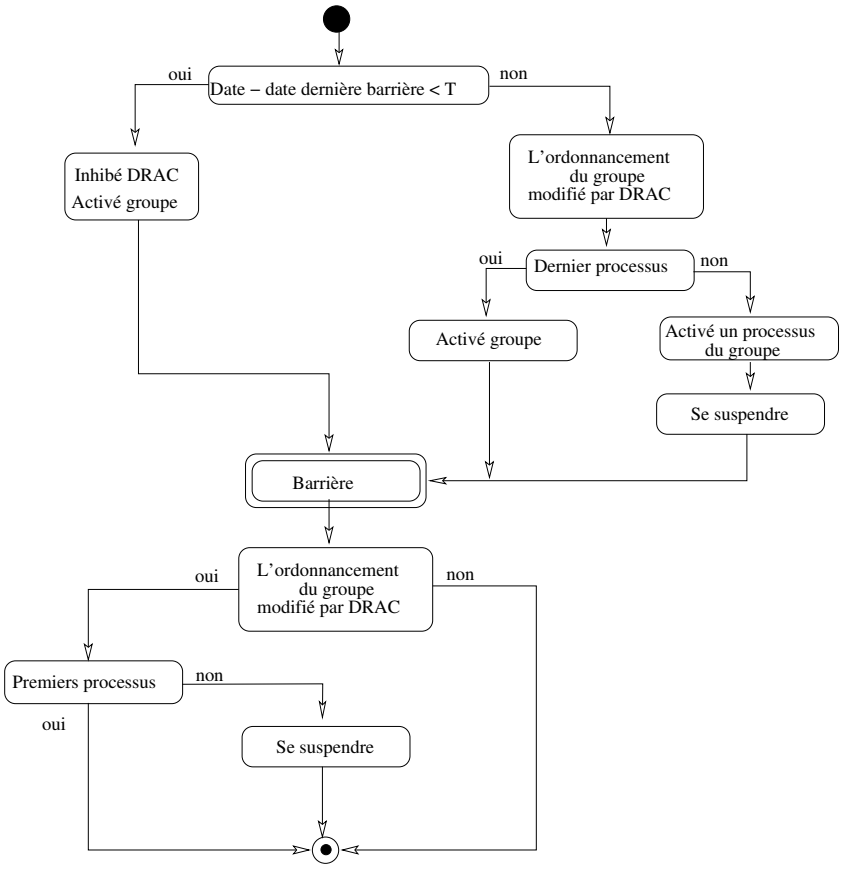

Figure 5.10 *Les processus qui appartiennent à un groupe dont l'ordonnancement a été modifié par DRAC sont suspendus à l'arrivée de la* barrière*, excepté le dernier processus qui libère l'ensemble. Si le premier processus appartient à un groupe dont l'ordonnancement n'est pas modifié par DRAC il l'inhibe jusqu'à la libération de la* barrière*. La réactivation est faite par le dernier processus arrivant à la* barrière*.*

suspendront aussitôt la *barrière* franchie.

La figure 5.11 présente un exemple de chronogramme d'exécution sur une machine biprocesseur. Il faut noter que pour traverser la *barrière* l'ensemble des processus sont activées. Au temps $T0$, nous avons un processus A et un processus B qui appartiennent à un même groupe de processus et s'exécutent en concurrence. Au temps $T1$, une saturation sur le bus mémoire est détectée, l'ordonnanceur demande au gestionnaire de tâches le remplacement du processus B par le processus C. Au temps $T2$, le processus A arrive à la *barrière* et est remplacé par son voisin, le processus B. À l'arrivée du processus B à la *barrière*, au temps $T3$, le processus A est activé. Durant la traversé de cette *barrière*, le système contient 3 processus en cours d'exécution. Puis, le système suspend le processus B au temps $T4$. Finalement, les processus A et B finissent leur exécutions l'un après l'autre.

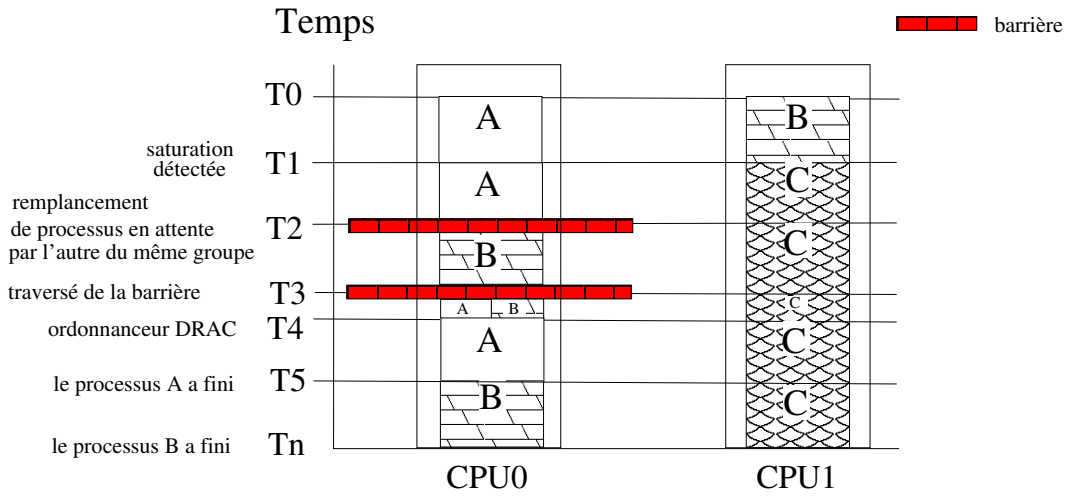

Figure 5.11 *Le scénario présente l'exécution des processus avec synchronisation (*barrière*) sur un biprocesseur. Une saturation sur le bus mémoire est détectée, et donc l'ordonnanceur DRAC intervient(Temps $T1$) en suspendant le processus B. À l'arrivée du processus A à la* barrière*, le système de contrôle remplace ce processus par son voisin, le processus B et, à la arrivée du processus B à la* barrière*, il libère l'ensemble afin de traverser la* barrière *de la bibliothèque.*

De cette façon, le gestionnaire est capable de contrôler l'ensemble des processus en supportant les fonctions d'insertion, de retrait ainsi que des synchronisations. Le traitement de l'insertion et du retrait d'un processus est faite par l'interception de leur soumission et son signal de terminaison. La synchronisation est alors faite par l'inhibition et la réactivation du système pour l'attente passive et pour les groupes de processus qui DRAC n'est pas modifié. Le système est normalement réactivé à la sortie d'une région critique ou l'arrivée du dernier processus au point de rendez vous, excepté, dans le cas des applications qui font des synchronisations trop fréquents. Dans ce cas, c'est encore le gestionnaire de tâches qui contrôle la période minimum de réactivation du système. Il vérifie donc le temps de la dernière demande d'inhibition avant réactivé le système.

5.4 Le Prototype DRAC

Nous consacrons cette section à la description du prototype DRAC qui a comme but de valider l'idée générale de l'architecture. La première version de ce prototype est restreinte aux architectures processeurs Intel $32bits$ plus précisément à celles qui appartiennent à la famille des processeurs *Pentium 6* et *Pentium 4*. Au niveau logiciel, la base de développement disponible et supportée est composée du système d'exploitation Linux, actuellement le noyau 2.4, et de la bibliothèque *pthread* de *Linuxthreads*. La bibliothèque d'accès aux compteurs de performance utilisée est celle développée par Mikael Patters-

son (Perfctr [90]), les compilateurs OpenMP supportés sont Omni [88] (version 1.5) et PGI [53] (version 4.3).

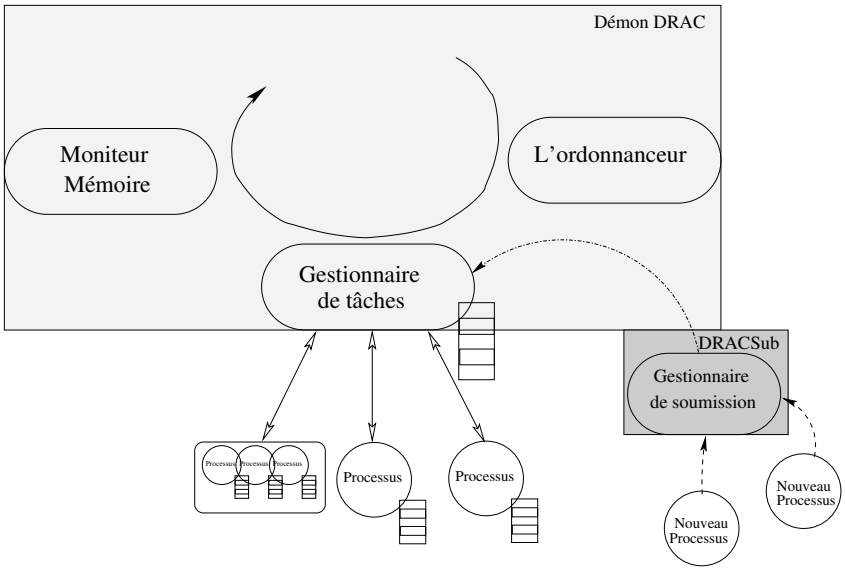

Figure 5.12 *Le prototype DRAC est composé uniquement par deux processus : le démon DRAC et DRACsub. Le démon DRAC est responsable des modules moniteur mémoire, ordonnanceur et gestionnaire de processus. DRACsub fait simplement la gestion des soumissions. Enfin, le démon DRAC reste toujours une boucle sans fin qui exécute ces modules l'un après l'autre et DRACsub est en attente des nouvelles soumissions.*

Ce prototype est composé de quatre modules (voir figure 5.12) : le *moniteur mémoire*, l'*ordonnanceur*, le *gestionnaire de tâches* et le *gestionnaire de soumission*. L'exécution de ces modules est répartie sur deux processus : le démon DRAC et DRACsub. Le démon DRAC est celui qui contrôle l'exécution des modules *moniteur mémoire*, *ordonnanceur* ainsi que *gestionnaire de tâches*. Il reste donc le gestionnaire de soumission à DRACsub. Toutes les informations concernant l'ensemble des processus soumis au système DRAC sont sauvegardées dans une liste chaînée. Cette liste est accédé, uniquement, par le gestionnaire de tâches. Les variables d'une entrée de la liste de processus sont :

- **key** : clé attribuée par DRAC à l'enregistrement du processus et qui ne change jamais.
- **mypid** : numéro d'identification du processus attribué par le système d'exploitation.
- **charge** : charge mémoire du processus observée pendant la dernière période.
- **status** : état courant du processus qui peut être : en attente, suspendu, en cours d'exécution et prêt à exécuter (voir dans la section 5.3.1).
- **group** : drapeau d'appartenance à un groupe.

Parmi ces variables, nous en avons une qui représente l'état d'un processus vue par DRAC. Cet état est parallèle à celui vue par le système d'exploitation (plus de détaille

1 Nœud de la liste de processus

```
1: typedef struct process {
2:         int key ;
3:         pid_t mypid ;
4:         int charge ;
5:         int status ;
6:         int group ;
7:         struct process *next ;
8:         struct process *prev ;
9: } node ;
```

section 5.3.1). Le changement d'état d'un processus vue par le système d'exploitation n'implique pas la mise-à-jour automatiquement de celui vue par DRAC. Il vérifie donc la cohérence de ces informations avant chaque prise de décision.

Finalement, le processus démon *DRAC* a un segment de mémoire partagée avec chaque groupe de processus. Le segment conserve la clé, un drapeau et l'état de chaque processus du groupe. Les processus appartenant à un groupe peuvent, en arrivant à une *barrière*, lancer un autre processus ce groupe et ensuite se suspendre. Ce remplacement est signalé uniquement par le changement d'état des processus du groupe sur le segment mémoire partagée. C'est donc, grâce à ce segment, que le *Démon DRAC* fait la mise-à-jour de la liste des processus. Ce mécanisme de bascule vise à accélérer le remplacement des processus. Enfin, le drapeau permet à chaque processus d'inhiber l'action du système DRAC sur soit même, quand nécessaire. Le système ne peut intervenir que si l'ensemble des drapeaux des processus du groupe sont *verts*. Donc, avant l'ordonnancement, le système vérifie les drapeaux des processus, s'il trouve un drapeau *rouge* il libère l'ensemble des processus de ce groupe.

Démon DRAC : Le premier composant du démon DRAC est le moniteur mémoire qui est situé dans le niveau le plus bas de l'architecture du système. Il fait essentiellement le recueil des compteurs matériels et l'analyse des données. L'accès aux compteurs est fait via la bibliothèque Perfctr, laquelle supporte les architectures des processeurs de famille Intel *Pentium 6* et *Pentium 4*.

Même s'il utilise une seule bibliothèque d'accès aux compteurs il dispose des mécanismes spécifiques à chacune des deux architectures. Le format des registres n'est pas pareil et les événements à observer sont différents. Nous avons l'événement qui compte le nombre des transactions sur le bus mémoire, pour le famille Intel *Pentium 6* et l'événement qui compte le nombre de cycles sur lesquels il y a des transferts sur le bus mémoire, sur la famille Intel *Pentium 4*.

Au démarrage du système avant la mise en route du moniteur mémoire, le démon DRAC doit identifier l'architecture processeur de la machine. Une fois que l'événement

matériel est proprement configuré, le moniteur mémoire doit identifier sur quel processeur les processus sont en train de s'exécuter afin de pouvoir calculer le débit par processus.

Le calcul du débit par processus est possible par le fait d'exécuter uniquement un processus par processeurs. Le moniteur observe donc les activités effectuées par un processeur dans une période et il peut connaître le processus qui s'exécutait à ce moment là sur ce processeur.

À son tour, l'ordonnanceur reçoit les débits mémoire observés par le moniteur mémoire et le contrôle de décision via l'algorithme de contrôle de décision 2. Il compare le débit du bus mémoire de la machine aux limites inférieure et supérieure définies ultérieurement. Lorsque le débit actuel est plus grand que la limite supérieure ou plus petit que la limite inférieure, l'ordonnanceur demande le remplacement des processus en cours d'exécution, dans le cas contraire, il maintient ces mêmes processus jusqu'à la fin de la prochaine période.

2 Contrôle de décision

```
1:  si (débit_mémoire > limite_supérieure) alors
2:      le bus mémoire est saturé.
3:      chercher un processus en cours d'exécution qui ait une charge d'utilisation mémoire élevée.
4:      chercher un processus suspendu avec une charge d'utilisation mémoire faible (basé sur son historique d'exécution).
5:      remplacer le processus en cours d'exécution par le processus suspendu.
6:  fin si
7:  si (débit_mémoire < limite_inférieur) alors
8:      le bus mémoire est sous-utilisé.
9:      chercher un processus en cours d'exécution qui ait une charge d'utilisation mémoire faible.
10:     chercher un processus suspendu avec une charge d'utilisation mémoire élevée (basé sur son historique d'exécution).
11:     remplacer le processus en cours d'exécution par le processus suspendu.
12: fin si
```

Enfin, le dernier composant du démon DRAC est le gestionnaire de tâches. Il est responsable de la suspension et du réveil des processus. Son fonctionnement est basé sur la bibliothèque de signal disponible dans les systèmes UNIX qui permet de suspendre un processus via le signal *SIGSTOP* et de réveiller via le signal *SIGCONT*. C'est lui aussi qui contrôle la synchronisation des processus.

DRACsub : Le gestionnaire de soumission, DRACsub, reçoit les demandes d'exécution et lance les processus. Il doit simplement contrôler la disponibilité de la mémoire de la machine afin de limiter le nombre de processus lancés dans le système. À l'arrivée d'une nouvelle soumission il se connecte au démon DRAC et enregistre le nouveau pro-

cessus dans la liste. Le fonctionnement du gestionnaire de tâches et de DRACsub sont basés au traitement des requêtes de soumission, de retrait et de synchronisation.

Les processus sont soumis au système DRAC via l'outil DRACsub qui reçoit le chemin et le nom du code exécutable de l'application ainsi que leurs paramètres. C'est donc de cette façon que le système obtient les informations de départ. C'est le gestionnaire qui fait le lancement de l'application, donc l'ensemble de processus est vu par le système d'exploitation comme étant ses fils. De cette façon, à la fin de l'exécution de chacun des processus, le gestionnaire peut récupérer son signal de terminaison (*SIGCHLD*) et fait le retrait du processus.

Pour les fonctions de synchronisations, nous avons alors traité, dans ce prototype, nous avons traité un certain nombre de cas afin de démontrer la faisabilité et les performance du système. Dans le cas des application utilisant les directives de compilation OpenMP nous avons considéré les compilateurs Omni [88] (version 1.5) et PGI [53] (version 4.3). Nous avons aussi traité une sous-partie des fonctions de la bibliothèque pthrea Linuxthreads. L'identification des fonctions est réalisé par l'instrumentation du code des bibliothèques.

Nous présentons, dans le tableau 5.2, quelques exemples de fonctions des bibliothèques parallèles OpenMP et *pthread* qui peuvent être contrôle selon les comportements de base décris dans la section 5.3.4.2.

	OpenMP Omni	pthread Linuxthreads
exclusion mutuelle	ompc_lock ompc_unlock ompc_atomic_lock ompc_atomic_unlock ompc_enter_critical ompc_exit_critical	pthread_mutex_lock pthread_mutex_trylock pthread_mutex_unlock pthread_rwlock_rdlock pthread_rwlock_unlock pthread_rwlock_wrlock sem_wait (sémaphore) sem_post (sémaphore)
barrière	ompc_in_parallel ompc_finalize ompc_terminate ompc_thread_barrier	pthread_exit pthread_join pthread_barrier_wait

TAB. 5.2 *Exemple des fonctions de synchronisation les plus courante dans les bibliothèques parallèles.*

Le compilateur Omni est basé sur une licence type GPL (*General Public License*) dont le code source est disponible. Les deux fonctions de la *barrière* sont *_ompc_thread_barrier* et *_ompc_thread_barrier2* qui utilisent une approche par attente active. Nous avons modifié les fonctions d'entrée dans une région d'*exclusion mutuelle* (*_ompc_critical_init*) et de sortie (*_ompc_critical_destroy*) afin d'inhiber et de réactiver l'ordonnanceur. Enfin, la dernière fonction, qui a été changée, est celle de création des processus *omp_in_parallel*

OpenMP qui, dans DRAC, ajoute les nouveaux processus.

Pour le compilateur PGI, nous avons changé le code objet de la bibliothèque *libpgmp.a* dont le code source est indisponible. Nous avons donc modifié la fonction *_mp_ncpus* qui nous permet de récupérer les identificateurs des processus et la fonction *_mp_barrier* qui nous permet de contrôler la *barrière*. Les fonctions d'entrée et de sortie d'une région critique n'ont pas encore été traitées pour cette bibliothèque, mais pourrait l'être.

Pour conclure, le prototype fonctionne, par contre il possède des restrictions pour l'exécution des applications parallèles avec OpenMP puisque nous n'avons pas encore instrumenté ni testé l'ensemble de fonctions de synchronisation. Il ne support aucune application parallélisée avec les directives de la bibliothèque *pthreads*, mais l'instrumentation de l'ensemble des fonctions de bases ne semble pas être complexe.

5.5 Bilan

Dans ce chapitre, nous avons présenté l'architecture et le prototype du système DRAC. Ce travail propose un nouveau système de contrôle de processus au niveau utilisateur. Le point de vue employé est celui de l'administrateur qui souhaite maximiser le rendement de la machine.

Le principe du système fait en sorte que la ressource bus mémoire ne soit pas saturée ni sous-utilisée en adaptant simplement l'exécution des processus selon la charge d'utilisation du bus mémoire. Le système est plus adapté aux applications de longue durée d'exécution, notamment les applications scientifiques. Dans ce contexte, l'utilisation des bibliothèques parallèles est courante, et donc, il est essentiel quelle soient traitées par le système DRAC.

Le système contrôle l'utilisation des ressources via la suspension et la réactivation des processus suivant la politique de décision employée. La méthode d'estimation de l'accélération des applications en cours d'exécution à travers l'observation du bus mémoire (chapitre 4), permet à l'ordonnanceur de faire la prise de décision en fonction des performances estimées. Le système dispose d'un support des applications avec des fonctions de synchronisation. Ce support nécessite l'instrumentation des bibliothèques et des fonctions de synchronisation.

Finalement, un prototype du système a été développé afin de valider l'architecture. Nous présentons dans le chapitre suivant une modélisation de l'ensemble du système ainsi que l'évaluation du prototype.

6

Modélisation et évaluation

Comme nous l'avons présenté précédemment, la base de ce travail est l'étude du problème de la contention mémoire sur les architectures multiprocesseurs à mémoire partagée. Nous proposons un système de contrôle qui utilise l'estimation du rendement en cours d'exécution pour limiter ce problème.

Pour approfondir ce travail et avant l'évalution du prototype du système, nous proposons une modélisation simplifiée de l'ensemble du système. Cette modélisation permet de décrire les régimes de fonctionnement de l'ensemble du système. Ces modélisations sont fondées sur des études de cas pour des machines avec p processeurs. Nous présentons les résultats pour $p = 2$ et $p = 4$ et concluons sur la généralisation du problème.

Enfin, la dernière partie du chapitre est consacrée à la comparaison des résultats données par les modèles avec ceux obtenus avec le prototype du système DRAC, pour les cas où $p = 2$ et $p = 4$.

6.1 Modélisation du système

Le modèle proposé permet de calculer le temps d'exécution d'un ensemble de processus et d'évaluer l'impact que l'utilisation de la hiérarchie mémoire des multiprocesseurs peut provoquer sur leurs performances. Dans la première partie de cette section, nous décrivons les hypothèses du modèle et nous posons nos premières définitions, notamment les formules représentant l'utilisation mémoire d'un processus ou de l'ensemble de la machine. Ensuite, nous présentons trois types de modèle d'ordonnancement, dont l'objectif est de calculer le temps d'exécution moyen d'un groupe de processus ainsi que les limites

supérieure et inférieure.

6.1.1 L'impact de l'utilisation de la hiérarchie mémoire sur la performance

Dans le chapitre 2 nous avons montré que sur les architectures multiprocesseurs à mémoire partagée, l'ensemble des processeurs accèdent aux données de la mémoire principale via un réseau d'interconnexion commun. Ainsi, des applications avec de forts besoins en mémoire peuvent provoquer la saturation du réseau. Cela implique la chute des performances et la baisse du rendement de la machine. Nous allons donc modéliser l'utilisation de la hiérarchie mémoire pour ces machines afin d'évaluer son impact sur les performances. Nous nous limiterons au cas des réseaux d'interconnexion non complexe afin de simplifier le modèle.

Hypothèses :

1. Le modèle mémoire est simple : un bus mémoire est partagé par tous les processeurs.
2. Le nombre de processus en cours d'exécution est égal au nombre de processeurs.
3. Les processus en cours d'exécution ont une seule dépendance commune, l'accès au bus mémoire.
4. Les processus ne font pas d'échanges entre le disque et la mémoire principale.
5. Les flux d'instructions exécutées par les processus sont identiques, que l'exécution soit séquentielle ou concurrente.
6. Le débit d'accès mémoire des processus est constant par morceaux.

Définitions : On note n le nombre total de processus à exécuter, et $P_0, P_1, \ldots P_{n-1}$ les processus qui s'exécutent sur p processeurs, accédant en concurrence à un bus mémoire unique. t_i est le temps d'exécution séquentielle du processus i. $d(i)$ est le débit mémoire demandé par un processus i où $i \in [0 \ldots n-1]$. On note d_{max} est le débit maximal supporté par le bus mémoire.

Nous définissons la *charge* $\alpha(i)$ d'un processus P_i ainsi :

$$\alpha(i) \stackrel{\text{def}}{=} \frac{d(i)}{d_{max}}. \tag{6.1}$$

α représente la fraction du débit mémoire utilisé en séquentiel par P_i quand il est seul sur la machine. Ainsi, si ce processus sature la hiérarchie mémoire $\alpha(i) = 1$, et s'il ne l'utilise pas du tout, $\alpha(i) = 0$. L'ordonnancement des processus sur les multiprocesseurs avec des temps d'exécution différents est décrit par Garey et Johnson [51] (problème $SS8$) comme

$NP - Complet$. Notre problème ajoute simplement la notion de débit mémoire alors le cas général est aussi $NP-Complet$. Nous allons décrire une sous-catégorie du problème sur lequel l'ensemble de processus qui ont le même temps d'exécution.

La *charge théorique* du bus est définie comme étant la somme des charges des processus en exécution : $\frac{\sum_{i=0}^{n-1} d(i)}{d_{max}}$. La charge théorique peut être plus grande que 1 si la somme des débits mémoire est plus grande que le débit maximal supporté par le bus mémoire. On définit alors la *charge réelle* de la machine : $min(1, \frac{\sum_{i=0}^{n-1} d(i)}{d_{max}})$, elle est égale à la charge théorique sauf si celle-ci dépasse 1, auquel cas elle vaut 1.

Sachant que les processus s'exécutent par groupe de p processus à la fois, nous avons alors le *temps total d'exécution* d'un ensemble de processus défini comme étant :

$$T_{total} \stackrel{\text{def}}{=} \frac{t_{seq}}{p} \times \sum_{j=0}^{ng} max(1, \frac{\sum_{i=0}^{p-1} d(i)}{d_{max}}). \tag{6.2}$$

où le nombre de groupes est ng et le temps séquentiel t_{seq} est le temps pris par le processeur pour exécuter les n processus les uns après les autres ($t_{seq} = \sum_{i=1}^{n} t_i$).

Si le bus mémoire n'est pas saturé ($\sum_{i=1}^{n} d_{seq}(i) < d_{max}$) :

$$T_{total} = \frac{t_{seq}}{p}. \tag{6.3}$$

En se basant sur cette définition de la charge d'un processus, on définit $P^{(k)}$ comme étant un processus quelconque avec une charge $\alpha(k) \in [0, 1]$. On note n_k le nombre de processus du type $P^{(k)}$ (avec bien sûr $\sum n_k = n$) et $\beta_k \stackrel{\text{def}}{=} \frac{n_k}{n}$ la fraction de processus du type $P^{(k)}$ par rapport à l'ensemble de tous les processus. On note $P_i^{(k)}, i = 0 \dots n_k - 1$ les n_k processus du type $P^{(k)}$ et $t_i^{(k)}$ le temps d'exécution séquentielle de $P_i^{(k)}$.

Lorsque deux processus $P_i^{(k)}$ et $P_j^{(k')}$ ($i, j \in \{0, \dots, n-1\}$) s'exécutent en concurrence, on considère qu'ils s'accélèrent d'un facteur $\delta^{k,k'}$ et $\delta^{k',k}$ respectivement par rapport à leur temps d'exécution séquentielle (t_i et t_j respectivement). On considère que les facteurs $\delta^{k,k'}$ et $\delta^{k',k}$ ne dépendent que de la charge de chacun des deux processus.

Enfin on définit le temps séquentiel d'exécution de l'ensemble de processus $P^{(k)}$ comme valant :

$$t_{seq}^{(k)} \stackrel{\text{def}}{=} \sum_{i=0}^{n_k} t_i^{(k)}. \tag{6.4}$$

Finalement, le temps total d'exécution de cette ensemble de processus ($P^{(k)}$) est $T_{total}^{(k)} \stackrel{\text{def}}{=} T_{total}$, où t_{seq} est remplacé par $t_{seq}^{(k)}$.

6.1.2 Étude de cas à deux types de charges : lourdes et légères

Dans la section précédente nous avons présenté un modèle d'utilisation mémoire pour des multiprocesseurs avec des processus qui ont des charges mémoires $\alpha(i) \in [0, 1]$, où $\alpha(i) = 0$ représente la non utilisation du bus mémoire et $\alpha(i) = 1$ l'utilisation maximale de ce dernier. Cette section est consacrée à l'étude du cas d'exécution avec seulement deux types de processus : le premier se caractérisant par une charge lourde ($\alpha(i) \geq 0.9$) et le second avec une charge légère ($\alpha(i) \leq 0.1$). Les charges mémoires choisies sont caractéristiques du cas ou l'on rencontre une contention mémoire pour la première, et du cas ou on n'en rencontre pas pour la seconde.

Dans ce contexte, nous définissons des modèles d'ordonnancements qui permettent d'évaluer les temps d'exécution selon trois scénarios différents :

- *Meilleur ordonnancement ($T_{meilleur}$)* : il fait en sorte que la charge sur le bus mémoire soit la plus petite pendant l'exécution de l'ensemble des processus.
- *Ordonnancement moyen (T_{moyen})* : il est basé sur la probabilité d'occurrence de chaque *groupe d'exécution* possible. Un *groupe d'exécution* est représenté par $G(P_0|P_1 \dots P_{p-1})$ où p est le nombre de processeurs. Dans le cas particulier des biprocesseurs nous utilisons le terme *paire d'exécution* ($G(P_0|P_1)$) à la place de *groupe d'exécution*.
- *Pire ordonnancement (T_{pire})* : il fait en sorte que la charge sur le bus mémoire soit la plus grande pendant l'exécution de l'ensemble des processus.

Ainsi, les cas du *meilleur ordonnancement* et du *pire ordonnancement* permettent d'avoir la limite supérieure et la limite inférieure, respectivement, des temps d'exécution d'un ensemble de processus. Le modèle d'*ordonnancement moyen*, quant à lui, correspond au cas où les processus s'exécutent de manière aléatoire, le temps d'exécution de l'ensemble des processus prend alors une valeur moyenne.

6.1.2.1 Biprocesseur

Nous commençons avec le modèle pour les machines biprocesseurs. L'idée est simple, nous envisageons de calculer le temps total d'exécution (T_{total}) d'un ensemble de processus pour les trois modèles d'ordonnancement définis précédemment : T_{moyen}, T_{pire} et $T_{meilleur}$.

Le scénario est composé par deux types de processus $P^{(0)}$ et $P^{(1)}$. On note que n est toujours le nombre de processus à exécuter, et $n = n_0 + n_1$. La charge sur le bus mémoire pour un processus $P^{(0)}$ est de $\alpha(0) = [0, 0.1]$, puisque ce processus a un débit mémoire négligeable. Par contre, celle de l'autre type de processus $P^{(1)}$ est de $\alpha(1) = [0.9, 1]$, car même les besoins d'un seul processus peuvent atteindre la capacité maximum du bus mémoire.

Le calcul de l'accélération provoqué par l'exécution concurrente de l'ensemble de processus $P^{(k)}$ est $t_{seq}^{(k)}/t_{total}^{(k)}$, avec $k \leq 0.1$ ou $k \geq 0.9$. Sachant que les processus en cours d'exécution ont comme seul dépendance le bus mémoire (hypothèse 4), l'accélération de l'exécution de deux processus du type $P^{(0)}$ ensemble est $\delta^{(0)} = 2$ et celui de l'exécution de deux processus du type $P^{(1)}$ est $\delta^{(1)} = \frac{2}{max(1,2\alpha(1))}$. Enfin nous rappelons que $\beta_k \stackrel{\text{def}}{=} \frac{n_k}{n}$. Nous utiliserons β_1 pour la description des ordonnancements.

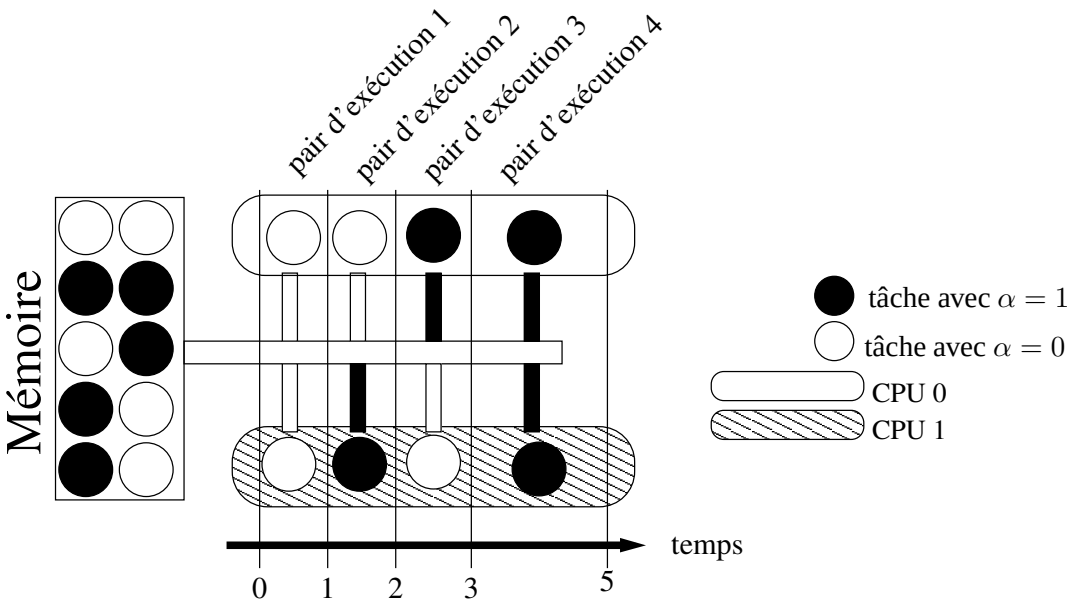

Figure 6.1 *Cette figure présente un scénario d'exécution sur une machine biprocesseur. Nous avons deux types de tâches à exécuter : les tâches noires qui ont une charge mémoire $\alpha(i) = 1$ et les tâches blanches qui ont une charge mémoire $\alpha(i) = 0$. L'ordre de tirage des tâches dépend de la politique d'ordonnancement, nous représentons donc les quatre cas d'exécution possible : d'abord une paire d'exécution* blancheXblanche*, puis deux paires semblables* noireXblanche *et* blancheXnoire*, enfin une paire* noireXnoire*. On voit que la dernière* paire d'exécution *prend du retard (2 fois le temps d'exécution de la tâche noire en séquentielle) contrairement aux trois précédentes.*

Dans cette étude du cas $p = 2$, puisque le nombre de combinaisons possibles est 2^p, nous avons 4 *paires d'exécution* possibles. Dans la figure 6.1, nous montrons ces quatre possibilités pour les biprocesseurs avec les deux types de processus $P^{(0)}$ et $P^{(1)}$. Nous définissons $t^{(0)}$ comme étant le temps d'exécution de $P^{(0)}$ et $t^{(1)}$ celui de $P^{(1)}$. Pour simplifier, nous prenons $t^{(0)} = t^{(1)} = 1$ unité de temps. Nous voyons que la première *paire d'exécution* qui est composée de deux processus du type $P^{(1)}$ dépense 2 unités de temps. Les deuxième et troisième *paires d'exécution* contiennent une occurrence de chacun des types de processus et l'exécution ne prend pas de retard, la charge réelle pourtant sur le bus mémoire est égale à 1. La dernière *paire d'exécution* est composée de deux occurrences

du type $P^{(0)}$. Elle dépense un temps d'exécution d'1 unité de temps alors qu'elle a une charge réelle sur le bus mémoire inférieure à 0.2.

Le temps total d'exécution de l'ensemble des processus est donc la somme des temps de toutes les *paires d'exécution*. Nous obtenons ainsi :

$$T_{total} = t_{G(\#P^{(0)}|\#P^{(0)})} + t_{G(\#P^{(0)}|\#P^{(1)})} + t_{G(\#P^{(1)}|\#P^{(0)})} + t_{G(\#P^{(1)}|\#P^{(1)})} \tag{6.5}$$

où $t_{G(\#P^{(k)}|\#P^{(k')})}$ représente le temps d'exécution d'une *paire d'exécution* composée d'un nombre de processus du type $P^{(k)}$ en concurrence avec des processus de $P^{(k')}$ pour $k, k' = [0|1]$.

Pour l'*ordonnancement moyen*, nous considérons le tirage aléatoire des tâches sur l'ensemble des tâches. Cela dit, nous estimons la probabilité d'occurrence de chacune des quatre combinaisons de *paires d'exécution* en fonction du pourcentage β_1 de processus du type $P^{(1)}$ par rapport l'ensemble des n processus à exécuter.

$$Prob_0(G(\#P^{(0)}|\#P^{(0)})) = (1-\beta_1)^2 \tag{6.6}$$
$$Prob_1(G(\#P^{(0)}|\#P^{(1)})) = \beta_1(1-\beta_1) \tag{6.7}$$
$$Prob_2(G(\#P^{(1)}|\#P^{(0)})) = \beta_1(1-\beta_1) \tag{6.8}$$
$$Prob_3(G(\#P^{(1)}|\#P^{(1)})) = \beta_1^2 \tag{6.9}$$

Puisque nous avons défini les quatre possibilités des *paires d'exécution*, nous pouvons conclure que :

$$Prob_0 + Prob_1 + Prob_2 + Prob_3 = 1 \tag{6.10}$$

Analysons les *paires d'exécution* obtenues. Nous savons que $G(P^{(1)}|P^{(1)})$ prend un retard $\frac{2}{\delta^{(1)}}$ due au partage du bus mémoire. $G(P^{(0)}|P^{(1)})$ et $G(P^{(1)}|P^{(0)})$ ont la même probabilité d'exécution et la même charge. Le fait qu'un processus s'exécute sur un processeur spécifique ne change rien au modèle. Nous rappelons que $t_{(0)} = t_{(1)} = 1$ unités de temps.

Donc, nous obtenons :

$$T_{moyen} = \frac{n}{2}(Prob_0 + Prob_1 + Prob_2 + \frac{2}{\delta^{(1)}}Prob_3) \tag{6.11}$$
$$T_{moyen} = \frac{n}{2}((1-\beta_1)^2 + 2\cdot\beta_1(1-\beta_1) + \frac{2}{\delta^{(1)}}\cdot\beta_1^2) \tag{6.12}$$
$$\boxed{T_{moyen} = \tfrac{n}{2}(1+\beta_1^2(\tfrac{2}{\delta^{(1)}}-1))} \tag{6.13}$$

Dans les figures 6.2 et 6.3, nous avons représenté les deux autres cas : le *pire* et le *meilleur* ordonnancement. Sachant que les processus du type $P^{(1)}$ en séquentiel peuvent utiliser la totalité de la ressource bus mémoire, nous considérons comme le pire cas le groupement de ces exécutions. La *paire d'exécution* $G(P^{(1)}|P^{(1)})$ demande prés du double du débit que le bus mémoire peut fournir et donc il prend du retard. Ensuite, la *paire d'exécution* $G(P^{(0)}|P^{(0)})$ s'exécute en faisant une sous-utilisation du bus mémoire. En effet, d'abord, nous ralentissions l'exécution des processus en mettant des processus du type $P^{(1)}$ ensemble (d'un facteur $\frac{2}{\delta^{(1)}}$), et ensuite nous ordonnançons les processus du type $P^{(0)}$ en gaspillant la ressource bus mémoire. Donc, le *pire ordonnancement* maximise le temps d'exécution où les processus du type processus $P^{(1)}$ sont exécutés en concurrence entre eux :

$$T_{pire} = t_{G(\#P^{(1)}|\#P^{(1)})} + t_{G(\#P^{(0)}|\#P^{(0)})} \tag{6.14}$$

$$T_{pire} = \frac{n}{2} \cdot \frac{2}{\delta^{(1)}} \cdot \beta_1 + \frac{n}{2} \cdot (1 - \beta_1) \tag{6.15}$$

$$\boxed{T_{pire} = \frac{n}{2}(1 + \beta_1(\frac{2}{\delta^{(1)}} - 1))} \tag{6.16}$$

Pire ordonnancement avec $0\% < \beta_1 < 100\%$

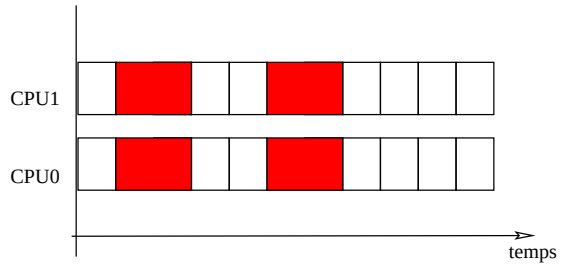

Figure 6.2 *Nous présentons un exemple de l'ordonnancement décrit dans le cas T_{pire}. Il n'y a que deux types de* paire d'exécution *: un composé par deux processus $P^{(1)}$ représenté par $G(P^{(1)}|P^{(1)})$ et l'autre composé par deux processus $P^{(0)}$ représenté par $G(P^{(0)}|P^{(0)})$. Les* paires d'exécution $G(P^{(1)}|P^{(1)})$ *chargent au maximum le bus mémoire et prennent du retard. Et, les* paires d'exécution $G(P^{(0)}|P^{(0)})$ *s'exécutent sans tirer profit du débit disponible sur le bus mémoire.*

Le dernier ordonnancement a été appelé le *meilleur ordonnancement*. Son but est de minimiser l'occurrence des *paires d'exécution* $G(P^{(1)}|P^{(1)})$. Nous présentons alors les deux situations : $\beta_1 \leq 50\%$ (figure 6.3) où $n_0 \geq n_1$ et $\beta_1 \geq 50\%$ (figure 6.4) où $n_0 \leq n_1$.

En fait, nous lançons le plus grand nombre de *paires d'exécution* $G(P^{(0)}|P^{(1)})$ et $G(P^{(1)}|P^{(0)})$, puis, pour $\beta_1 \leq 50\%$, nous aurons, forcement, $G(P^{(0)}|P^{(0)})$ et pour $\beta_1 \geq 50\%$ $G(P^{(1)}|P^{(1)})$. L'ordre d'exécution des *paires d'exécution* est aléatoire, donc, les

paires d'exécution composés par deux processus du même type ne sont pas forcement à la fin.

Meilleur ordonnancement avec $0\% < \beta_1 < 50\%$

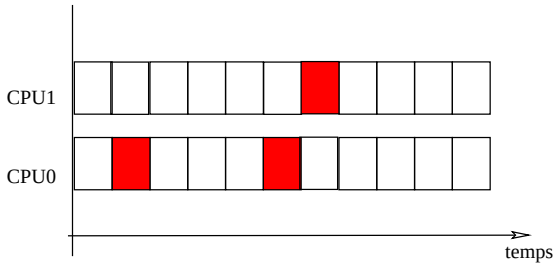

Figure 6.3 *Nous présentons un exemple d'ordonnancement appelé* meilleur ordonnancement *où $\beta_1 \leq 50\%$. Cet ordonnanceur essaie de maintenir l'utilisation du bus mémoire autour de 1 (capacité maximal), sans provoquer de ralentissement sur l'exécution des processus ni de gaspillage de la ressource. Cela étant, il exécute le plus grand nombre possible de* paires d'exécution $G(P^{(0)}|P^{(1)})$ *et $G(P^{(1)}|P^{(0)})$. Puisque le $n_0 \leq n_1$, donc aucun* paire d'exécution *du type $G(P^{(1)}|P^{(1)})$ ne s'exécutera.*

La représentation formelle pour le cas du *meilleur l'ordonnancement* est :

– Si $0\% \leq \beta_1 \leq 50\%$, donc $n_0 \geq n_1$:

$$T_{meilleur} = [t_{G(\#P^{(0)}|\#P^{(1)})} \text{ ou } t_{G(\#P^{(1)}|\#P^{(0)})}] + t_{G(\#P^{(0)}|\#P^{(0)})} \tag{6.17}$$

$$T_{meilleur} = \beta_1 n + \frac{(1-\beta_1)n - \beta_1 n}{2} \tag{6.18}$$

$$\boxed{T_{meilleur} = \tfrac{n}{2}} \tag{6.19}$$

– Si $50\% \leq \beta_1 \leq 100\%$, donc $n_0 \leq n_1$:

$$T_{meilleur} = [t_{G(\#P^{(0)}|\#P^{(1)})} \text{ ou } t_{G(\#P^{(1)}|\#P^{(0)})}] + \frac{2}{\delta^{(1)}} \cdot t_{G(\#P^{(1)}|\#P^{(1)})} \tag{6.20}$$

$$T_{meilleur} = (1-\beta_1)n + \frac{2}{\delta^{(1)}} \cdot \frac{\beta_1 n - (1-\beta_1)n}{2} \tag{6.21}$$

$$\boxed{T_{meilleur} = n(1 - \beta_1 + \tfrac{1}{\delta^{(1)}} \cdot (2 \cdot \beta_1 - 1))} \tag{6.22}$$

Finalement, ces modèles nous permettent de calculer une limite supérieure et une limite inférieure ainsi que le temps d'exécution moyen d'un ensemble de processus sur un biprocesseur. Nous présentons alors dans la figure 6.5 le comportement du temps d'exécution de ces trois ordonnancements en fonction de β_1. Le gain maximum est possible à $\beta_1 = 50\%$. Dans ce cas, $T_{meilleur}$ est 20% plus rapide que T_{moyen} et 33% que T_{pire}.

Lorsque nous achevons cette section, nous concluons alors que même une politique d'ordonnancement simple, comme celle proposée en $T_{meilleur}$, permet des gains supé-

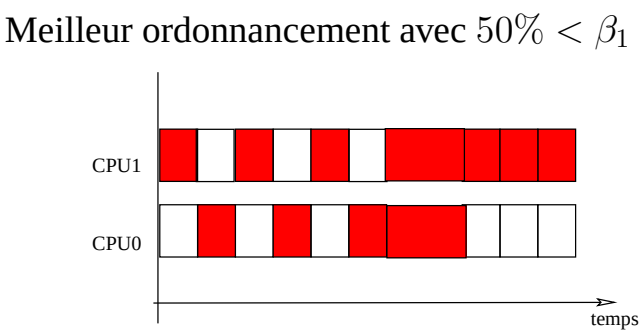

Figure 6.4 *Le deuxième exemple d'ordonnancement* meilleur ordonnancement *avec* $\beta_1 \geq 50\%$*. La technique employée pour cet ordonnancement est la même. Cependant, le nombre de processus du type* $P^{(1)}$ *est plus grand que* $P^{(0)}$*. Cela étant, cet ordonnanceur fait en sorte que les* paires d'exécution *lancées soient* $G(P^{(1)}|P^{(1)})$*,* $G(P^{(0)}|P^{(1)})$ *et* $G(P^{(1)}|P^{(0)})$*.*

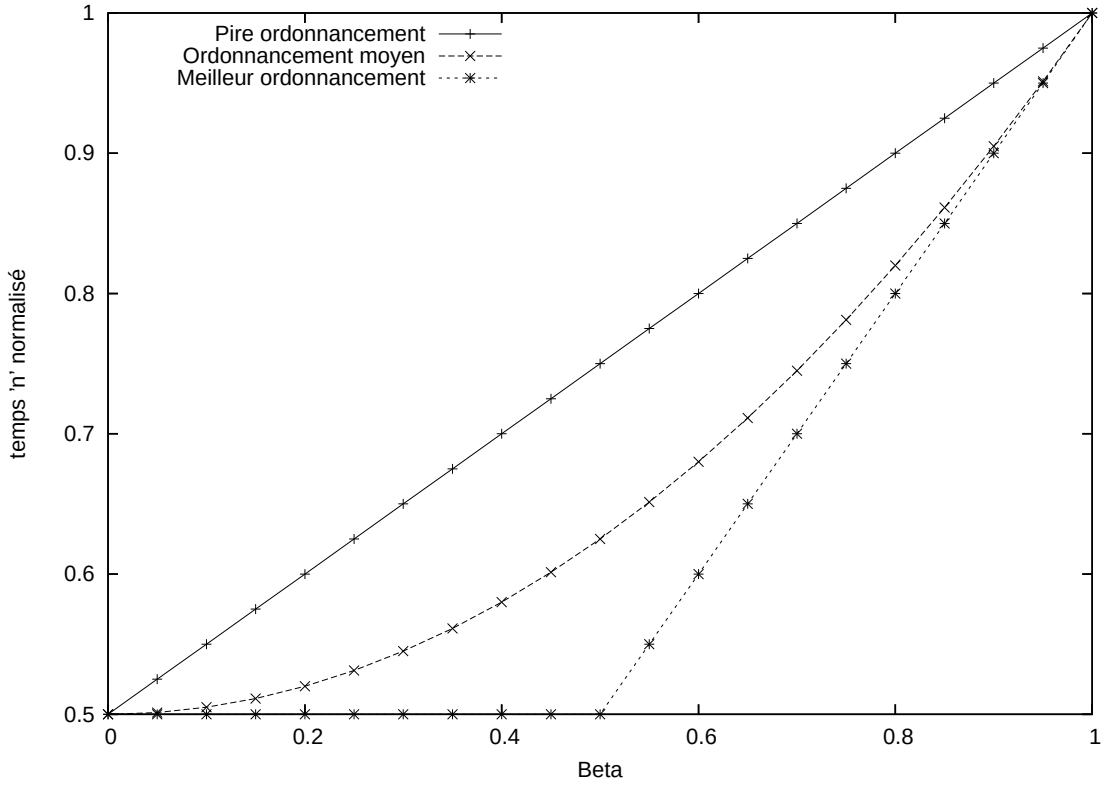

Figure 6.5 *Nous présentons ici les trois types d'ordonnancements modélisés par les formules* $T_{meilleur}$*,*T_{moyen} *et* T_{pire} *pour* $p = 2$*. Ces limites présentent un gain maximum quand* $\beta_1 = 50\%$ *dans lequel* $T_{meilleur}$ *est 20% plus rapide que* T_{moyen} *et 33% que* T_{pire}*.*

rieurs à 20% au cas moyen sur une machine biprocesseur. Nous modelons dans la prochaine section, cette étude de cas sur les machines quadriprocesseurs.

6.1.2.2 Quadriprocesseur

La deuxième étude de cas présentée décrit le modèle pour les mêmes trois types d'ordonnancements (T_{pire}, $T_{meilleur}$ et T_{moyen}) avec des machines quadriprocesseurs. L'objectif est de calculer le temps total d'exécution T_{total} d'un ensemble de processus concurrent.

La différence essentielle par rapport aux modèles précédents vient du nombre de processus en cours d'exécution qui passe de 2 à 4. Le scénario est également composé de deux types de processus $P^{(0)}$ et $P^{(1)}$ qui ont des charges sur le bus mémoire $\alpha(0) = [0, 0.1]$ et $\alpha(1) = [0.9, 1]$, respectivement. On note β_1 comme étant le pourcentage des processus du type $P^{(1)}$ sur l'ensemble de n processus. Enfin, l'accélération de l'exécution en concurrences des processus reste $t^{(k)}_{seq}/t^{(k)}_{total}$, avec $k = [0, 0.1]$ ou $[0.9, 1]$. Pourtant, l'augmentation du nombre de processeurs dégrade le cas de l'exécution en concurrence des processus du type $P^{(1)}$ entre eux, lequel peut prendre 4 fois plus de temps dans le pire cas. La valeur pour $P^{(0)}$ est évidement $\delta^{(0)} = 4$, puisque il n'y a pas de contention sur le bus mémoire, par contre, le valeur pour $P^{(1)}$ est $\delta^{(1)} = \frac{4}{max(1,4\alpha(j))}$.

Les ordonnancements pour le quadriprocesseur disposent ainsi de 2^p *groupe d'exécution* ce qui fait 16 combinaisons possibles. Afin de simplifier la représentation, on note $G(P^{(0)}|P^{(0)}|P^{(0)}|P^{(0)})$ ou $G(0 \cdot P^{(1)} 4 \cdot P^{(0)})$ comme étant un seul *groupe d'exécution* avec quatre processus du type $P^{(0)}$ et $G(\#P^{(0)}|\#P^{(0)}|\#P^{(0)}|\#P^{(0)})$ ou $G(0 \cdot \#P^{(1)} 4 \cdot \#P^{(0)})$ la représentation de tous les *groupes d'exécution* $G(P^{(0)}|P^{(0)}|P^{(0)}|P^{(0)})$.

Nous avons donc les 16 combinaisons possibles :

$$G(0 \cdot P^{(1)} 4 \cdot P^{(0)}) = G(P^{(0)}|P^{(0)}|P^{(0)}|P^{(0)}) \tag{6.23}$$

$$\begin{aligned} G(1 \cdot P^{(1)} 3 \cdot P^{(0)}) = {} & G(P^{(1)}|P^{(0)}|P^{(0)}|P^{(0)})|G(P^{(0)}|P^{(1)}|P^{(0)}|P^{(0)})| \\ & G(P^{(0)}|P^{(0)}|P^{(1)}|P^{(0)})|G(P^{(0)}|P^{(0)}|P^{(0)}|P^{(1)}) \end{aligned} \tag{6.24}$$

$$\begin{aligned} G(2 \cdot P^{(1)} 2 \cdot P^{(0)}) = {} & G(P^{(1)}|P^{(1)}|P^{(0)}|P^{(0)})|G(P^{(1)}|P^{(0)}|P^{(0)}|P^{(1)})| \\ & G(P^{(1)}|P^{(0)}|P^{(1)}|P^{(0)})|G(P^{(0)}|P^{(1)}|P^{(1)}|P^{(0)})| \\ & G(P^{(0)}|P^{(1)}|P^{(0)}|P^{(1)})|G(P^{(0)}|P^{(0)}|P^{(1)}|P^{(1)}) \end{aligned} \tag{6.25}$$

$$\begin{aligned} G(3 \cdot P^{(1)} 1 \cdot P^{(0)}) = {} & G(P^{(1)}|P^{(1)}|P^{(1)}|P^{(0)})|G(P^{(1)}|P^{(1)}|P^{(0)}|P^{(1)})| \\ & G(P^{(1)}|P^{(0)}|P^{(1)}|P^{(1)})|G(P^{(0)}|P^{(1)}|P^{(1)}|P^{(1)}) \end{aligned} \tag{6.26}$$

$$G(4 \cdot P^{(1)} 0 \cdot P^{(0)}) = G(P^{(1)}|P^{(1)}|P^{(1)}|P^{(1)}) \tag{6.27}$$

Le lien d'un processus sur un processeur nous est indifférent, nous considérons comme étant identiques les cas $G(P^{(1)}|P^{(1)}|P^{(0)}|P^{(0)})$ et $G(P^{(0)}|P^{(1)}|P^{(1)}|P^{(0)})$.

Donc, nous obtenons :

$$\begin{aligned} T_{total} = t_{G(0\cdot\#P^{(1)}|4\cdot\#P^{(0)})} + t_{G(1\cdot\#P^{(1)}|3\cdot\#P^{(0)})} + \\ t_{G(2\cdot\#P^{(1)}|2\cdot\#P^{(0)})} + t_{G(3\cdot\#P^{(1)}|1\cdot\#P^{(0)})} + \\ t_{G(4\cdot\#P^{(1)}|0\cdot\#P^{(0)})} \end{aligned} \tag{6.28}$$

où $t_{G(h'\cdot\#P^{(1)}|h''\cdot\#P^{(0)})}$ est le temps d'exécution dépensé par l'ensemble des *groupes d'exécution* défini par $G(h' \cdot P^{(1)}|h'' \cdot P^{(0)})$.

Nous allons donc vérifier les probabilités d'occurrence de chaque *groupe d'exécution*. Nous rappelons que la proportion de processus $P^{(1)}$ sur l'ensemble est de β_1 et que celle de processus $P^{(0)}$ est de $(1 - \beta_1)$, pour $n = n_0 + n_1$.

Les probabilités sont :

$$Prob_1(G(0 \cdot \#P^{(1)}|4 \cdot \#P^{(0)})) = (1 - \beta_1)^4 \tag{6.29}$$

$$Prob_2(G(1 \cdot \#P^{(1)}|3 \cdot \#P^{(0)})) = 4 \cdot \beta_1(1 - \beta_1)^3 \tag{6.30}$$

$$Prob_3(G(2 \cdot \#P^{(1)}|2 \cdot \#P^{(0)})) = 6 \cdot \beta_1^2(1 - \beta_1)^2 \tag{6.31}$$

$$Prob_4(G(3 \cdot \#P^{(1)}|1 \cdot \#P^{(0)})) = 4 \cdot \beta_1^3(1 - \beta_1) \tag{6.32}$$

$$Prob_5(G(4 \cdot \#P^{(1)}|0 \cdot \#P^{(0)})) = \beta_1^4 \tag{6.33}$$

Il est donc possible d'estimer le temps d'exécution moyen comme étant une distribution aléatoire des *groupes d'exécution* définis antérieurement. Pour cela, nous ajoutons alors les retards pris lors du partage du bus mémoire suivant le *groupe d'exécution*. Nous considérons que ces retards varient selon le nombre de processus du type $P^{(1)}$ sur l'ensemble du *groupe d'exécution*, qu'ils sont donc linéaires.

Nous obtenons :

$$T_{moyen} = \frac{n}{4}(Prob_1 + Prob_2 + \frac{2}{\delta^{(1)}} \cdot Prob_3 + \frac{3}{\delta^{(1)}} \cdot Prob_4 + \frac{4}{\delta^{(1)}} \cdot Prob_5) \tag{6.34}$$

$$\boxed{T_{moyen} = \frac{\beta_1^2 n}{\delta^{(1)}}(\beta^2 - 3 \cdot \beta + 3) + \frac{n}{4}(-3 \cdot \beta_1^4 + 8 \cdot \beta_1^3 - 6 \cdot \beta_1^2 + 1)} \tag{6.35}$$

Le *pire ordonnancement* utilise exactement le même principe que pour les machines à deux processeurs. La figure 6.6 illustre donc un exemple possible de cet ordonnancement, il ne contient que deux *groupes d'exécution*, un composé par des processus du type $P^{(0)}$ et l'autre par $P^{(1)}$. Le *groupe d'exécution* $G(4 \cdot P^{(1)}|0 \cdot P^{(0)})$ a une accélération $\delta^{(1)}$ et l'autre groupe $G(0 \cdot P^{(1)}|4 \cdot P^{(0)})$ a une accélération $\delta^{(0)}$.

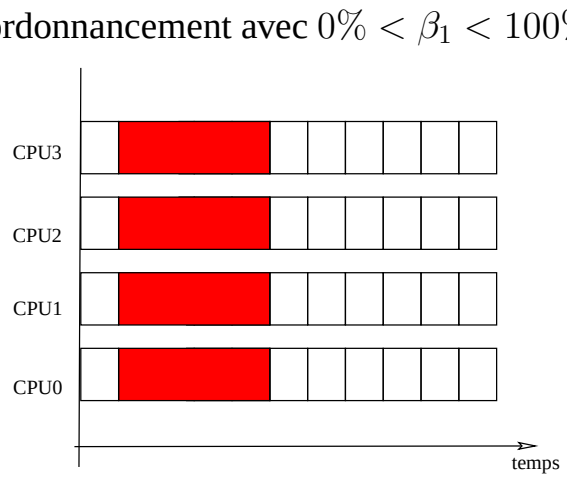

Figure 6.6 *Le pire ordonnancement est basé sur le même principe que celui sur les machines biprocesseurs. Il ne contient que deux* groupes d'exécution *un composé par des processus du type $P^{(0)}$ et l'autre par $P^{(1)}$. Ces groupes ont, respectivement, des accélérations $\delta^{(0)}$ et $\delta^{(1)}$.*

La représentation formelle pour le cas du *pire ordonnancement* est :

$$T_{pire} = t_{G(0\cdot\#P^{(1)}|4\cdot\#P^{(0)})} + \frac{4}{\delta^{(1)}} \cdot t_{G(4\cdot\#P^{(1)}|0\cdot\#P^{(0)})} \tag{6.36}$$

$$T_{pire} = \frac{n}{4} \cdot (1 - \beta_1) + \frac{n}{4} \cdot \frac{4}{\delta^{(1)}} \cdot \beta_1 \tag{6.37}$$

$$\boxed{T_{pire} = \tfrac{n}{4} \cdot (1 + \beta_1(\tfrac{4}{\delta^{(1)}} - 1))} \tag{6.38}$$

Après avoir présenté les ordonnancements T_{moyen} et T_{pire}, nous consacrons cette partie au dernier modèle d'ordonnancement, le $T_{meilleur}$. Il vise à former des *groupes d'exécution* avec le plus petit nombre de processus du type $P^{(1)}$ en concurrence entre eux. Nous présentons donc un exemple de chacun des trois cas possibles en fonction de la taille de β_1, figures 6.7, 6.8 et 6.9. Elles montrent des exemples du *meilleur ordonnancement*, dans le cas des machines quadriprocesseurs, lequel envisage de maintenir au niveau de l'utilisation du bus mémoire ni sous-utilisé ni saturé.

La stratégie de formation des *groupes d'exécution* dépend du nombre de processus du type $P^{(1)}$ sur l'ensemble de processus. Il est important de rappeler que la charge, sur le

bus mémoire, provoquée par un processus du type $P^{(1)}$ est de $\alpha(i) \in [0.9 \ldots 1]$. Prenons donc l'exemple avec α valant 0.9, c'est-à-dire que l'exécution d'un seul processus utilise déjà 90% du bus mémoire.

En se basent sur cette analogie, les *groupes d'exécution* formés quand $\beta_1 \leq 25\%$ sont : $G(1 \cdot P^{(1)}|3 \cdot P^{(0)})$ et $G(0 \cdot P^{(1)}|4 \cdot P^{(0)})$. De cette façon, il n'y a jamais de contention sur le bus mémoire pendant l'exécution. Malhereuseument, au fur et à mesure que β_1 augmente le nombre de *groupe d'exécution* qui provoque des retards augmente aussi. Il s'agit déjà du cas pour $25\% \leq \beta_1 \leq 50\%$ où nous avons des *groupes d'exécution* comme $G(2 \cdot P^{(1)}|2 \cdot P^{(0)})$ qui ralentissent l'exécution de l'ensemble des processus d'un facteur entre 1.8 et 2.0. Cependant, il est encore possible de former des *groupes d'exécution* comme $G(1 \cdot P^{(1)}|3 \cdot G^{(0)})$ qui ne provoquent pas des retards. Enfin, le dernier cas où $50\% \leq \beta_1 \leq 100\%$, nous avons les *groupes d'exécution* $G(2 \cdot P^{(1)}|2 \cdot P^{(0)})$ avec des retards entre 1.8 et 2.0 ainsi que $G(4 \cdot P^{(1)}|0 \cdot P^{(0)})$ avec des retards entre 3.6 et 4.0.

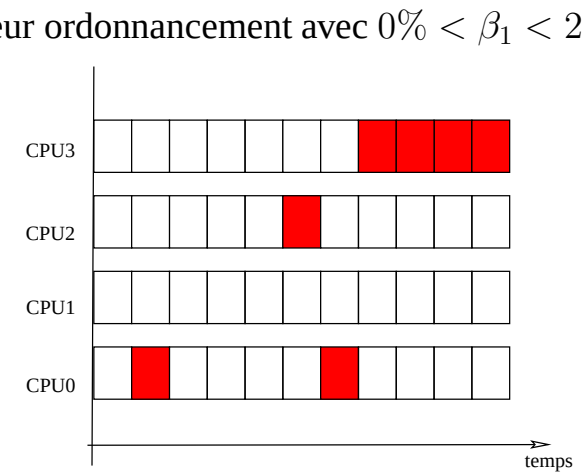

Figure 6.7 *Sur les machines quadriprocesseurs nous avons différentes situations pour le meilleur cas. Dans cette exemple, nous présentons le cas de* $0\% \leq \beta_1 \leq 25\%$. *Les deux* groupes d'exécution *formés sont :* $G(0 \cdot P^{(1)}|4 \cdot P^{(0)})$ *et* $G(1 \cdot P^{(1)}|3 \cdot P^{(0)})$. *Il n'y a pas de contention sur le bus mémoire.*

La représentation formelle pour le cas du *meilleur ordonnancement* est :

– Si $0\% \leq \beta_1 \leq 25\%$:

$$T_{meilleur} = t_{G(1 \cdot \#P^{(1)}|3 \cdot \#P^{(0)})} + t_{G(0 \cdot \#P^{(1)}|4 \cdot \#P^{(0)})} \tag{6.39}$$

$$T_{meilleur} = \beta_1 n + \frac{(1-\beta_1)n - 3 \cdot \beta_1 n}{4} \tag{6.40}$$

$$\boxed{T_{meilleur} = \tfrac{n}{4}} \tag{6.41}$$

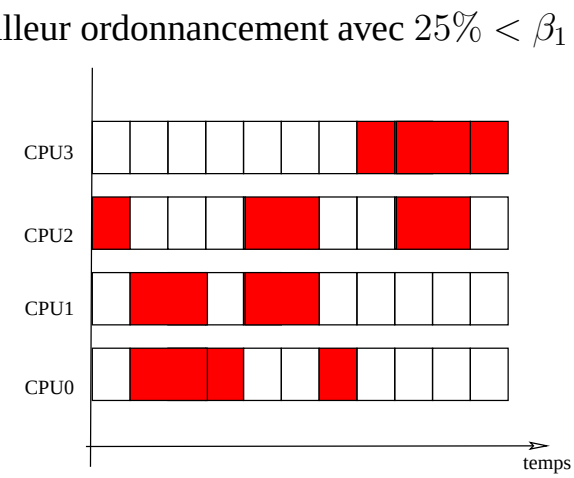

Figure 6.8 *Le deuxième exemple représente le cas pour* $25\% \leq \beta_1 \leq 50\%$ *où les* groupes d'exécution *formés sont :* $G(2 \cdot P^{(1)}|2 \cdot P^{(0)})$ *et* $G(1 \cdot P^{(1)}|3 \cdot P^{(0)})$*. Puisque il n'y a plus assez de processus du* groupe de processus $P^{(0)}$*. Il apparaît des cas avec de la contention mémoire et des retards provoqués par l'exécution de deux processus* $P^{(1)}$*.*

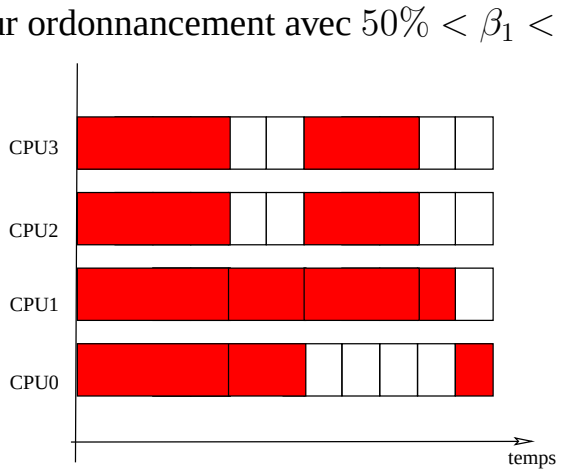

Figure 6.9 *Enfin, le dernier exemple montre le cas pour* $50\% \leq \beta_1 \leq 100\%$*. Les* groupes d'exécution *formés sont :* $G(2 \cdot P^{(1)}|2 \cdot P^{(0)})$ *et* $G(4 \cdot P^{(1)}|0 \cdot P^{(0)})$*. L'exécution de l'ensemble de processus du type* $P^{(1)}$ *atteind le niveau maximal de retards* $\delta^{(1)}$*.*

– Si $25\% \leq \beta_1 \leq 50\%$:

$$T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot t_{G(2 \cdot \#P^{(1)}|2 \cdot \#P^{(0)})} + t_{G(1 \cdot \#P^{(1)}|3 \cdot \#P^{(0)})} \tag{6.42}$$

$$T_{meilleur} = \frac{2n}{\delta^{(1)}} \cdot (\beta_1 - \frac{1}{4}) + (4 \cdot (\frac{1}{2} - \beta_1) \cdot \frac{n}{4}) \tag{6.43}$$

$$\boxed{T_{meilleur} = \frac{\beta_1 n}{\delta_1}} \tag{6.44}$$

– Si $50\% \leq \beta_1 \leq 100\%$:

$$T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot t_{G(2\cdot\#P^{(1)}|2\cdot\#P^{(0)})} + \frac{4}{\delta^{(1)}} \cdot t_{G(4\cdot\#P^{(1)}|0\cdot\#P^{(0)})} \tag{6.45}$$

$$T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot \frac{(1-\beta_1)n}{2} + \frac{\frac{4}{\delta^{(1)}} \cdot (\beta_1 - (1-\beta_1))n}{4} \tag{6.46}$$

$$\boxed{T_{meilleur} = \frac{\beta_1 n}{\delta_1}} \tag{6.47}$$

L'objectif du *meilleur ordonnancement* est d'optimiser l'utilisation du bus mémoire, c'est-à-dire qu'il ne doit pas être saturé ni sous-utilisé. Dans le cas du biprocesseur, il fait un maximum d'exécution d'un seul processus du type $P^{(1)}$ en concurrent avec un autre processus du type $P^{(0)}$. Sachant qu'un processus du type $P^{(1)}$ en cours d'exécution n'occupe que 90% du bus mémoire pour le cas $\alpha(i) = 0.9$, nous avons encore 10% du bus non utilisé.

Dans le cas de quadriprocesseur, le *meilleur ordonnancement* suit cette même politique pour $\beta_1 \leq 25\%$. Néanmoins, aussitôt que β_1 dépasse ce seuil, il augmente le nombre de processus du type $P^{(1)}$ dans le *groupe d'exécution* un à un afin de réduire l'ocurrence du *groupe d'exécution* $G(4 \cdot P^{(1)}|0 \cdot P^{(0)})$. Il minimise ainsi l'exécution du *groupe d'exécution* qui provoque le plus important niveau de contention sur le bus mémoire.

Lorsque nous finissons les modèles d'ordonnancements pour les quadriprocesseurs, nous illustrons, dans la figure 6.10, les temps d'exécution calculés via ces modèles en fonction de β_1. Sachant que les processus du type $P^{(1)}$ tous seul demandent déjà plus de 90% du débit du bus mémoire, la saturation du bus est atteind aussitôt que deux processus de ce type s'exécutent ensemble. Même pour le *meilleur ordonnancement*, la contention sur le bus est alors inevitable après $\beta_1 > 25\%$. Il s'agit exactement du cas $\beta_1 = 25\%$ lorsque $T_{meilleur}$ atteind son plus grand gain par rapport aux deux autres ordonnancements. Il est ainsi 24% plus rapide que T_{moyen} et 43% que T_{pire}.

Dans ces deux sections, nous avons présentés des modèles d'ordonnancement sur les biprocesseurs et les quadriprocesseurs. Nous envisageons alors d'évaluer le système de contrôle DRAC (chapitre 5) sur ces deux types de machines, dans la dernière partie de ce chapitre. Néanmoins, avant de présenter ces évaluations, nous concluons cette section par une généralisation de ces modèles d'ordonnancements.

6.1.2.3 Général

Cette section présente des modèles généraux pour les mêmes trois types d'ordonnancements théoriques définis précédemment. Les modèles considèrent des machines à p processeurs où p est un numéro entier fini plus grand que 2.

Nous commençons avec la formule pour obtenir le temps total d'exécution T_{total}. Nous n'avons que deux types de processus, $P^{(0)}$ ($\alpha(i) \sim 0$) et $P^{(1)}$ ($\alpha(i) \sim 1$). Un *groupe*

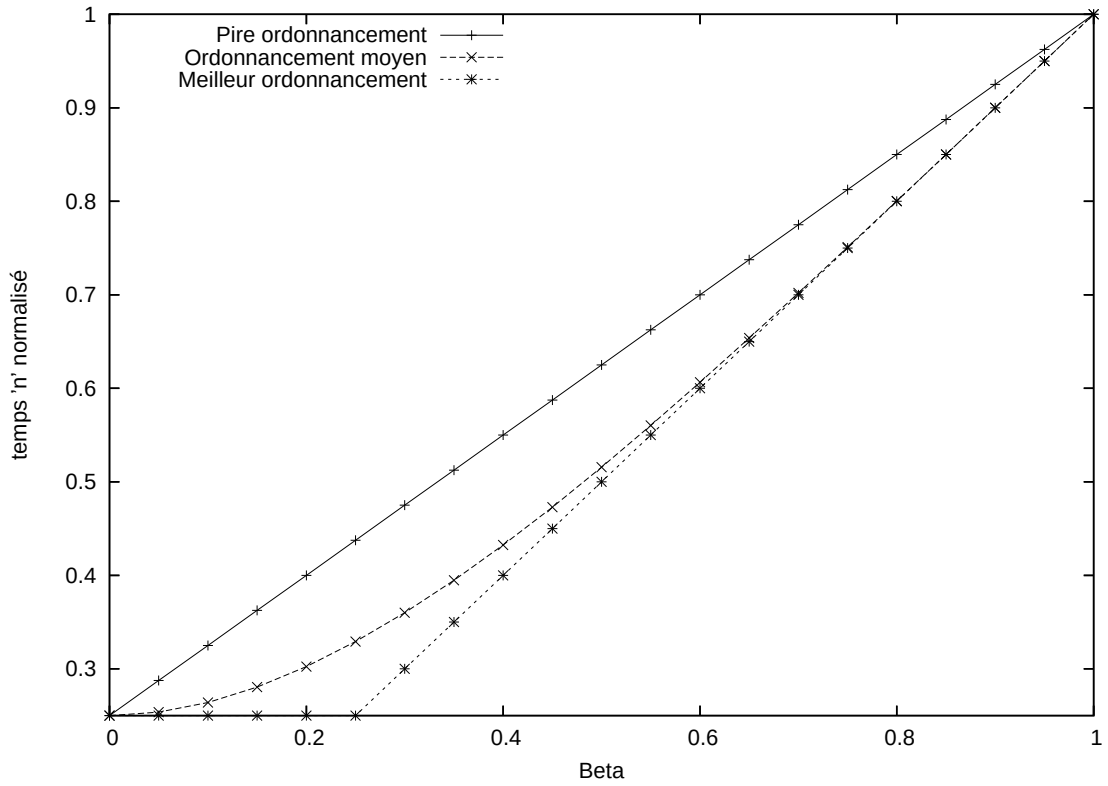

Figure 6.10 *Sur le quadriprocessueurs, les gains sont plus importants quand $\beta_1 \leq 25\%$. Dans ce cas, le* meilleur ordonnancement *est 24% plus rapide que l'*ordonnancement moyen *et 43% que le* pire ordonnancement.

d'exécution est représenté par $G(h \cdot P^{(1)}|(p-h) \cdot P^{(0)})$, où $h = 0 \ldots p$ et l'ensemble d'un *groupe d'exécution* $G(h \cdot \#P^{(1)}|(p-h) \cdot \#P^{(0)})$. Le nombre total de processus est $n = n_0 + n_1$. Enfin, le temps d'exécution de l'ensemble d'un *groupe d'exécution* spécifique est $t_{G(h \cdot \#P^{(1)}|(p-h) \cdot \#P^{(0)})}$.

$$T_{total} = \sum_{h=0}^{p} t_{G(h \cdot \#P^{(1)}|(p-h) \cdot \#P^{(0)})} \tag{6.48}$$

La somme des probabilités de tous les cas possibles avec p processeurs et deux types de processus ($P^{(0)}$ et $P^{(1)}$) est égal à 1. Le calcul de ces combinaisons est représenté par 2^p.

Donc, nous obtenons :

$$\sum_{h=0}^{p} Prob(G(h \cdot \#P^{(1)} | (p-h) \cdot \#P^{(0)})) = 1 \tag{6.49}$$

Prenons en compte le fait que β_1 représente le pourcentage de $P^{(1)}$ sur un ensemble n où $n = n_0 + n_1$, nous pouvons conclure que le nombre de processus du type $P^{(0)}$ sur l'ensemble est égal à $(1 - \beta_1)$. Enfin, le nombre de combinaisons possibles pour l'ensemble des *groupes d'exécution* est de $p + 1$.

Donc, la probabilité d'occurrence des *groupes d'exécution* est représente par :

$$Prob(G(h \cdot \#P^{(1)} | (p-h) \cdot \#P^{(0)})) = \sum_{i=0}^{p} C_i^p \cdot \beta_1^i (1-\beta_1)^{(p-i)} = 1 \tag{6.50}$$

Le calcul du temps total d'exécution sur une distribution aléatoire (T_{moyen}) est alors basé sur la somme des probabilités en ajoutant les retards dues au partage du bus mémoire. Puisque nous avons n processus qui s'exécutent en concurrence sur p processeurs, le nombre d'exécution est de $\frac{n}{p}$ exécutions. On note que l'exécution des processus du type $P^{(1)}$ en concurrence a ainsi une accélération $\delta^{(1)}$ obtenue par $\delta^{(1)} = \frac{p}{max(1, p\alpha(1))}$. Enfin, le modèle général qui permet de calculer le temps d'exécution de n processus sous un tirage aléatoire est :

$$\boxed{T_{moyen} = \frac{n}{p}(\sum_{i=0}^{p} C_i^p \cdot \beta_1^i (1-\beta_1)^{(p-i)})} \tag{6.51}$$

Toujours basé sur la définition de *groupe d'exécution*, nous présentons donc notre deuxième modèle général, *pire l'ordonnancement* : T_{pire}. Son objectif est de maximiser la formation des *groupes d'exécution* avec le plus grand nombre de processus du type $P^{(1)}$ en concurrence.

Nous obtenons :

$$\boxed{T_{pire} = \frac{n}{p}((1-\beta_1) + \frac{p}{\delta^{(1)}} \cdot \beta_1)} \tag{6.52}$$

où, $(1 - \beta_1)$ représente le *groupe d'exécution* $G(0 \cdot \#P^{(1)} | p \cdot \#P^{(0)})$ et β_1 le *groupe d'exécution* $G(p \cdot \#P^{(1)} | 0 \cdot \#P^{(0)})$, bien sûr, que ce dernier est multiplié par $\frac{p}{\delta^{(1)}}$ dû au retard provoqué par l'exécution des processus du type $P^{(1)}$ en concurrence.

Enfin, nous présentons le troisième modèle d'ordonnancement général, le *meilleur l'ordonnancement*, lequel vise à maintenir l'utilisation du bus mémoire à une charge

proche de 1. Il fait donc en sorte que la formation des *groupes d'exécution* minimise l'occurrence des groupes $G(p \cdot P^{(1)}|0 \cdot P^{(0)})$. Ce modèle privilégie ainsi les *groupes d'exécution* où la valeur $h = 2$ afin de s'assurer que la charge sur le bus mémoire n'est pas inférieure à 1 ni supérieure à 2. Cela étant, dans le cas où $\beta_1 \leq \frac{1}{p}\%$, la priorité devient la formation des *groupes d'exécution* où la valeur h est 1. Cela se justifie car il y a assez de processus du type $P^{(0)}$ pour s'exécuter avec tous ceux du type $P^{(1)}$, donc il n'y a jamais du retard. Les règles générales définies pour ce modèle ne sont valides que pour $p > 2$. Le cas particulier pour $p = 2$ a été déjà présenté précédemment.

Nous obtenons :

– Si $0\% \leq \beta_1 \leq (\frac{1}{p} \cdot 100)\%$:

$$T_{meilleur} = t_{G(1 \cdot \#P^{(1)}|(p-1) \cdot \#P^{(0)}))} + t_{G(0 \cdot \#P^{(1)}|p \cdot \#P^{(0)}))} \tag{6.53}$$

$$\boxed{T_{meilleur} = \beta_1 n + \frac{(1-\beta_1)n-(p-1)\cdot\beta_1 n}{p}} \tag{6.54}$$

– Si $(\frac{1}{p} \cdot 100)\% \leq \beta_1 \leq (\frac{2}{p} \cdot 100)\%$:

$$T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot t_{G(2 \cdot \#P^{(1)}|(p-2) \cdot \#P^{(0)}))} + t_{G(1 \cdot \#P^{(1)}|(p-1) \cdot \#P^{(0)}))} \tag{6.55}$$

$$\boxed{T_{meilleur} = \frac{2n}{\delta^{(1)}} \cdot (\beta_1 - \frac{1}{p}) + (\frac{2}{p} - \beta_1) \cdot n} \tag{6.56}$$

– Si $(\frac{2}{p} \cdot 100)\% \leq \beta_1 \leq 100\%$:

$$T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot t_{G(2 \cdot \#P^{(1)}|(p-2) \cdot \#P^{(0)}))} + \frac{p}{\delta^{(1)}} \cdot t_{G(p \cdot \#P^{(1)}|0 \cdot \#P^{(0)}))} \tag{6.57}$$

$$\boxed{T_{meilleur} = \frac{2}{\delta^{(1)}} \cdot \frac{(1-\beta_1)n}{p-2} + \frac{\frac{p}{\delta^{(1)}} \cdot (\beta_1 n - (1-\beta_1)n)}{p}} \tag{6.58}$$

Les trois modèles d'ordonnancement présentés dans cette section permettent ainsi d'évaluer les performances d'un ensemble de processus sur des multiprocesseurs à p processeurs. Nous considérons que les retards dus au partage du bus sont linéairement proportionnels à la quantité de processus du type $P^{(1)}$ placés dans les *groupes d'exécution*. Finalement, les temps d'exécutions d'un ensemble de processus dans les différentes ordonnancements peuvent être calculés.

Ces modèles sont basés sur l'hypothèse que les n processus peuvent être lancées en même temps. Malheureusement, le lancement d'un ensemble de processus dépend de la capacité mémoire. La prochaine section est donc consacrée à la description de ce problème.

6.1.2.4 Pourcentage d'un type de processus sur l'ensemble des processus lancés

Au lancement d'un processus, le système d'exploitation fait l'allocation mémoire initialement exigée à l'exécution de ce processus. La ressource mémoire est limitée et donc le nombre de processus lancés l'est aussi. Cela étant, dans les sections précédentes, les modèles d'ordonnancement présentés considèrent l'ensemble des processus et non seulement les processus lancés. Les résultats du modèle ne sont pas modifiés tant que l'ordonnanceur peut former les *groupes d'exécution* souhaités. Cela revient à trouver, parmi les processus lancés, un processus d'un type spécifique car il existe parmi l'ensemble des processus.

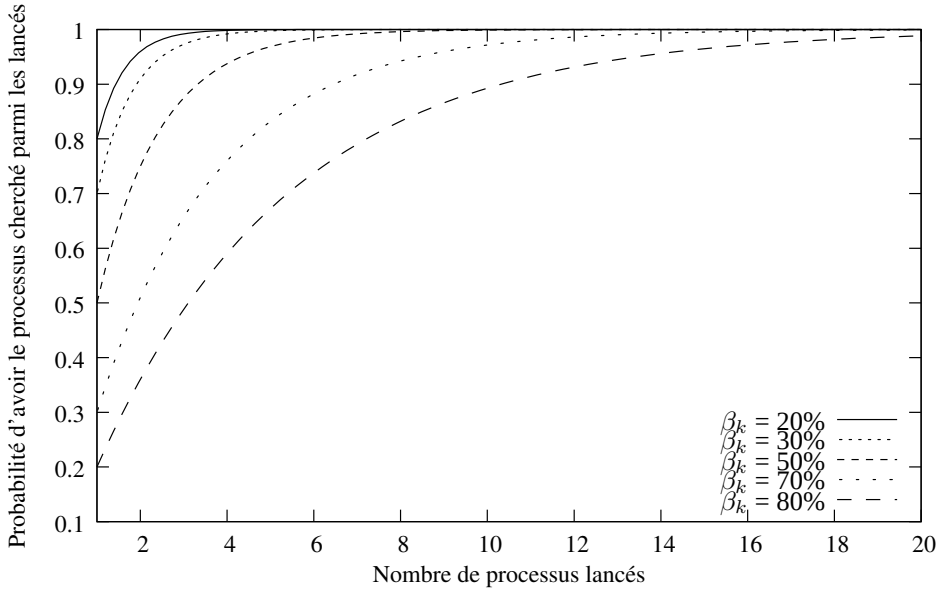

Figure 6.11 *La courbe présente la probabilité du tirage d'un processus d'un type spécifique sur l'ensemble de processus selon le nombre des processus lancés.*

Nous allons montrer que la probabilité qu'un processus du type cherché soit parmi les *processus lancés*. Dans l'ensemble de processus, nous avons ceux qui sont en cours d'exécution, en mémoire et en attente. On définit les *processus lancés* comme étant les processus *en cours d'exécution* et *en mémoire*. Le nombre de *processus lancés* est alors représenté par $nLance$. On note par β_k le pourcentage des processus du type $P^{(k)}$ sur l'ensemble des processus, où $k = [0, 1]$. Ainsi, on définit $Prob(P^{(k)})$ comme étant la probabilité d'avoir un processus du type $P^{(k)}$ sur l'ensemble des *processus lancés*.

Basé sur un tirage aléatoire, nous obtenons :

$$Prob(P^{(k)}) = (1 - \beta_k)^{nLance} \tag{6.59}$$

Nous présentons, dans la figure 6.11, les probabilités d'avoir un processus du type $P^{(k)}$ selon le nombre de *processus lancés*. Pour le cas $\beta_k = 20\%$, cette probabilité est proche de 1 pour $nLance = 20$. Si $\beta_k = 80\%$, la probabilité est proche de 1 pour un nombre de *processus lancés* plus petit, $nLance = 5$.

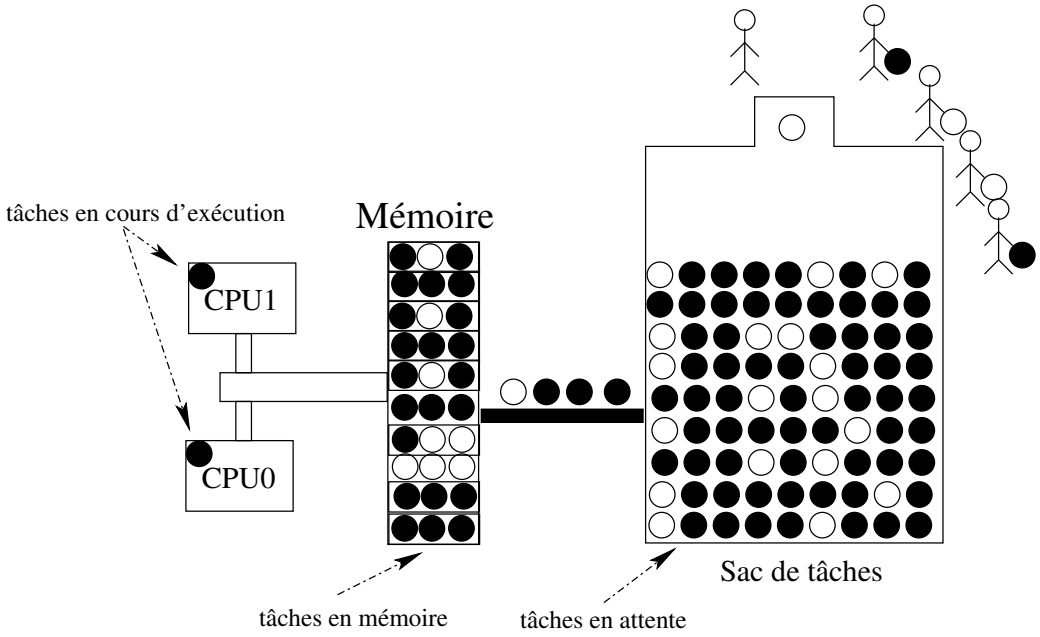

Figure 6.12 *La capacité mémoire limite le lancement de l'ensemble de n processus. Ainsi, la totalité des processus sont divisée en trois états : en cours d'exécution, en mémoire et en attente.*

Finalement, un exemple de scénario sur une machine biprocesseur est présenté dans la figure 6.12. Nous avons une tâche en cours d'exécution sur chaque processeur, la totalité de la mémoire est utilisée pour le lancement de processus et enfin, ce qui reste de l'ensemble de processus est dans le "sac de tâche" en attente. Nous avons deux types de processus, les noirs et les blancs. Le pourcentage des processus noirs sur l'ensemble est de 20%, et donc, celui des blancs est de 80%. Le nombre de *processus lancés* est au maximum 30. Nous avons alors une probabilité proche de 1 de trouver tant un *processus noir* qu'un *processus blanc* parmi les *processus lancés*. Si nous changeons le nombre maximum de *processus lancés* par 10, cette probalité est proche de 0.9. Nous pouvons donc supposé le tampon représenté para les tâches lancées suffisent aux bon fonctionnement du système DRAC.

Nos modèles sont restreints à l'exécution des *groupes d'exécution* composé par, seulement, deux types de processus $P^{(0)}$ et $P^{(1)}$. Dans un premier temps, nous avons étudié deux cas de figure, le premier pour les biprocesseurs, et le seconde pour les quadriprocesseurs, pour conclure avec une généralisation du problème. Nous sommes maintenant capables d'évaluer l'impact sur les performances provoqué par l'utilisation mémoire et de faire la comparaison entre les résultats du système DRAC et les modèles. Ainsi, nous

consacrons la prochaine section à la description de la méthodologie d'évaluation et aux résultats du prototype DRAC.

6.2 Évaluation du système DRAC

L'architecture du système de contrôle, DRAC, présentée dans le chapitre précédent propose l'adaptation de l'exécution des processus basée sur l'observation de l'utilisation mémoire. Le principe de contrôle du système est ainsi possible grâce au mécanisme d'estimation des performances en cours d'exécution décrit dans le chapitre 4.

Une partie de ce travail de thèse a été le développement d'un prototype du système. Il permet de valider l'architecture et d'évaluer les gains et les surcoûts éventuels provoqués par le démon de contrôle. Nous présentons alors dans cette section les résultats obtenus avec le prototype et les gains ou pertes par rapport aux modèles d'ordonnancements (voir section 6.1).

6.2.1 Présentation des résultats

Les résultats préliminaires du prototype du système ont été fait sur trois jeux de tests différents présentés dans la section 4.2. Les deux premiers utilisent les applications séquentielles, *STREAM Copy*, *SPEC2000 ART*, *SPEC2000 EON* et finalement un nouveau noyau de calcul, l'*Expo*. Ce dernier est une boucle qui fait le calcul d'une exponentielle ($a = (a^2)$). Enfin, l'algorithme *NAS CG* et *NAS EP* parallélisés avec les directives OpenMP forment l'autre jeu de tests. Chaque jeu de tests n'est composé que par deux types de programmes, un avec une charge proche de 1 et l'autre avec une charge proche de 0.

Tout d'abord, nous avons défini un jeu de test synthétique. Il est le seul flexible au sens qu'on peut l'adapter à la quantité de mémoire disponible sur la machine. Le premier programme, *STREAM Copy*, utilise 10% de la mémoire principale de la machine et le deuxième programme, *Expo*, une quantité de mémoire négligeable. Ce jeu de tests nous permet aussi de configurer le temps d'exécution de chaque noyau d'exécution. Le programme *STREAM Copy* contient une charge proche de 1 et la boucle exponentielle proche de 0 sur les architectures testées.

Le deuxième jeu de test a pour but d'évaluer le système avec des applications réelles, nous avons ainsi utilisé des programmes de l'ensemble de SPEC2000. Les algorithmes choisis sont *SPEC2000 ART* et *SPEC2000 EON*, le premier sollicite intensivement le bus mémoire même s'il n'a besoin que de $30Mo$ de mémoire et le seconde utilise seulement le processeur. Cela étant il utilise environ $10Mo$ de mémoire principale.

Enfin, le dernier jeu de test doit permettre d'évaluer le système avec des programmes

avec synchronisations. Nous avons ainsi utilisé l'algorithme *NAS CG* classe "B" qui a besoin d'environ $400Mo$ pour chaque instance et *NAS EP* aussi en classe "B" qui utilise environ $10Mo$.

Pour les programmes séquentiels, nous avons utilisé le compilateur GNU version 3.2 avec l'option *"-O3"* et pour les programmes parallélisé avec OpenMP le compilateur de *Portland Group* version 3.2 avec l'option d'optimisation *"-fast"*. Les machines testées sont sur le système d'exploitation Linux avec la version noyau 2.4. Les résultats représentent la moyenne de 30 exécutions et les courbes le pourcentage de programmes avec un fort niveau d'utilisation du bus mémoire ($P^{(1)}$) sur l'ensemble en fonction du temps d'exécution normalisé de la totalité des programmes.

6.2.1.1 Architecture processeur de la Famille Intel Pentium 6

Dans cette section, nous présentons les résultats obtenus sur les machines de la famille Intel *Pentium 6*. Les machines de cette famille testées sont : un *Pentium Pro* et un *Pentium II*.

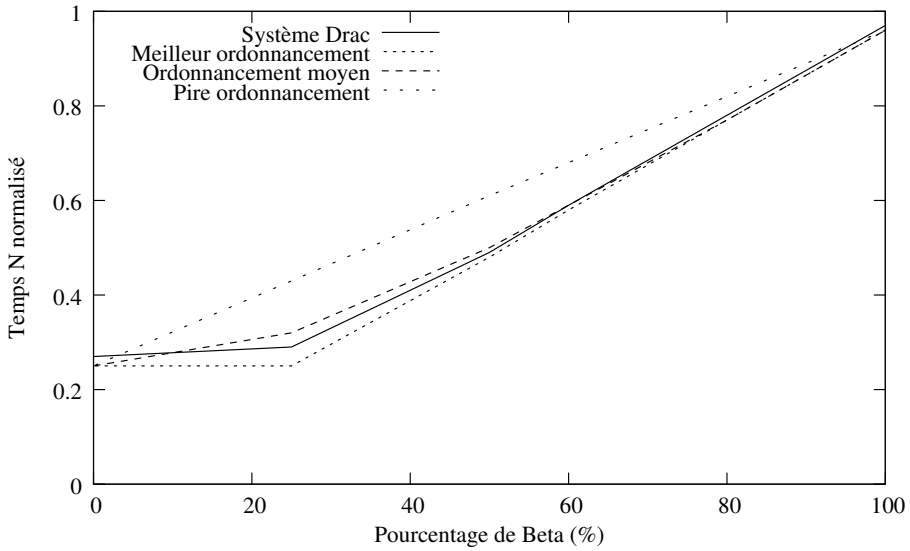

Figure 6.13 ***DRAC sur le Quadriprocesseur Pentium Pro :*** *l'axe X représente le pourcentage des programmes* STREAM Copy *lancés sur l'ensemble de 30 applications et dans l'axe Y leurs temps d'exécution normalisés. Le système DRAC est 9% plus rapide que l'ordonnancement moyen pour $\beta_1 = 25\%$, mais celui-ci s'inverse pour $\beta_1 \leq 10\%$, l'ordonnancement moyen devient ainsi 10% plus rapide que DRAC.*

Nos premiers résultats ont été obtenus sur le quadriprocesseur *Pentium Pro* à $200MHz$ et $512Mo$ de mémoire principale. Chaque test est composé par 30 programmes, qui sont soit *STREAM Copy* soit l'*Expo*. Le choix est réalisé par un tirage aléatoire. La charge sur

le bus mémoire pour le programme *STREAM Copy* est de $\alpha = 0.9$ et pour l'*Expo* est de $\alpha = 0$.

Au niveau du paramétrage de DRAC, nous avons fixé à 90% de la mémoire principale la limite maximum utilisée par l'ensemble de processus lancés puis à 0.5 secondes l'intervalle de prise de mesures et 10 le nombre de mesures utilisées par la prise de décision. Dans le cas $\beta_1 = 0\%$, il n'y a que des applications *Expo* avec des besoins en mémoire negligeables, elles peuvent donc être toutes lancées, dans le cas $\beta_1 = 100\%$, il n'y a que des applications *STREAM Copy* avec des besoins en mémoire d'environ 10% de la taille de la mémoire principale, et donc, une liste d'attente de 21 applications sur l'ensemble de 30.

Pour ce quadriprocesseur, la possibilité du gain le plus important, est pour $\beta_1 = 25\%$ (voir figure 6.13). Dans ce cas DRAC est 9% plus rapide que le résultat obtenu avec le modèle d'ordonnancement moyen. Néanmoins, l'ordonnancement moyen devient 10% plus rapide que DRAC pour $\beta_1 = 0\%$. Cette perte se prolonge jusqu'à ce que $\beta_1 \geq 10\%$. Cela s'explique simplement par le fait que le système essaie, à chaque nouvelle prise de décision, d'augmenter l'utilisation du bus mémoire en remplaçant les processus en cours d'exécution par un autre de type chargé. Puisque le nombre des processus chargés est réduit ou même inexistant, il est rare qu'il trouve l'équilibre du bus. La solution évidente pour ce problème est la prolongation automatique de l'intervalle de prise de décision par le démon DRAC aussitôt qu'il identifie cette situation.

De plus, les résultats sur le quadriprocesseur montrent que le système peut obtenir des gains sur cette architecture, mais il doit être amélioré afin de résoudre le problème de surcoût présenté dans certains cas.

La deuxième machine est un biprocesseur *Pentium II* à $450MHz$ et $512Mo$ de mémoire principale. Sur cette architecture nous avons lancé deux jeux de test : le jeu de tests synthétique avec les mêmes paramètres que sur la machine précédente et celui composé par les deux programmes séquentiels SPEC2000.

Avec le jeu de test synthétique, la seule différence par rapport à l'évaluation précédente est la charge mémoire du programme *STREAM Copy* qui passe de $\alpha = 0.9$ à $\alpha = 0.84$. Dans la figure 6.14, nous constatons que le système DRAC est plus performant que l'ordonnancement au pire et l'ordonnancement moyen pour n'importe quel β_1. Dans le meilleur cas, où $\beta_1 = 50\%$, l'ordonnancement moyen est 11% plus lent que DRAC. Nous rappelons que β_1 est le pourcentage de *STREAM Copy* sur l'ensemble d'applications.

Le deuxième jeu de test utilise l'algorithme *SPEC2000 ART* et l'algorithme *SPEC2000 EON*. Ils ont respectivement une charge mémoire d'un peu plus de 0.65 et de 0. Leurs besoins en mémoire sont inférieurs à $30Mo$, on peut donc lancer l'exécution d'au moins 18 programmes en concurrence. Dans ce cas, β_1 représente le pourcentage de *SPEC2000 ART* sur l'ensemble d'applications.

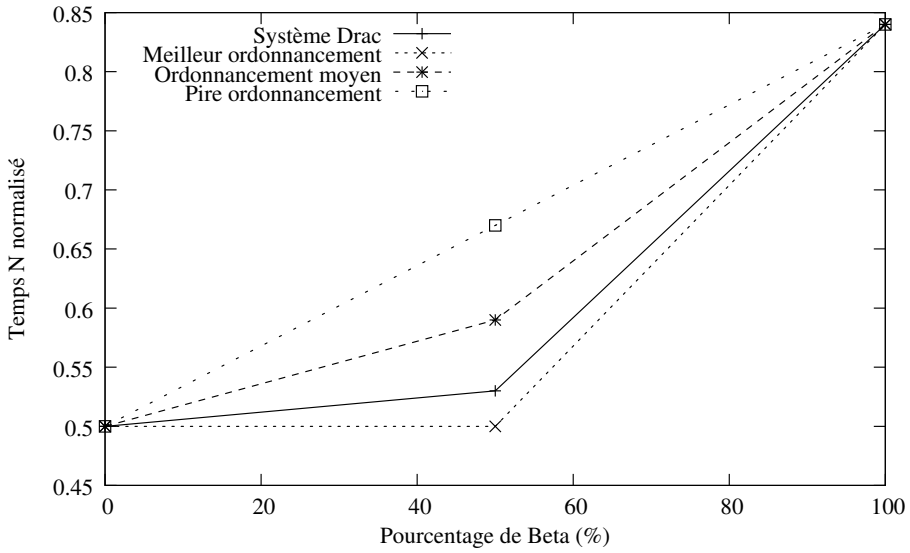

Figure 6.14 ***DRAC sur le Biprocesseur Pentium II :*** *Cette courbe illustre les résultats du temps d'exécution d'un ensemble de programmes (*STREAM Copy *et l'*Expo*) en fonction d'un pourcentage de l'application* STREAM Copy *(β_1) sur l'ensemble. Le système DRAC est plus performant que le pire l'ordonnancement et l'ordonnancement moyen pour n'importe quel β_1.*

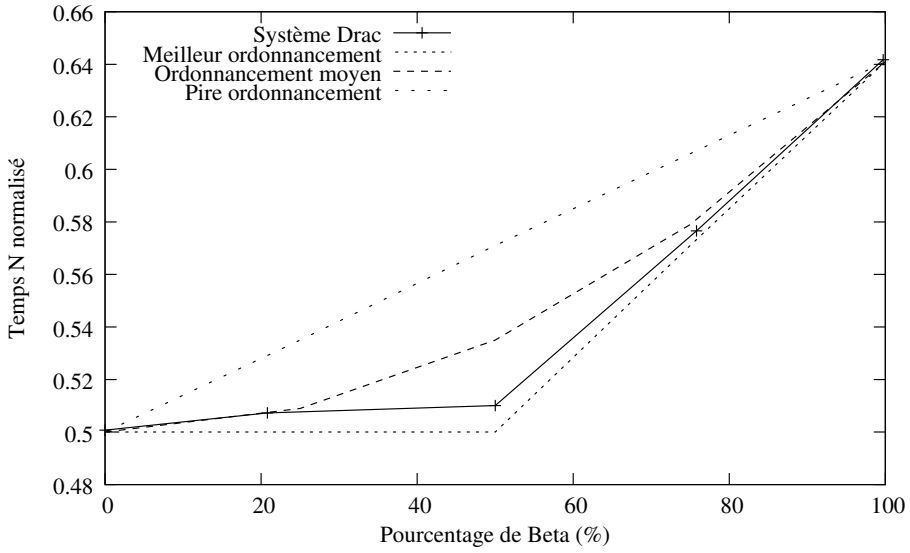

Figure 6.15 ***DRAC sur le Biprocesseur Pentium II :*** *Le deuxième jeux de test sur l'architecture* Pentium II *utilise les programmes* SPEC2000 ART *et* SPEC2000 EON. *Dans le meilleur cas ($\beta_1 = 50\%$), le système DRAC est 5% plus rapide que l'ordonnancement moyen et 12% que le pire ordonnancement.*

Les résultats de la figure 6.15 montrent que DRAC n'est que 2% plus lent que le meilleur ordonnancement et qu'il a les mêmes performances que l'ordonnancement moyen pour $\beta_1 = 25$. Ensuite, pour $\beta_1 = 50\%$, le système DRAC est 5% plus rapide que l'ordonnancement moyen et 12% que le pire ordonnancement.

Finalement, les résultats sur les deux machines nous permettent de valider le prototype et nous a permis d'identifier des problèmes lors de la mise au point, tels que les remplacements répétés par des processus d'un même type.

6.2.1.2 Architecture processeur de la Famille Intel Pentium 4

Dans cette section, nous présentons les résultats obtenus avec le prototype DRAC sur une machine de la famille *Pentium 4*. Cette machine possède deux processeurs *Pentium 4* à $1.7GHz$ et $1.5Go$ de mémoire principale. La méthodologie employée a été la même que pour les évaluations précédentes.

Néanmoins, nous avons élargi nos jeux de test, grâce à la capacité mémoire bien supérieure de cette machine aux précédentes. Nos premiers résultats ont été obtenus avec *STREAM Copy* et l'*Expo*, ensuite nous avons utilisé les deux algorithmes séquentiels *SPEC2000 ART* et *SPEC2000 EON* pour conclure avec *NAS CG* et *NAS EP* parallélisé avec les directives OpenMP.

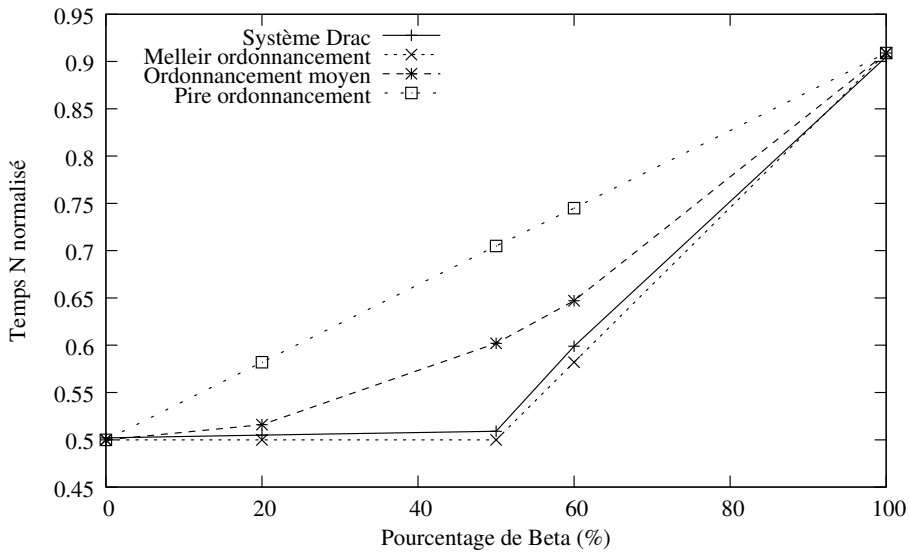

Figure 6.16 ***DRAC sur le biprocesseur Pentium 4 :*** *Le premier jeu de test lancé sur l'architecture* Pentium 4 *montre que DRAC obtient toujours des performances proches à celui du modèle du meilleur ordonnancement. L'ordonnancement moyen est 18% plus lent que DRAC et le pire ordonnancement arrive à 38% pour* $\beta_1 = 50\%$

La charge mémoire de l'algorithme *STREAM Copy* (α) est un peu plus que 0.9 et celle de l'*Expo* reste à 0. La figure 6.16 présente les courbes obtenues. Nous pouvons constater que l'exécution du système DRAC n'implique pas des coûts supplémentaires et que notre système obtient toujours des temps d'exécution proches du modèle meilleur ordonnancement. Quand $\beta_1 = 50\%$, l'ordonnancement moyen est 18% plus lent que ce obtenu par DRAC et le pire ordonnancement à 38%.

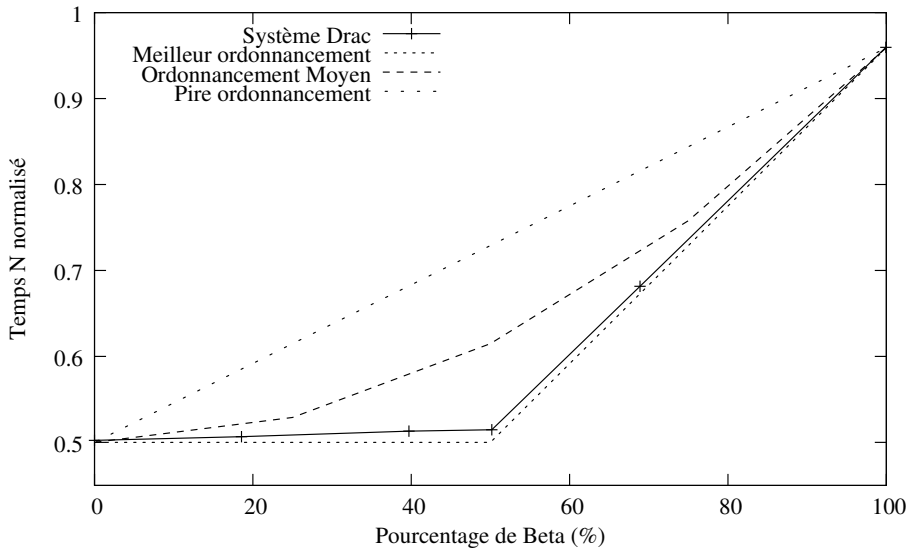

Figure 6.17 ***DRAC sur le biprocesseur Pentium 4 :*** *Ce second jeu de tests utilisé les algorithmes* SPEC2000 ART *et* SPEC2000 EON. *Notre système obtient des performances proches de celles du meilleur ordonnancement pour toutes les β_1. Quand $\beta_1 = 50\%$ notre système est 20% plus rapide que l'ordonnancement moyen et il est 41% plus rapide que le pire ordonnancement.*

Le second jeu de tests est composé par les algorithmes *SPEC2000 ART* et *SPEC2000 EON*. Le faible besoin en mémoire principale de ces algorithmes permet de lancer la totalité des programmes (30) en même temps. Néanmoins, nous voulons tester le comportement du système avec des processus dans la liste d'attente et donc nous l'avons limité à 20 les processus lancés.

Les résultats présentés dans la figure 6.17 montrent que les performances de DRAC sont proches de celles obtenues par le meilleur ordonnancement pour tous les β_1 (à 2%). Finalement, quand $\beta_1 = 50\%$ notre système est 20% plus rapide que l'ordonnancement moyen et il est 41% plus rapide que le pire ordonnancement.

Le troisième jeu de tests est composé par les algorithmes *NAS EP* et *NAS CG* parallélisés avec les directives OpenMP. Les résultats présentés dans la figure 6.18 montrent une chute de performances en relation à celles présentées sur la figure 6.17. DRAC obtient des performances proches à celui du modèle de l'ordonnancement moyen sauf pour $\beta_1 = 20\%$. La fine granularité de la boucle principale de calcul de l'algorithme *NAS CG*

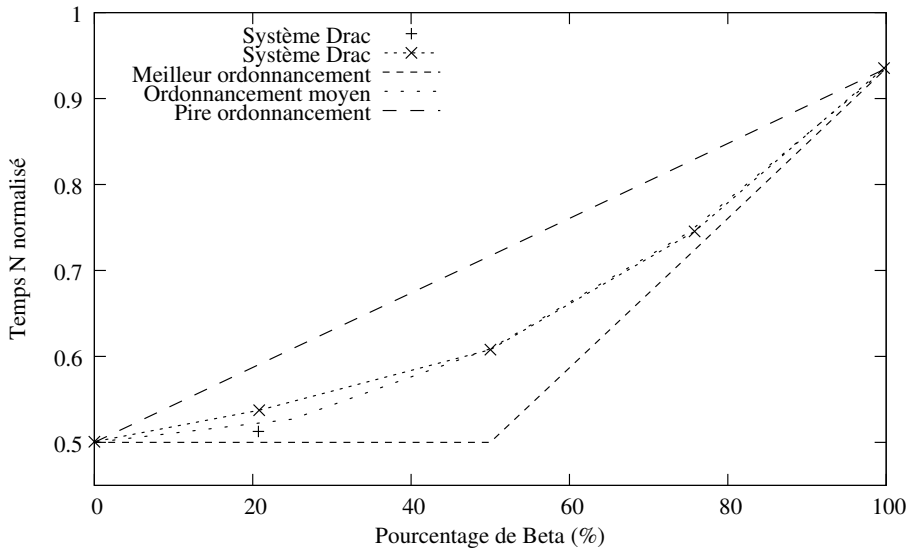

Figure 6.18 ***DRAC sur le biprocesseur Pentium 4 :*** *Le troisième jeu de test utilise des algorithmes* NAS EP *et* NAS CG *parallélisé avec les directives OpenMP. DRAC obtient des performances proches à celui du modèle de l'ordonnancement moyen sauf pour* $\beta_1 = 20\%$ *qu'il est* 7.5% *plus lent. Afin d'expliquer ce comportement, nous avons augmenté la granularite de la boucle parallèle de l'algorithme* NAS CG *d'environ* 10 *fois. Finalement, DRAC est devenu* 2.6% *plus rapide que l'ordonnancement moyen.*

explique ce comportement. En augmentant la période de temps entre les synchronisations de 10 fois, les performances de DRAC passent de 7.5% plus lent que l'ordonnancement moyen à 2.6% plus que rapide que ce même ordonnancement.

6.3 Bilan

Nous avons décrit dans ce chapitre un modèle simple des multiprocesseurs à mémoire partagée. Il permet de calculer le temps d'exécution avec le ralentissement produit par l'utilisation de la hiérarchie mémoire à mesure qu'elle devient saturée.

L'objectif principal de ce modèle est de définir des ordonnancements afin de comparer les résultats obtenus par le prototype du système DRAC. Dans l'étude présentée, nous avons utilisé seulement deux types de charges afin de simplifier le modèle.

C'est dans ce contexte, que nous avons évalué le système DRAC. Dans un premier temps, nous avons présenté les résultats sur des algorithmes séquentiels et puis les résultats avec deux algorithmes parallèles.

Finalement, le système est plus performant que l'ordonnancement moyen sur toutes

les tests des machines biprocesseurs et atteint dans la plupart des cas des résultats proche du meilleur ordonnancement. Il atteint des gains de plus de 40% sur certains cas.

7

Conclusion

Dans ce travail de thèse une solution a été proposée pour le problème de la saturation de la hiérarchie mémoire sur les machines multiprocesseurs à mémoire partagée. Cette solution repose sur un point de vue particulier qui est celui de l'augmentation du rendement global des machines au détriment des performances par application. Cette position répond généralement aux souhaits des administrateurs des machines. Nous avons proposé un système de contrôle d'exécution nommé DRAC qui permet l'augmentation du rendement. Son principe est basé sur la détection de saturations ou de sous-utilisations de la hiérarchie mémoire et de modifié en conséquence l'ordonnancement des processus. En cas de saturation le système cherche à soulager la hiérarchie mémoire, remplaçant le processus ayant le plus gros débit par un nécessitant moins de débit. Et en cas de sous-utilisation, le critère de remplacement est inversé.

Ce système pour fonctionner a besoin d'estimer le rendement de la machine en cours d'exécution. C'est une partie centrale du système proposé. Cela revenait à identifier les composants de la hiérarchie mémoire capable d'estimer les performances des applications en cours d'exécution. Grâce à l'étude des compteurs matériels de performances et aux tests avec les événements qu'ils mettent à disposition sur l'ensemble des architectures testées, nous avons identifié le moyen d'établir la relation souhaitée entre l'activité sur la hiérarchie mémoire et le rendement de la machine.

La complexité du choix des événements dépend essentiellement de l'architecture processeur. D'une part, cela suppose une bonne connaissance du fonctionnement interne des processeurs utilisés, ainsi que de la disponibilité de bibliothèque stable d'accès aux compteurs d'événements matériels. Parmi les architectures étudiées, une architecture s'est révélée particulièrement complexe à manipuler (famille de processeur *Pentium 4*), une autre ne disposait pas d'événement assez riches pour établir la relation recherchée (famille de

processeur *Athlon*).

L'architecture du système de contrôle DRAC est composée de trois modules : le moniteur, l'ordonnanceur et le gestionnaire de tâches. Le module moniteur surveille le niveau d'activité de la hiérarchie mémoire via l'accès aux compteurs d'événements. Les prises de mesures sont d'un coût négligeable, par contre, les valeurs sont bruitées et elles nécessitent d'être lissées. L'ordonnanceur applique une politique simple de contrôle de processus suivant l'utilisation du bus mémoire avec un algorithme de contrôle assurant une bonne robustesse. Enfin, le gestionnaire de tâches fait la gestion effective des processus du système. Le contrôle des processus exercé par ce module est réalisé au niveau utilisateur via la suspension et le redémarrage des processus. Dans le cas simple, sans synchronisation, le système a seulement besoin d'identifier l'entrée et la terminaison des processus. Par contre, dans les cas où les processus possèdent des points de synchronisation, le système a besoin d'identifier l'arrivée des processus sur ces points. Cette identification est réalisée par l'instrumentation des différentes bibliothèques de synchronisations.

Une modélisation de ce système a été présentée, elle permet de donner les performances dans différent cas d'ordonnancement ; pire, aléatoire et meilleur. Cette modélisation est valable sous certaines hypothèses réalistes. Les résultats de ce système sont encourageants, ils présentent des gains supérieurs à 40% dans certains cas. Actuellement, il fonctionne sur les processeurs de la famille Pentium 6 et de la famille Pentium 4. Parmi l'ensemble de tests, il n'y a eu qu'un cas où le système allongeait le temps d'exécution de l'ensemble des applications. Ce résultat vient d'un problème de réglage automatique d'intervalle du temps de la boucle de contrôle.

Finalement, ce travail de thèse laisse entrevoir plusieurs perspectives. À court terme, nous disposerons d'un portage du système sur les architectures $64bits$, Opteron et Itanium 2, ainsi qu'une extension du support des fonctions de synchronisation pour OpenMP. Cela nous permettra d'enrichir nos tests notamment en évaluant le système sur des machines multiprocesseur ayant un plus grand nombre de processeurs. À moyen terme avec la large diffusion de processeur muti-coeur, il sera intéressant d'observer si l'impact sur la hiérarchie mémoire s'est accru et d'étudier les bénéfices apportées par le système DRAC. La politique d'ordonnancement utilisée est très simple et ne prend comme critère que le débit de mémoire, nous manquons actuellement de recul pour affirmer que ce seul critère suffise dans de cas complexe : présence d'entrée-sortie, dépendance avec des applications sur d'autre noeuds. Une nouvelle étude sera nécessaire.

Ce dernier point, nous ramène à la problématique du contrôle d'exécution dans le cadre des grappes et des grilles. Les principes utilisés dans ce travail sont transposables dans ces domaines. Ils le sont en partie dans des projet comme CODE [100], MUSE [50] ou dans une certaine mesure par les méthodes de co-ordonnancement (*co-scheduling*). Une première piste qui nous semble intéressante est la transposition du contrôle d'exécution dans le cas du traitement d'un très grand nombre de tâches dans les grilles. Dans ce cadre la ressource critique à observer est principalement le réseau reliant les différents grappes qui peut être surchargé par les transferts de données dû à l'exécution des tâches.

Annexe A

L’ensemble des résultats de l’activité mémoire

Dans cette section, nous présentons l’ensemble des résultats effectués pour obtenir la relation entre le débit mémoire et l’accélération ainsi que le profil temporel des applications étudiées. Les quatre premières courbes présentées dans cette section sont les mêmes que celles de la section 4.3.2 où nous avons ajouté la valeur maximale et minimale pour chaque point. Ensuite, nous avons l’ensemble des courbes du profil du débit mémoire pour chacune des applications sur l’architecture *Pentium II* et *Pentium 4* (section 4.3.3).

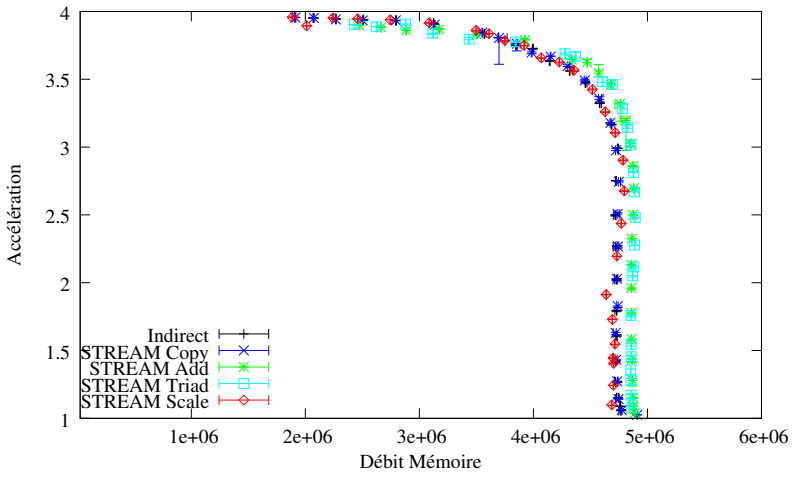

Figure A.1 ***Quadriprocesseur Intel Pentium Pro :*** *Les résultats sur les programmes de tests STREAM, Stride et Copie2 montrent l’évolution de l’accélération en fonction de l’utilisation mémoire pour un quadriprocesseur Pentium Pro.*

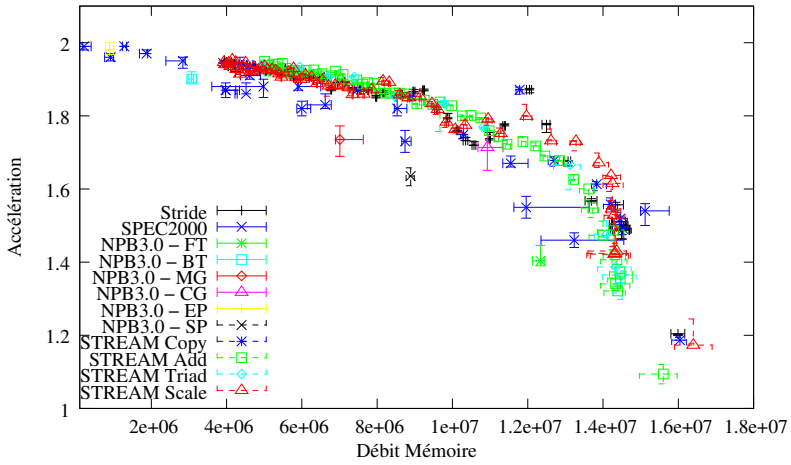

Figure A.2 ***Biprocesseur Intel Pentium II :*** *Les programmes NAS sont représentés pour des points et n'ont pas eu de modifications. Sur les autres programmes, nous avons ajouté des instructions assembler (*nop*) sur leur boucle principale pour soulager l'utilisation mémoire. Nous observons des points qui varient le débit mémoire, telle que quelques applications SPEC2000.*

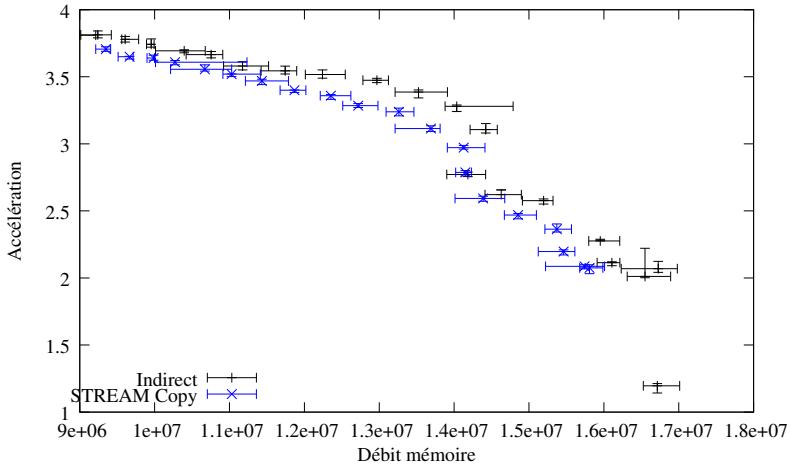

Figure A.3 ***Quadriprocesseur Intel Pentium III :*** *Les résultats préliminaires présentés sur la machine quadriprocesseur Pentium III constatent aussi le rapport entre la chute de l'accélération et le niveau d'utilisation du bus mémoire. La plus grande variation est observée pour Stride avec un niveau mémoire bien élevé.*

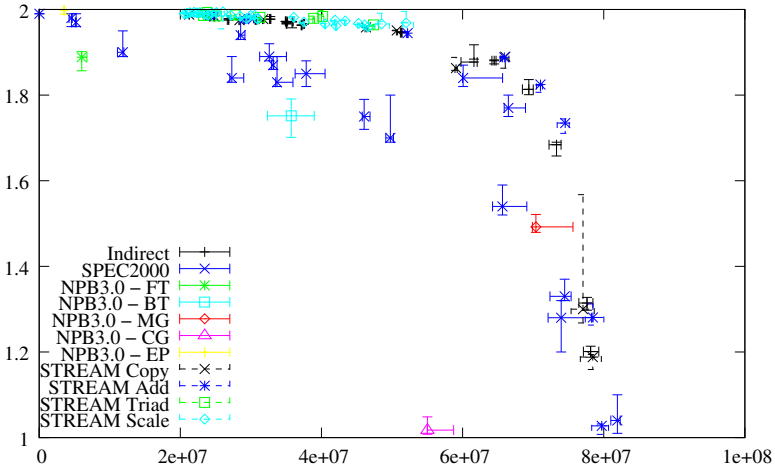

Figure A.4 ***Biprocesseur Intel Pentium 4 :*** *Les résultats de cette courbe permettent d'établir un rapport entre l'accélération et l'utilisation mémoire des applications. Les applications peuvent être classées en deux zones* non-saturée *et* intermédiaire. *Malheureusement, le comportement des applications ne permet pas une analyse fine de l'évolution de l'accélération selon le débit mémoire.*

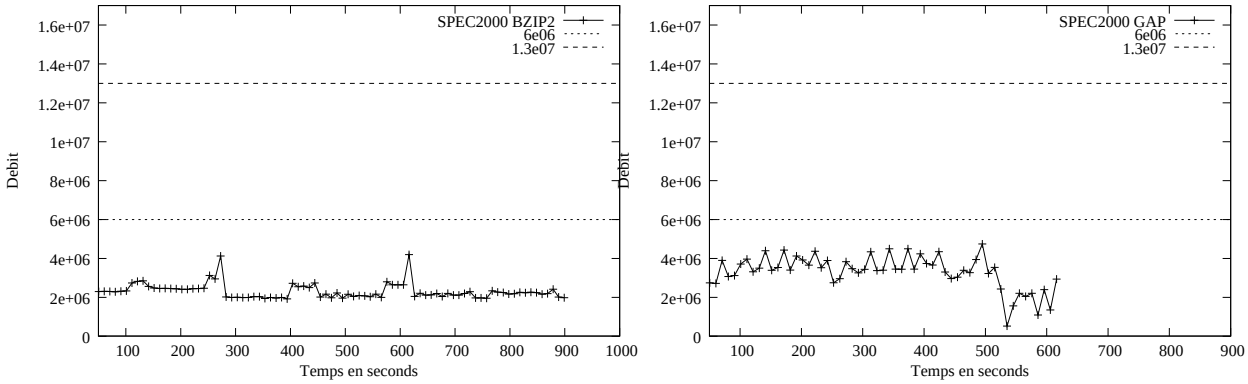

Figure A.5 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Bzip2 et, à gauche, celui de l'application SPEC2000 Gap. Les deux applications sont classées comme constantes.*

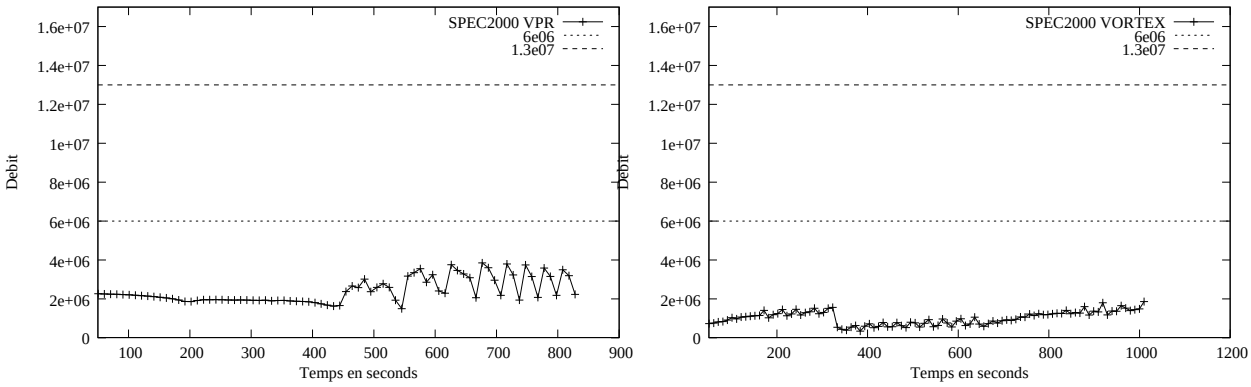

Figure A.6 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Vpr et, à gauche, celui de l'application SPEC2000 Vortex. Les deux applications sont classées comme constantes.*

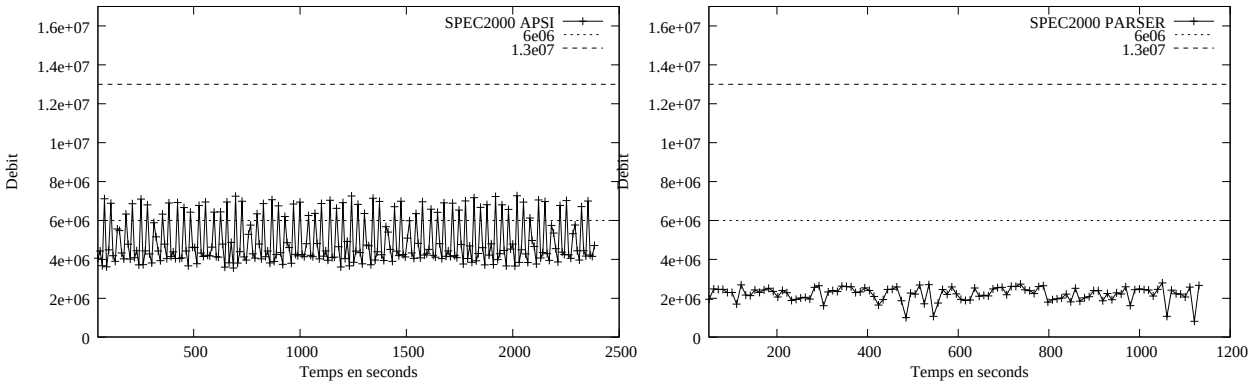

Figure A.7 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Apsi et, à gauche, celui de l'application SPEC2000 Parser. L'application Apsi se trouve bien à la frontière entre la zone non-saturée et la zone intermédiaire. Même si les différences entre les débits ne sont pas énormes, elle présente un comportement classé comme irrégulier. Par contre, l'application Parser est classée comme constante.*

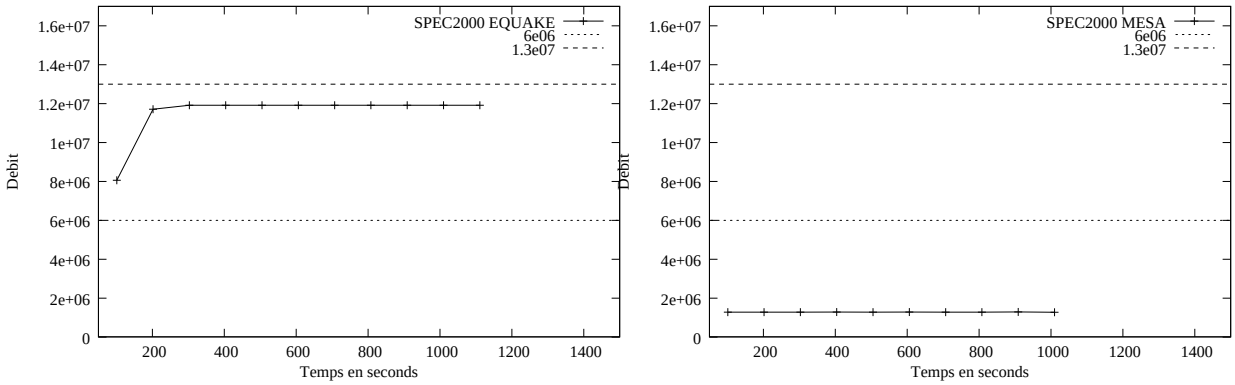

Figure A.8 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Equake et, à gauche, celui de l'application SPEC2000 Mesa. Les deux applications sont classées comme constantes, par contre la première est dans une zone non-saturée et la deuxième dans une zone intermédiaire.*

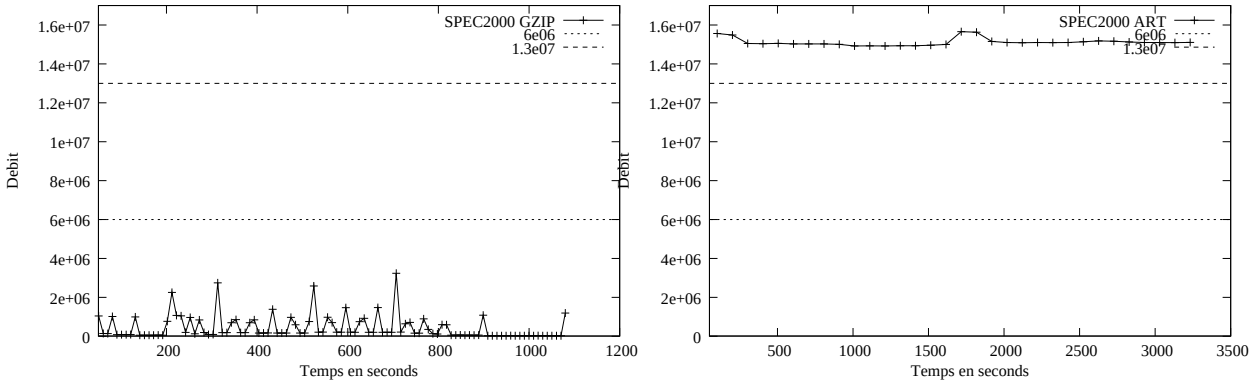

Figure A.9 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Gzip et, à gauche, celui de l'application SPEC2000 Art. Les deux applications sont classées comme constantes. L'Art est dans une zone saturée.*

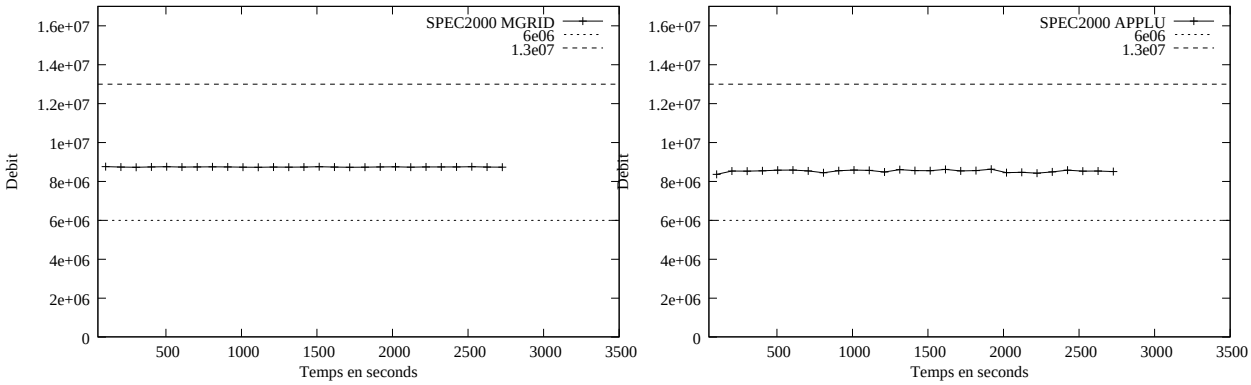

Figure A.10 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Mgrid et, à gauche, celui de l'application SPEC2000 Applu. Les deux applications sont classées comme constantes et se trouvent dans la zone intermédiaire.*

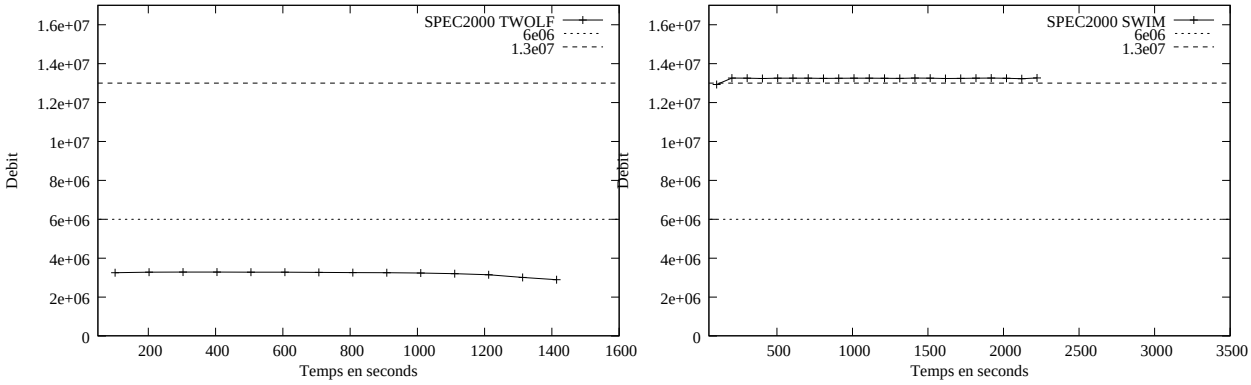

Figure A.11 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Twolf et, à gauche, celui de l'application SPEC2000 Swim. Les deux applications sont classées comme constantes, par contre la première est dans une zone non-saturée et la deuxième dans une zone saturée.*

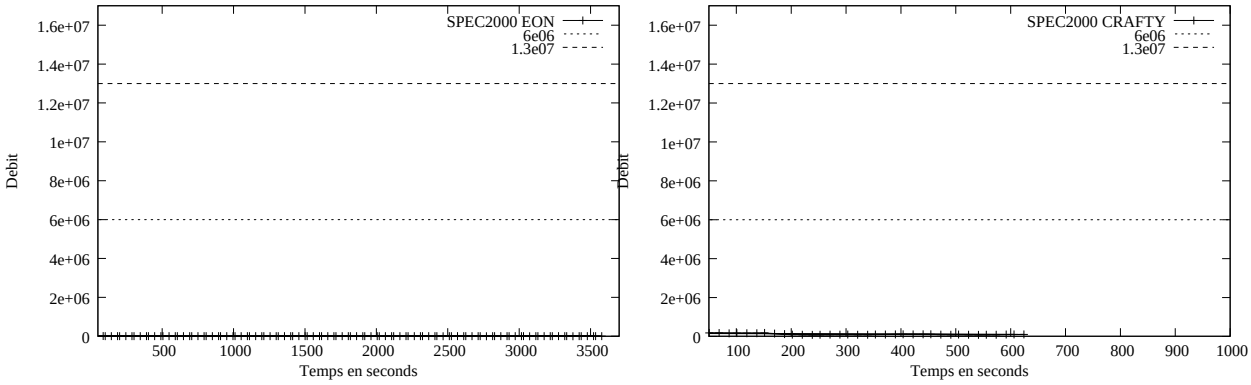

Figure A.12 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Eon et, à gauche, celui de l'application SPEC2000 Crafty. Les deux applications sont classées comme constantes et font très peu d'accès mémoire (zone non-saturée).*

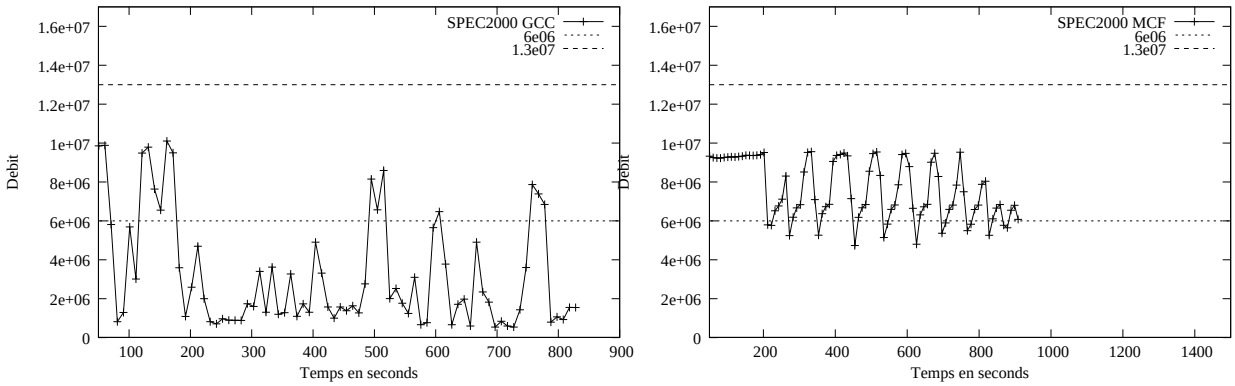

Figure A.13 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Gcc et, à gauche, celui de l'application SPEC2000 Mcf. Les deux applications sont classées comme irrégulières.*

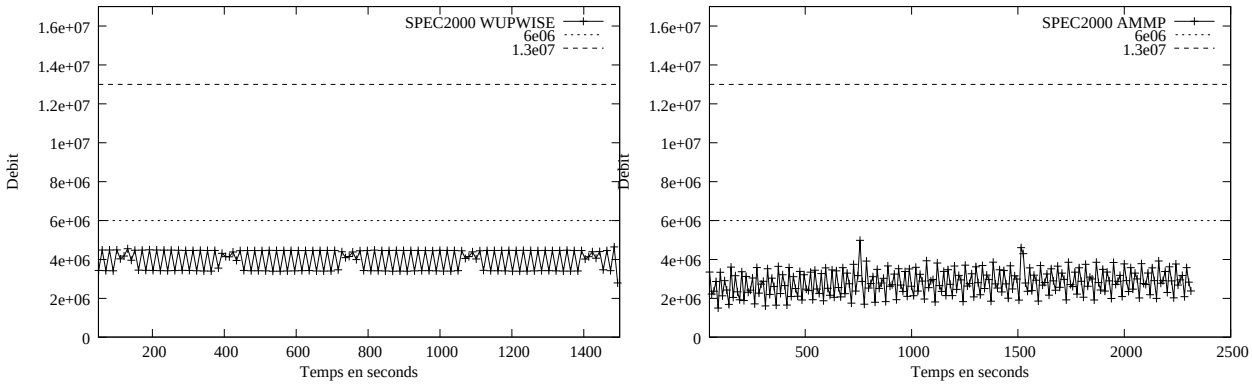

Figure A.14 ***Biprocesseur Intel Pentium II :*** *À droite, nous avons le comportement de l'application SPEC2000 Wupwise et, à gauche, celui de l'application SPEC2000 Ammp. Les deux applications sont classées comme constantes.*

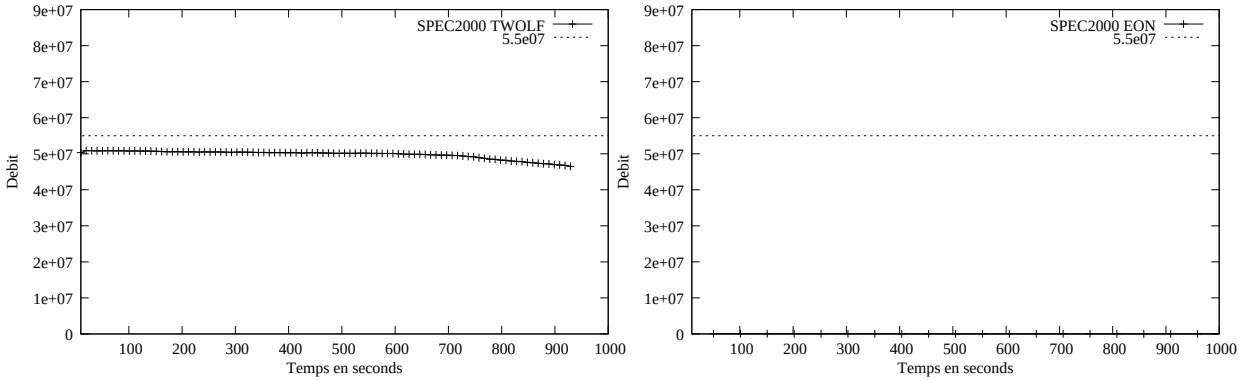

Figure A.15 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Twolf et, à gauche, celui de l'application SPEC2000 Eon. Les deux applications sont classées comme constantes et sont dans la zone non-saturée, même si les différences des débits entre les deux sont considérables.*

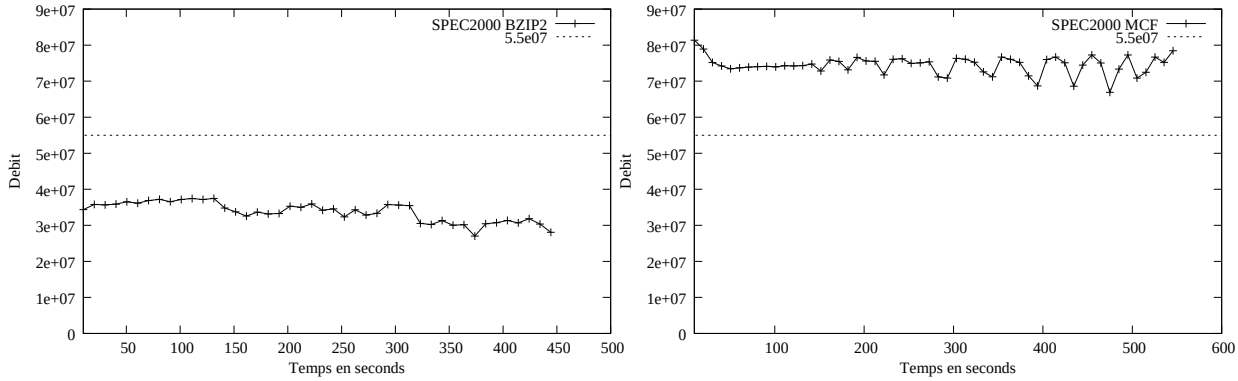

Figure A.16 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Bzip2 et, à gauche, celui de l'application SPEC2000 MCF. Les deux applications sont classées comme constantes. La première est dans une zone non-saturée et la deuxième intermédiaire.*

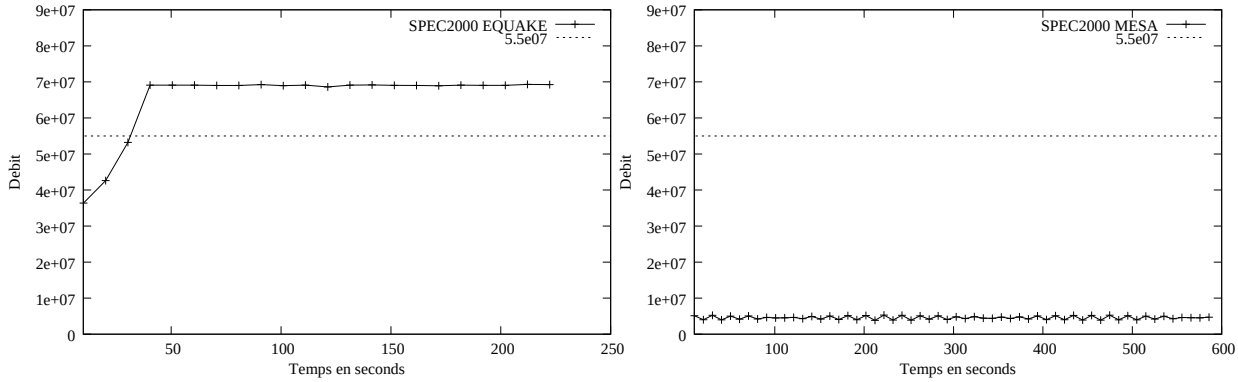

Figure A.17 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Equake et, à gauche, celui de l'application SPEC2000 Mesa. Les deux applications sont classées comme constantes, une dans une zone non-saturée et lautre dasn une zone intermédiaire.*

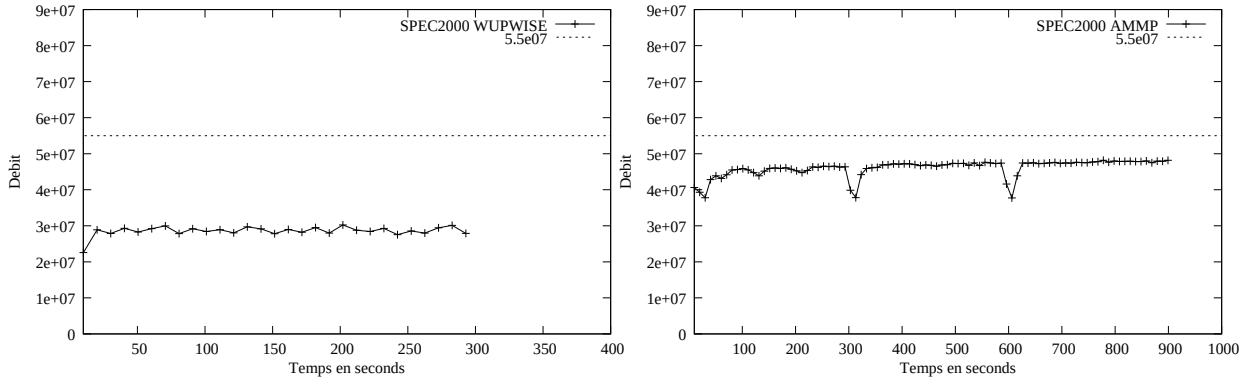

Figure A.18 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Wupwise et, à gauche, celui de l'application SPEC2000 Ammp. Les deux applications sont classées comme constantes.*

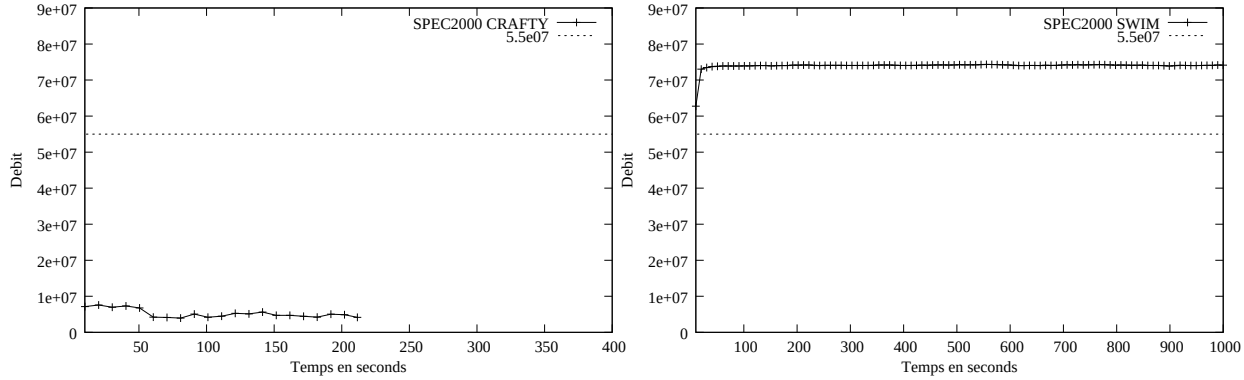

Figure A.19 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Crafty et, à gauche, celui de l'application SPEC2000 Swim. Les deux applications sont classées comme constantes.*

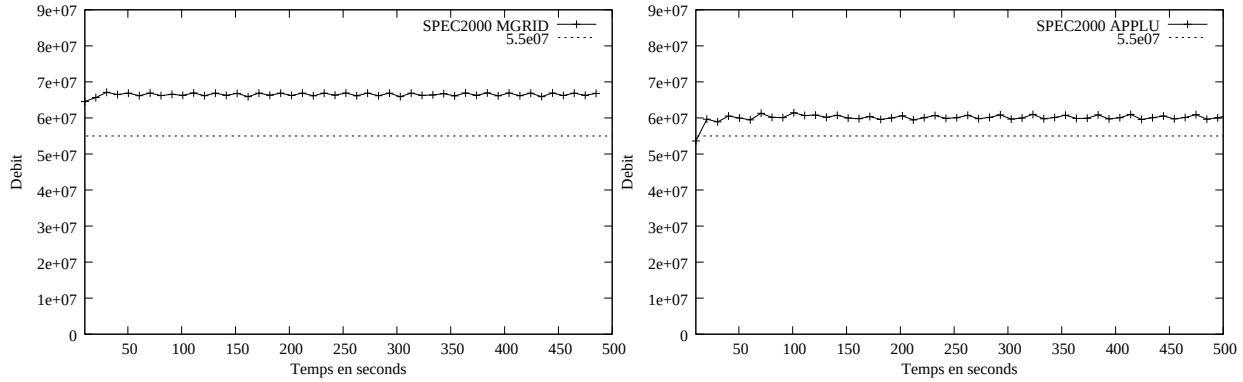

Figure A.20 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Mgrid et, à gauche, celui de l'application SPEC2000 Applu. Les deux applications sont classées comme constantes.*

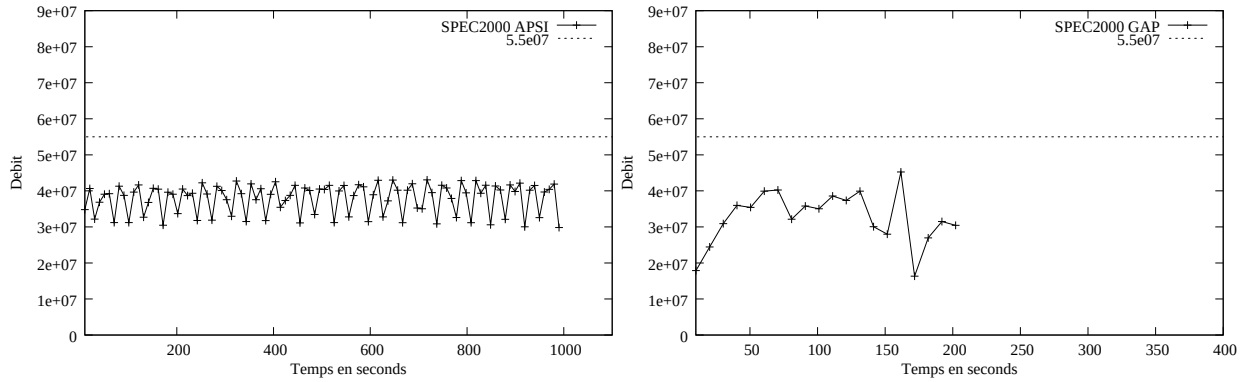

Figure A.21 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Apsi et, à gauche, celui de l'application SPEC2000 Gap. Les deux applications sont classées comme constantes.*

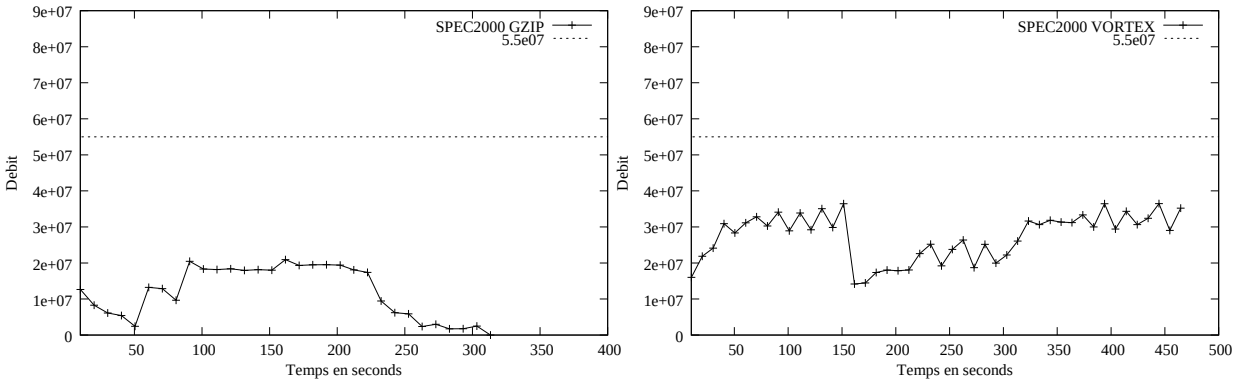

Figure A.22 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Gzip et, à gauche, celui de l'application SPEC2000 Vortex. Les deux applications sont classées comme constantes.*

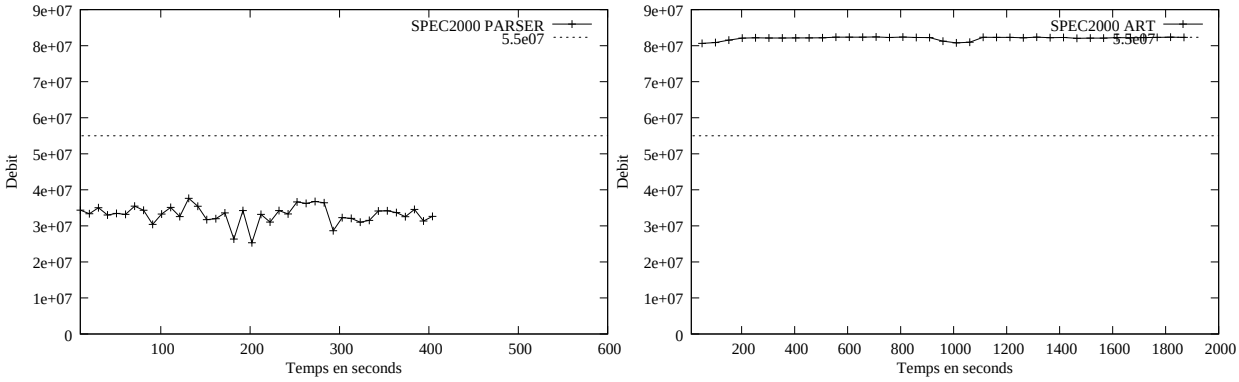

Figure A.23 ***Biprocesseur Intel Pentium 4 :*** *À droite, nous avons le comportement de l'application SPEC2000 Parser et, à gauche, celui de l'application SPEC2000 Art. Les deux applications sont classées comme constantes.*

Annexe B

Tableau d’association des compteurs matériels pour les architectures processeurs Pentium 4 et Xeon

Compteur		CCCR	ESCR	
MSR_BPU_COUNTER0	0	MSR_BPU_CCCR0	MSR_BSU_ESCR0	7
			MSR_FSB_ESCR0	6
			MSR_MOB_ESCR0	2
			MSR_PMH_ESCR0	4
			MSR_BPU_ESCR0	0
			MSR_IS_ESCR0	1
			MSR_ITLB_ESCR0	3
			MSR_IX_ESCR0	5
MSR_BPU_COUNTER1	1	MSR_BPU_CCCR1	MSR_BSU_ESCR0	7
			MSR_FSB_ESCR0	6
			MSR_MOB_ESCR0	2
			MSR_PMH_ESCR0	4
			MSR_BPU_ESCR0	0
			MSR_IS_ESCR0	1
			MSR_ITLB_ESCR0	3
			MSR_IX_ESCR0	5
MSR_BPU_COUNTER2	2	MSR_BPU_CCCR2	MSR_BSU_ESCR1	7
			MSR_FSB_ESCR1	6
			MSR_MOB_ESCR1	2
			MSR_PMH_ESCR1	4
			MSR_BPU_ESCR1	0
			MSR_IS_ESCR1	1
			MSR_ITLB_ESCR1	3
Tournez la page				

Suite ...				
Compteur		CCCR	ESCR	
			MSR_IX_ESCR1	5
MSR_BPU_COUNTER3	3	MSR_BPU_CCCR3	MSR_BSU_ESCR1	7
			MSR_FSB_ESCR1	6
			MSR_MOB_ESCR1	2
			MSR_PMH_ESCR1	4
			MSR_BPU_ESCR1	0
			MSR_IS_ESCR1	1
			MSR_ITLB_ESCR1	3
			MSR_IX_ESCR1	5
MSR_MS_COUNTER0	4	MSR_MS_CCCR0	MSR_MS_ESCR0	0
			MSR_TBPU_ESCR0	2
			MSR_TC_ESCR0	1
MSR_MS_COUNTER1	5	MSR_MS_CCCR1	MSR_MS_ESCR0	0
			MSR_TBPU_ESCR0	2
			MSR_TC_ESCR0	1
MSR_MS_COUNTER2	6	MSR_MS_CCCR2	MSR_MS_ESCR1	0
			MSR_TBPU_ESCR1	2
			MSR_TC_ESCR1	1
MSR_MS_COUNTER3	7	MSR_MS_CCCR3	MSR_MS_ESCR1	0
			MSR_TBPU_ESCR1	2
			MSR_TC_ESCR1	1
MSR_FLAME_COUNTER0	8	MSR_FLAME_CCCR0	MSR_FIRM_ESCR0	1
			MSR_FLAME_ESCR0	0
			MSR_DAC_ESCR0	5
			MSR_SAAT_ESCR0	2
			MSR_U2L_ESCR0	3
MSR_FLAME_COUNTER1	9	MSR_FLAME_CCCR1	MSR_FIRM_ESCR0	1
			MSR_FLAME_ESCR0	0
			MSR_DAC_ESCR0	5
			MSR_SAAT_ESCR0	2
			MSR_U2L_ESCR0	3
MSR_FLAME_COUNTER2	10	MSR_FLAME_CCCR2	MSR_FIRM_ESCR1	1
			MSR_FLAME_ESCR1	0
			MSR_DAC_ESCR1	5
			MSR_SAAT_ESCR1	2
			MSR_U2L_ESCR1	3
MSR_FLAME_COUNTER2	11	MSR_FLAME_CCCR3	MSR_FIRM_ESCR1	1
			MSR_FLAME_ESCR1	0
			MSR_DAC_ESCR1	5
			MSR_SAAT_ESCR1	2
Tournez la page				

Suite ...				
Compteur		CCCR	ESCR	
			MSR_U2L_ESCR1	3
MSR_IQ_COUNTER0	12	MSR_IQ_CCCR0	MSR_CRU_ESCR0	4
			MSR_CRU_ESCR2	5
			MSR_CRU_ESCR4	6
			MSR_IQ_ESCR0	0
			MSR_RAT_ESCR0	3
			MSR_SSU_ESCR0	2
			MSR_ALF_ESCR0	1
MSR_IQ_COUNTER1	13	MSR_IQ_CCCR1	MSR_CRU_ESCR0	4
			MSR_CRU_ESCR2	5
			MSR_CRU_ESCR4	6
			MSR_IQ_ESCR0	0
			MSR_RAT_ESCR0	3
			MSR_SSU_ESCR0	2
			MSR_ALF_ESCR0	1
MSR_IQ_COUNTER2	14	MSR_IQ_CCCR2	MSR_CRU_ESCR1	4
			MSR_CRU_ESCR3	5
			MSR_CRU_ESCR5	6
			MSR_IQ_ESCR1	0
			MSR_RAT_ESCR1	2
			MSR_ALF_ESCR1	1
MSR_IQ_COUNTER3	15	MSR_IQ_CCCR3	MSR_CRU_ESCR1	4
			MSR_CRU_ESCR3	5
			MSR_CRU_ESCR5	6
			MSR_IQ_ESCR1	0
			MSR_RAT_ESCR1	2
			MSR_ALF_ESCR1	1
MSR_IQ_COUNTER4	16	MSR_IQ_CCCR4	MSR_CRU_ESCR0	4
			MSR_CRU_ESCR2	5
			MSR_CRU_ESCR4	6
			MSR_IQ_ESCR0	0
			MSR_RAT_ESCR0	3
			MSR_SSU_ESCR0	2
			MSR_ALF_ESCR0	1
MSR_IQ_COUNTER5	17	MSR_IQ_CCCR5	MSR_CRU_ESCR1	4
			MSR_CRU_ESCR3	5
			MSR_CRU_ESCR5	6
			MSR_IQ_ESCR1	0
			MSR_RAT_ESCR1	2
			MSR_ALF_ESCR1	1

B – Tableau d’association des compteurs matériels pour les architectures processeurs Pentium 4 et Xeon

Bibliographie

[1] AMD-760 MP chipset overview. Order number 24229. http ://www.amd.com/.

[2] AMD Opteron processor data sheet. Order number 23932. http ://www.amd.com/.

[3] HP Labs : perfmon hardware counter.
http ://www.hpl.hp.com/research/linux/perfmon/.

[4] IA-32 Intel Architecture Optimization - Reference Manual. Order number 248966. http ://www.intel.com/.

[5] Intel E7501 chipset memory controller hub (MCH) datasheet. Order number 251927. http ://www.intel.com/.

[6] VTune Performance Analyzer. http ://developer.intel.com/vtune/.

[7] IA-32 Intel Architecture Software Developer's Manual, 2001. Order number 245470–245472. http ://www.intel.com/.

[8] AMD Athlon Processor x86 Code Optimization Guide, 2002. Order number 22007. http ://www.amd.com/.

[9] Intel Itanium 2 processor : Reference manual, 2002. Order number 251110-001. http ://www.intel.com/.

[10] AMD64 Architecture Programmer's Manual Volume 2 : System Programming, 2003. Order number 24593. http ://www.amd.com/.

[11] BIOS and kernel developer's guide for AMD Athlon 64 and AMD Opteron processors, march 2004. Order number 26094. http ://www.amd.com/.

[12] T. Aivazian and al. Linux kernel 2.4 internals. In *The Linux Documentation Project*, August 2002. http ://www.tldp.org/LDP/lki/index.html.

[13] K. J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[14] A. Malony B. Mohr, D. Brown. TAU : A Portable Parallel Program Analysis Environment for pC++. In *Proceedings of CONPAR 94 - VAPP VI, University of Linz, Austria, LNCS 854*, pages 29–40, September 1994.

[15] T. Baer. Lperfex : A hardware performance monitor for linux/ia32 systems. http ://www.osc.edu/ troy/lperfex/.

[16] R. Bell, A. D. Malony, and S. Shende. Paraprof : A portable, extensible, and scalable tool for parallel performance profile analysis. In *International Conference on Parallel and Distributed Computing, Klagenfurt, Austria*, 2003.

[17] Erik Berg and Erik Hagersten. SIP : Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.

[18] R. Berrendorf and B. Mohr. PCL - The Performance Counter Library : A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2), 2003. http ://www2.inf.fh-bonn-rhein-sieg.de/ rberre2m/PCL/.

[19] R. Berrendorf and H. Ziegler. PCL – The Performance Counter Library : A Common Interface to Access Hardware Performance Counters on Microprocessors. In *Internal Report FZJ-ZAM-IB-9816, Forschungszentrum Julich.*, 1998.

[20] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 2nd Edition*. O'Reilly, December 2002. ISBN : 0-596-00213-0.

[21] T. Bray. A program for benchmarking the performance of unix filesystems, 1990. http ://www.textuality.com/bonnie/.

[22] S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. In *International Journal of High-Performance and Supercomputer Applications*, volume 14, pages 189–20, 2000.

[23] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *The Proc. of SC 2000 : High Performance Networking and Computing Conference, Dallas TX*, November 2000.

[24] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *SC2000, Supercomputing 2000, Dallas*, November 2000.

[25] F. Cappello, O. Richard, and D. Etiemble. Investigating the performance of two programming models for clusters of smp pcs. In *Proc. of the 6th Int. Symposium on High Performance Computer Architecture Conference, Toulouse, France*, pages 349–359, January 2000.

[26] A. Carissimi. *Le noyau exècutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, nov 1999.

[27] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2001.

[28] Yen-Kuang Chen, Rainer Lienhart, Eric Debes, Matthew J. Holliman, and Minerva M. Yeung. The impact of smt/smp designs on multimedia software engineering - a workload analysis study. In *4th International Symposium on Multimedia Software Engineering (ISMSE 2002)*, pages 336–343. IEEE Computer Society, 2002. ISBN : 0-7695-1857-5.

[29] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on multiprocessors. Technical report, University of Rochester, 1991.

[30] Jeffrey Dean, James Hicks, Carl Waldspurger, William Weihl, and George Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[31] Jeffrey Dean, Carl Waldspurger, and William Weihl. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-Directed Compilation*, October 13 1998.

[32] L. DeRose. The hardware performance monitor toolkit. In *Procedings of Euro-Par, Manchester, United Kingdom.*, pages 122–131, August 2001.

[33] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. SIGMA : A simulator infrastructure to guide memory analysis, 2002. In Supercomputing, Nov. 2002.

[34] L. DeRose, T. Hoover, and J.K. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. In *IPDPS 2001 International Parallel and Distributed Processing Symposium 2001*, April 2001.

[35] L. DeRose and A. Reed. Svpablo : A multi-language architecture independent performance analysis system. In *Proc. of the International Conference on Parallel Processing (ICPP'99), Fukushima (Japan), September.*, pages 311–318, 1999.

[36] Y. Diao, S. Parekh J.L. Hellerstein, and J.P. Bigus. Managing web server performance with autotine agents. *IBM System Journal*, 42(1) :136–149, 2003. http ://www.research.ibm.com/journal/.

[37] Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh. Optimizing quality of service using fuzzy control. In Metin Feridun, Peter G. Kropf, and Gilbert Babin, editors, *Management Technologies for E-Commerce and E-Business Applications, 13th IFIP/IEEE International Workshop on Distributed Systems : Operations and Management, DSOM 2002, Montreal, Canada, October 21-23, 2002, Proceedings*, volume 2506 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2002.

[38] K. Diefendorff. Power4 focuses on memory bandwidth. In *MicroProcessor Report*, pages 11–17, 1999.

[39] P. Dinda, T. Gross, R. Karrer, B. Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the remos system. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing.*, 2001.

[40] M.J. Domeika, C.W. Roberson, E.W. Page, and G.A. Tagliarini. Adaptive resonance theory 2 neural network approach to star field recognition. In Steven K. Rogers and Dennis W. Ruck, editors, *SPIE–The International Society for Optical Engineering*, volume 2760, pages 589–596, 1996.

[41] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. Technical report, 2003.

[42] Ralf S. Engelschall. Gnu pth - the gnu portable threads. Technical report, 2003.

[43] D. Bailey et al. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3) :63–73, Fall 1991.

[44] D. Bailey et al. The nas parallel benchmarks (94). In *RNR Technical Report RNR-94-007 - NASA Ames Research Center*, 1994.

[45] K Olukotun et al. The case for a single-chip multiprocessor. In *Proc. International Conference ASPLOS-VII*, ACM Press, New York, pages 2–11, 1996.

[46] Xue Liu et al. Online response time optimization of apache web server. In K. Wehrle (Eds.) K. Jeffay, I. Stoica, editor, *Quality of Service - IWQoS 2003 : 11th International Workshop, Berkeley, CA, USA*, volume 2707 / 2003, pages 461–478. Springer-Verlag Heidelberg, June 2-4 2003. ISSN : 0302-9743.

[47] Michael J. Flynn. Some computer organizations and their effectivenessr. *IEEE Transactions on Computers*, C-21(9) :948–960, sep 1972.

[48] Manoj Franklin and Gurindar S. Sohi. Arb : A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5) :552–571, 1996.

[49] M. K. Gardner, W. Feng, M. Broxton, A. Engelhart, and G. Hurwitz. Magnet : A tool for debugging, analysis and adaptation in computing systems. In *Proc. of the 3rd IEEE/ACM International International Symposium on Cluster Computing and the Grid (CCGrid 2003) - Tokyo Japan*, pages 12–15, May 2003.

[50] Mark K. Gardner, Michael Broxton, Adam Engelhart, and Wu chun Feng. Muse : A software oscilloscope for clusters and grids. In *Proc. of the 17th IEEE Internation Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 22–26, April 2003.

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

[52] A. Goldberg and J. Hennessy. MTOOL : A method for isolating memory bottlenecks in shared memory multiprocessor programs. In *Proceeding of the International Conference on Parallel Processing*, pages 251–257, 1991.

[53] Portland Group. *PGI User's Guide*. Portland Group, United States of America, 2000.

[54] Cyril Guilloud. *Traçage flexible d'exécutions de programmes parallèles.* Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, fev 2004.

[55] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 120–132. ACM Press, 1991.

[56] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Transactions on Computers*, 30(9) :75–85, 1997.

[57] Lance Hammond and Kunle Olukotun. Considerations in the design of hydra : A multiprocessor-on-a-chip microarchitecture. Technical report, 1998.

[58] D. Heller. Rabbit : A performance counters library for intel processors and linux. http ://www.scl.ameslab.gov/Projects/Rabbit/.

[59] Stephen A. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. Ph.d. thesis, Stanford University, EUA, feb 1998.

[60] Hewlett-Packard. Netperf, 2003.
http ://www.netperf.org/.

[61] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly efficient gang scheduling implementation. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. IEEE Computer Society, 1998.

[62] Cray Research Inc. Unicos performance utilities reference manual. Technical report, Cray Research Publication, 1994.

[63] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, and Nicolai Kosche. Memory profiling using hardware counters. In *Proceedings of the ACM/IEEE SC2003 Conference*, pages 17–30, Phoenix, Arizona, November 15 - 21 2003. Sun Microsystems, Inc., Menlo Park, California.

[64] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks. Technical Report NAS-99-011, NASA Ames Research Center, 1999. H. Jin, M. Frumkin and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks.

[65] R. Jin and G. Agrawal. Performance prediction for random write reductions : a case study in modeling shared memory programs. In *Proc. of the 2002 ACM SIGMETRICS int. conf. on Measurement and modeling of computer systems, New York, USA*, pages 117–128, 2002.

[66] James T. Kajiya. The rendering equation. In *SIGGRAPH*, pages 143–150, 1986.

[67] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip : a dual-core multithreaded processor. In *Micro IEEE*, volume 24, Issue : 2, pages 40–47. IEEE Computer Society, 2004. ISSN : 0272-1732.

[68] Philippe Kloos and Philippe Blaise. OpenMP and MPI programming with a CG algorithm. In *Second European Workshop on OpenMP, Edinburgh, Scotland, U.K.*, September 14-15th 2000.

[69] Géraud Krawezik and Franck Cappello. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *15th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, San Diego, USA*, June 2003.

[70] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9) :866–880, 1999.

[71] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 226–236. ACM Press, 1990.

[72] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis, using hardware counters. International Conference on Parallel and Distributed Computing Systems, August 2001.

[73] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The papi cross-platform interface to hardware performance counters. In *Department of Defense Users' Group Conference Proc., Biloxi, Mississippi*, pages 18–21, June 2001.

[74] Robert Love. Interactive kernel performance. In *Proceedings of the Linux Symposium, Ottawa, Ontario, Canada*, July 2003. http ://www.linuxsymposium.org/.

[75] Peter S. Magnusson and al. SimICS/sun4m : A virtual workstation.

[76] Jussi Maki. A free aix performance monitor. In *Joint G.U.I.D.E / SHARE Europe Conference*, 10–13 October - Vienna, Austria 1994. Helsinki University of Technology - Computing Centre - Otakaari 1 - FIN-02150 - Espoo - FINLAND – http ://staff.csc.fi/ jmaki/monitor-paper.html.

[77] Jussi Maki. Power2 hardware performance monitor tools. In *CSC - Center for Scientific Computing - PL 405, FIN-02101 Espoo*, 27 November 1995. http ://staff.csc.fi/ jmaki/rs2hpm-paper/.

[78] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal.*, 06(01), February 14 2002. ISSN 1535-766X.

[79] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy : Analyzing memory system bottlenecks in programs. In *Proc. 1992 ACM SIGMETRICS Conference*, pages 1 – 12, May 1992.

[80] John M. May. Mpx : Software for multiplexing hardware performance counters. In *Proc. of the IDPS 2001 International Parallel and Distributed Processing Symposium*, april 2001.

[81] J. D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. In *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter.*, December 1995.

[82] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. In *IEEE Computer.*, volume 28, pages 37–46, 1995. citeseer.nj.nec.com/miller95paradyn.html.

[83] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 67–77. ACM Press, 1996.

[84] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. ISBN : 1-56592-115-ISBN : 1-56592-115-11. O'Reilly, 1996.

[85] D. Nikolopoulos, E. Polychronopoulos, and T. Papatheodorou. Enhancing the performance of autoscheduling in distributed shared memory multiprocessors, 1998. EuroPar'98.

[86] D. S. Nikolopoulos and C. D. Polychronopoulos. Adaptive scheduling under memory pressure on multiprogrammed clusters. In *Proc. of the Second IEEE/ACM Int. Symp. on Cluster Computing and the Grid (CCGrid 2002), Berlin, Germany, (Best Paper Award)*, May 2002.

[87] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11. ACM Press, 1996.

[88] Omni OpenMP. Omni openmp compiler project. http ://phase.hpcc.jp/Omni/.

[89] David A. Patterson and John L. Hennessy. *Computer architecture : a quantitative approach*. Morgan Kaufmann Publishers Inc., 1990.

[90] M. Pattersson. Linux x86 performance-monitoring counters driver. http ://user.it.uu.se/∼mikpe/linux/perfctr/.

[91] A. L. Peressini and Francis E. Sullivan. *The mathematics of nonlinear programming*. Springer-Verlag New York, Inc., 1988.

[92] Eleftherios D. Polychronopoulos and Theodore S. Papatheodorou. Scheduling user-level threads on distributed shared-memory multiprocessors. In *European Conference on Parallel Processing*, pages 358–368, 1999.

[93] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis : The pablo performance analysis environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[94] R. L. Ribler, H. Simitci, and D. A. Reed. The autopilot performance-directed adaptive control system. In *Future Generation Computer Systems.*, volume 18, pages 175–187, 2001.

[95] Olivier Richard. *Contribution à l'étude des grappes de serveurs multiprocesseurs.* PhD thesis, Université de Paris-Sud U.F.R Scientifique d'Orsay, novembre 1999.

[96] Luiz De Rose and Felix Wolf. CATCH - a call-graph based automatic tool for capture of hardware performance metrics for mpi and openmp applications. In Burkhard Monien and Rainer Feldmann, editors, *Proc. 1992 ACM SIGMETRICS Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 167–176. Springer, 2002. ISBN : 3-540-44049-6.

[97] P. Roth and B. Miller. Deepstart : A hybrid strategy for automated performance problem searches. In *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*, volume 2400 of *Lecture Notes in Computer Science*, pages 86–95. Springer, 2002.

[98] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers, 2nd Edition*. ISBN : 0-596-00008-1. O'Reilly, June 2001. http ://www.oreilly.com/catalog/linuxdrive2/.

[99] E. Shaffer, S.Whitmore, B. Schaeffer, , and D. Reed. Virtue : Immersive performance visualization of parallel and distributed applications. In *IEEE Computer*, pages 44–51, December 1999.

[100] W. Smith. A framework for control and observation in distributed environments. In *Technical Report NASA Advanced Supercomputing Division, NASA Ames Research Center NAS-01-006, July.*, 2001.

[101] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI : The Complete Reference*. MIT Press, 1995.

[102] SPEC2000. Standard performance evaluation corporation. spec cpu2000 benchmarks. December 1999. http ://www.specbench.org/osg/cpu2000.

[103] OpenMP Specifications. Simple, portable, scalable smp programming. http ://www.openmp.org/.

[104] Brinkley Sprunt. Managing the Complexity of Performance Monitoring Hardware. *Submitted to International Journal of High Performance Computing Applications (IJHPCA).*, 2004.

[105] Andrew S. Tanenbaum. *Distributed Operating Systems*. ISBN : 0132199084. Prentice Hall ; 1st edition, 1994.

[106] Andrew S. Tanenbaum. *Systèmes d'exploitation*. ISBN : 2-7440-7002-5. Pearson Education France, 2003.

[107] Hong-Linh Truong and Thomas Fahringer. SCALEA : A Performance Analysis Tool for Distributed and Paralle Programs. In *European Conference on Parallel Processing*, pages 75–85, 2002.

[108] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, page 26, September 27 - October 01 2003.

[109] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading : Maximizing on-chip parallelism. In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[110] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice : Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecturem*, pages 191–202, 1996.

[111] Theo Ungerer, Borut Robic, and Jurij Silc. Multithreaded processors. *The Computer Journal*, 45(3) :320–348, 2002.

[112] R. F. Van-Der-Wijngaart and P. Wong. Nas parallel benchmarks i/o version 2.4. In *Technical Report NAS-03-002, NASA Ames Research Center*, 2003. Computer Sciences Corporation - NASA Advanced Supercomputing (NAS) - Moffett Field, CA 94035-1000.

[113] T. N. Vijaykumar, Sridhar Gopal, James E. Smith, and Gurindar Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12) :1305–1317, 2001.

[114] E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks. Power2 hardware performance monitoring. *IBM Journal of Research and Development*, 38(5) :545–554, September 1994.

[115] F. Wolf and B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processin*, 49 :421–439, 2003.

[116] F. Wolf, B. Mohr, J. Dongarra, and S. Moor. Efficient pattern search in large traces through successive refinement. In *Euro-Par 2004, Parallel Processing, International Euro-Par Conference Paderborn, Italy*, Lecture Notes in Computer Science. Springer, August 31 - Sept. 3 2004.

[117] Yoshimitsu Yanagawa, Luong Dinh Hung, Chitaka Iwama, Niko Demus Barli, Shuichi Sakai, and Hidehiko Tanaka. Complexity analysis of a cache controller for speculative multithreading chip multiprocessors. *High Performance Computing - HiPC*, pages 393–404, 2003.

[118] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing*, 1996.