Etude de méthodes et mécanismes pour un accès transparent et efficace aux données dans un système multiprocesseur sur puce

P. Guironnet de Massas

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

ISBN 978-2-84813-145-0

THESE

pour obtenir le grade de

DOCTEUR DE L'Institut polytechnique de Grenoble

Spécialité : Informatique

préparée au laboratoire TIMA

dans le cadre de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement

par

Pierre Guironnet de Massas

le 12 novembre 2009

Étude de méthodes et mécanismes pour un accès transparent et efficace aux données dans un système multiprocesseur sur puce

DIRECTEUR DE THESE

Frédéric Pétrot

JURY

M. Jean-François Méhaut, Président
M. Alain Greiner, Rapporteur
M. Olivier Temam, Rapporteur
M. Frédéric Pétrot, Directeur de thèse
M. Olivier Franza, Examinateur
M. Thierry Collette, Examinateur

Étude de méthodes et mécanismes pour un accès transparent et efficace aux données dans un système multiprocesseur sur puce

Pierre Guironnet de Massas

20 novembre 2009

Remerciements

Je remercie en premier lieu mon directeur de thèse Frédéric Pétrot, chef de l'équipe SlS du laboratoire TIMA, pour le temps qu'il a consacré au suivi de mes travaux et à son expertise qui m'a souvent aidée à résoudre les problèmes rencontrés.

Je remercie également les membres du jury. Messieurs Olivier Temam, directeur de recherche à l'INRIA et Alain Greiner, directeur du département SoC au LIP6 d'avoir accepté d'être rapporteurs de ces travaux. Messieurs Thierry Collette, chef de service au CEA LIST et Olivier Franza, ingénieur chez Intel d'avoir été examinateurs de ma soutenance de thèse. Enfin je remercie monsieur Jean-François Méhaut d'avoir présidé le jury lors de ma soutenance de thèse.

Je tiens à remercier tous les collègues et amis qui m'ont soutenus, aidés durant ces trois années passées au laboratoire. Tout particulièrement Xavier Guérin et Patrice Gerin pour leur expertise technique sur les systèmes d'exploitation, langages de programmation et outils d'édition. Quentin Meunier pour son aide précieuse à la re-lecture d'articles, documents et manuscrit de thèse, ainsi que pour son aide sur les problèmes algorithmiques.

Je remercie également tous ceux qui m'ont consacré du temps pour les lectures et relectures du manuscrit de thèse, tout particulièrement ma compagne Amandine Brugière et mon père Eric de Massas, ainsi que Frédéric Rousseau, Paul Amblard et Olivier Muller qui m'ont permis de mener à bien ma soutenance de thèse.

Enfin, je remercie tous les membres du laboratoire TIMA qui m'ont accordé de leur temps pour partager avec moi leur expertise technique, scientifique ou administrative.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Depuis plus d'un demi-siècle, les systèmes informatiques n'ont cessé de progresser. Au début destinés à réaliser du calcul, ils ont très vite été détournés de leur usage premier pour s'intégrer dans tous les domaines de notre existence : finance, santé, loisirs, communications, transports, etc... L'homme demande toujours plus à ces systèmes, plus de vitesse, plus de puissance, un meilleur rendement énergétique, plus de fonctionnalités. Si bien qu'en l'espace d'une génération humaine nous sommes passés de simples calculateurs arithmétiques à des systèmes d'une puissance de calcul inimaginable alors, et offrant des possibilités applicatives très variées. Le cœur de ces systèmes est le couple processeur-mémoire auquel nous apporterons toute notre attention.

Les technologies d'intégration des systèmes informatiques ont permis d'intégrer toujours plus de transistors dans une même puce. La diminution de la taille des circuits permet également l'augmentation de leur fréquence de fonctionnement. Nous avons été témoins cette dernière décennie d'une évolution majeure dans la conception des processeurs et des systèmes informatiques : la généralisation d'architectures multiprocesseur.

Afin d'obtenir de meilleures performances, les architectes privilégient l'intégration de nombreux processeurs à l'usage d'un coeur unique très complexe. Au regard de la puissance de calcul, la complexité moindre de ces processeurs est largement compensée par l'exploitation du parallélisme à l'exécution.

Il y a principalement trois causes à cette évolution : une architecture des processeurs complexe qui conduit à des consommations d'énergie inacceptables, des limitations physiques à la montée en fréquence des systèmes et une capacité d'intégration en augmentation presque constante.

Les deux premières causes concernent surtout le domaine du calcul haute performance où la complexité des processeurs n'a cessé de croître depuis leur invention. Les architectes ont imaginé et implanté des techniques complexes tel que le *pipelining*, l'exécution dans le désordre des instructions ou encore le calcul superscalaire. De nouvelles améliorations de l'architecture offraient un gain faible vis-à-vis de la complexité de la solution et de l'effort fourni.

La montée en fréquence de ces systèmes n'est plus possible à cause essentiellement des fuites de courants associés [Naf06]. Pour s'affranchir de cette limitation, les concepteur multiplient le nombre de processeurs dans une même puce afin d'exécuter en parallèle plusieurs tâches.

La troisième cause concerne également le domaine des systèmes embarqués. La capacité à intégrer de multiples processeurs dans une même puce permet de faire face aux demandes des applications embarquées gourmandes en puissance de calcul. En effet, l'essor de la téléphonie mobile, des assistants personnels (PDA) et des technologies multimédia en général

a créé de nouveaux besoins : décodage vidéo, jeux interactifs en trois dimensions, décodage audio numérique, etc...

Des systèmes complexes

Si dans certains cas l'architecture des processeurs a été simplifiée, d'une manière générale le système a gagné en complexité. Cette évolution concerne également le logiciel. L'usage de mécanismes d'abstraction tels que les langages interprétés par des machines virtuelles, le recours à des techniques de génération de code ou tout simplement l'intégration d'un système d'exploitation ont simplifié le développement des applications. Néanmoins, cette simplification s'est faite au dépend de la simplicité de l'ensemble.

La première conséquence est une abstraction totale de l'architecture matérielle et l'impossibilité qu'a le programmeur d'exploiter facilement ces caractéristiques particulières. Nos travaux portent un intérêt tout particulier à cet aspect de l'évolution des systèmes.

Contraintes supplémentaires

Les systèmes embarqués ont profité de ces évolutions mais pâtissent également de l'augmentation de la complexité. A ceci s'ajoutent des contraintes qui leurs sont toutes particulières, comme la consommation d'énergie ou la surface de silicium restreinte. Ceci conduit à l'intégration du plus grand nombre possible de composants et périphériques dans la même puce. Nous y retrouvons maintenant en plus des processeurs : des mémoires, des périphériques d'entrée sortie ou des décodeurs vidéo. Ces puces deviennent de véritables systèmes sur puce ou SoC (*System on Chip*).

Les solutions du futur, tendances actuelles

D'après les prévisions ITRS [ITR07], il sera possible en 2020 d'intégrer 128 processeurs dans une même puce (cf. figure 1.1). Les architectes et concepteurs d'aujourd'hui proposent deux tendances à l'évolution des systèmes. La première tendance est la conception de systèmes très hétérogènes où chaque unité de calcul est optimisée pour une tâche précise [KTJR05]. Ils permettent une exécution optimisée des tâches logicielles ou matérielles d'un système, offrant ainsi une exécution rapide à moindre coût énergétique. Malgré cela, l'exploitation de tels systèmes est une tâche ardue, et leur complexité rebute bon nombre de programmeurs et de décideurs.

La deuxième tendance est l'intégration d'un grand nombre de processeurs identiques, simples et génériques dans une même puce avec une hiérarchie mémoire conçue pour simplifier l'accès aux données. Le défaut de cette solution est que pour une tâche spécifique les performances obtenues seront probablement en deçà de ce que peut fournir une architecture dédiée et spécifique à cette même tâche. Néanmoins, l'évolution rapide des besoins ainsi que la pluralité des tâches exécutées par un même système requiert une solution flexible tant en terme d'évolutivité que de capacité. De plus, la simplicité du modèle de programmation offert par ces architectures en font une solution idéale pour une exploitation efficace des ressources avec un effort et un coût minimum.

Dans le cadre de cette thèse nous sommes convaincus que cette deuxième tendance est la plus pertinente et nous parions qu'à l'avenir elle marquera l'évolution des systèmes.

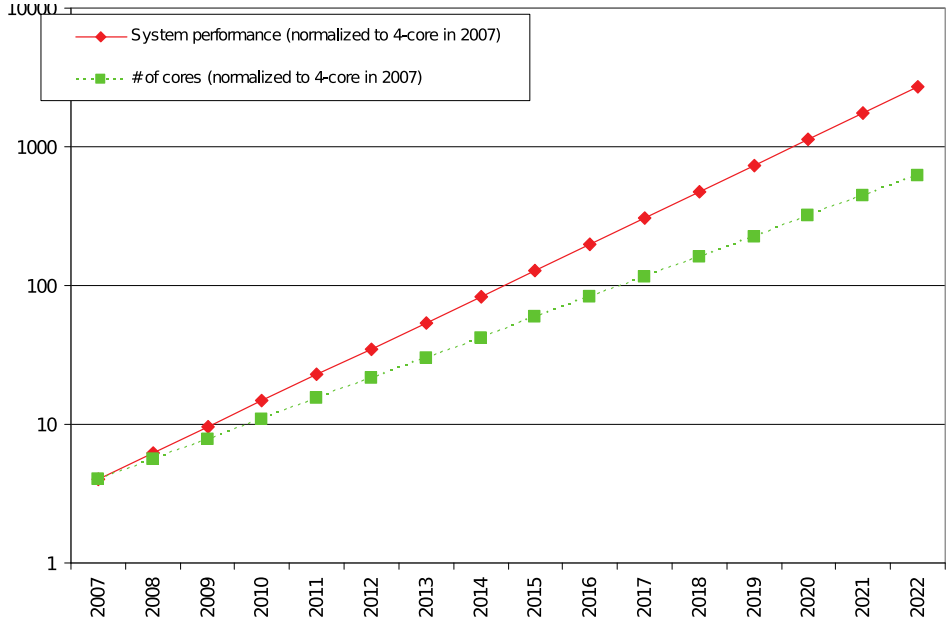


FIG. 1.1 – Perspectives d'évolution du nombre de cœurs dans une puce et de la puissance de calcul (source ITRS 07 [ITR07]).

Problème général et objectifs de la thèse

Nos travaux se penchent sur l'aspect mémoire de ces architectures. Puisque l'avancement des tâches est conditionné par l'accès aux données et aux instructions, l'accès à la mémoire est un point critique de tout système informatique.

Ce problème peut être vu sous plusieurs angles différents : la latence des accès, la bande passante occupée, la complexité du modèle de programmation et la consommation d'énergie.

Nous abordons cette problématique en deux parties : Dans un premier temps nous présenterons une étude sur les caches et les protocoles de cohérence mémoire. Nous présenterons également une méthode pour les comparer et les évaluer. Dans un deuxième temps, nous décrirons une solution innovante pour déplacer les données sur le système. L'objectif est d'améliorer les latences d'accès tout en fournissant un accès transparent aux données.

Plan de la thèse

Dans le deuxième chapitre de cette thèse nous présentons les problématiques auxquelles ce travail apportera une réponse.

Le troisième chapitre est un état de l'art en matière de conception des mémoires cache et des protocoles de cohérence mémoire. On y présente également toutes les notions nécessaires à la compréhension de nos travaux. Le quatrième chapitre présente notre contribution liée aux problèmes de cohérence mémoire et de l'utilisation de certains protocoles dans le cadre des systèmes embarqués. Cette contribution sera étayée par des expérimentations et des résultats concluants.

Cette première partie a soulevé de nouveaux problèmes liés aux méthodes de comparaison et d'évaluation des protocoles de cohérence mémoire. Le chapitre suivant dresse un état de l'art lié à cette problématique. Le sixième chapitre développe une méthode d'évaluation

innovante basée sur la simulation précise au niveau du cycle d'horloge. Les expérimentations et résultats sont mis en comparaison avec ce qui a été obtenu dans le chapitre quatre.

Certaines limitations de l'utilisation des caches vont être soulevées dans la problématique. Dans le septième chapitre nous détaillons les techniques complémentaires basées sur la migration des données dans une puce. Cet état de l'art présente également des techniques de coopération des caches, qui s'intègrent parfaitement dans ce contexte. Les chapitre huit et neuf décrivent une nouvelle solution à la migration des données. Entièrement gérée en matériel, elle vient en complément de l'usage des caches et des mémoires cohérentes.

Dans le chapitre 10 nous présentons quelques limitations de ces travaux et des pistes de recherche pour les travaux futurs.

Le dernier chapitre reprend les questions posées dans la problématique et fait état des réponses apportées dans nos diverses contributions. Il résume l'ensemble de nos résultats et présente les conclusions de notre travail de thèse.

Chapitre 2

Problématique

Nous présentons dans ce chapitre les différents problèmes auxquels nous apporterons une réponse. Nous présentons d'abord le problème général afin de situer le contexte de nos travaux. Puis, nous présenterons en détail les sous-problèmes autour desquels s'articulent les différentes parties de ce manuscrit.

2.1 L'accès à la mémoire, un point clé

Comme nous l'avons vu en introduction, le couple processeur-mémoire est le cœur de tout système informatique. La mémoire contient toutes les instructions exécutées et toutes les données accédées par un ou plusieurs processeurs (cf. fig. 2.1).

Toute instruction exécutée requiert donc au minimum un accès à la mémoire. Puisqu'elles sont exécutées dans l'ordre d'apparition du programme tant qu'aucun branchement ou déroutement n'a été effectué, il est très facile d'anticiper leur chargement à l'aide de composants spécialisés.

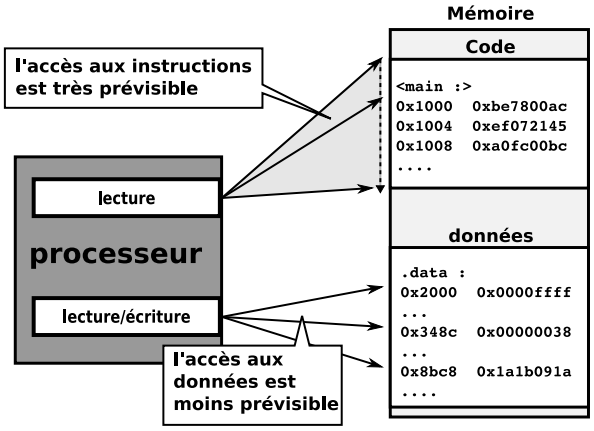


FIG. 2.1 – Couple processeur-mémoire : l'accès aux instructions et aux données n'ont pas les mêmes caractéristiques de localité temporelle ou spatiale.

Toutefois, à peu près une instruction sur cinq lit ou écrit une donnée en mémoire. Les adresses accédées dépendent du type de donnée (variable locale, variable dynamique, donnée statique), et de leur emplacement parmi un ensemble (case d'un tableau, champ d'une structure, variable spécifique). L'ordre des accès à ces données dépend du comportement

dynamique du programme, bien qu'il ne soit pas aléatoire, il est bien moins prévisible que l'ordre des accès aux instructions.

Les architectes et les concepteurs ont proposé d'innombrables solutions pour optimiser l'accès aux données, mais l'évolution des technologies les remet en cause à chaque nouvelle génération. En effet, l'amélioration des capacités de calcul des processeurs est bien plus rapide que la diminution des temps d'accès aux mémoires ou l'augmentation de leur débit.

L'accès aux données et à la mémoire en général constitue un point critique de tout système. Les architectes cherchent des solutions efficaces pour les problèmes d'accès à la mémoire. La notion d'efficacité varie en fonction des besoins et des contraintes. Nous définissons donc ci-dessous ce qu'est dans le cadre de nos travaux, un accès efficace à la mémoire.

2.2 Un accès efficace à la mémoire

Nous définissons l'efficacité d'un accès mémoire par trois métriques différentes : la latence, la bande passante et l'énergie mise en œuvre.

2.2.1 La latence d'accès

La latence d'accès ou temps d'accès se mesure en cycles d'horloge du processeur. Cette latence est due principalement à l'éloignement physique du banc mémoire auquel on souhaite accéder (cf. figure 2.2).

En général, l'organisation de la mémoire présente une hiérarchie. Les contraintes physiques et les coûts d'intégration imposent que les éléments placés à proximité des processeurs aient une capacité limitée. Par exemple, le banc des registres qui est accessible en un seul cycle d'horloge a une taille de quelques centaines d'octets seulement. Les caches et mémoires locales ont un accès de l'ordre du cycle d'horloge ($1 \sim 5$) mais ont une taille comprise entre quelques kilo-octets et quelques méga-octets. Les mémoires dynamiques (DRAM) dont la taille peut atteindre plusieurs giga-octets sont placées à l'extérieur de la puce et s'accèdent au mieux en une centaine de cycles d'horloge.

La distance n'est pas le seul élément qui influe sur la durée d'un accès. Le franchissement des interfaces joue également un rôle de premier ordre ; interface de bus, pont d'accès entre deux bus, contrôleurs et files d'attentes contribuent tous au temps d'accès global.

Certaines de ces interfaces permettent de changer de milieu physique ou de technologie (fibre optique), elles permettent par exemple d'accéder à un bus extérieur à la puce ou encore un lien série vers une autre carte. La latence de ces interfaces peut se mesurer en dizaines de cycles d'horloge.

L'enjeu majeur pour les architectes est de réduire la latence moyenne des accès. Une amélioration sensible de la latence moyenne aura une influence directe sur la vitesse d'exécution des applications.

Si l'espace d'adressage est réparti sur un ensemble de nœuds mémoire, les latences d'accès ne vont pas être uniformes pour toutes les données. De telles architectures sont de type NUMA (*Non Uniform Memory Access*).

2.2.2 La bande passante

La bande passante d'un lien de communication (bus, NoC, lien série) se mesure en quantité de données transférées par unité de temps.

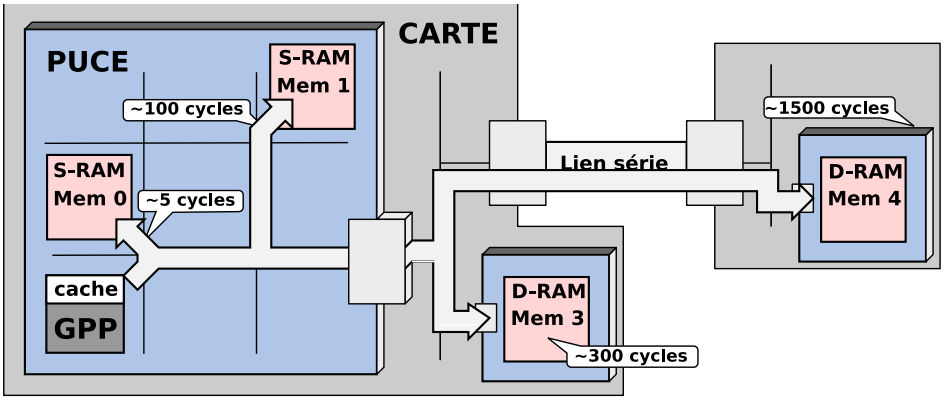


FIG. 2.2 – Latence indicative entre un processeur et des composants mémoire.

Si sur un lien de communication il n'y a qu'un seul composant capable d'émettre des requêtes, il suffit que la bande passante fournie soit supérieure au débit maximum que peut générer l'émetteur afin de garantir un temps de transfert sensiblement constant.

Si le lien de communication est partagé par plusieurs émetteurs potentiels, il est très probable que sa bande passante soit bien inférieure au débit maximum cumulé des émetteurs. Dès que la charge sur le lien dépasse un seuil critique, les latences d'accès croissent de façon exponentielle [PGJ+05]. Cette situation critique doit être évitée à tout prix sous peine de voir les performances du système s'effondrer.

Les architectes conçoivent le lien de communication en fonction des besoins, du type d'application, des performances souhaitées et des contraintes de coût. Le coût d'implantation d'un lien de communication est directement lié à la surface qu'il occupe.

Le coût en consommation de bande passante est également pris en compte lors de la conception de solutions pour améliorer l'efficacité des accès à la mémoire. Ceci s'applique aux solutions matérielles (cache, co-processeur, DMA) mais également aux solutions logicielles (flux de données, placement des variables et des structures, communications entre tâches).

2.2.3 Consommation

Les systèmes embarqués ont la vocation d'être autonomes. Par conséquent, ils dépendent bien souvent d'une source d'énergie limitée (batterie), ou de faible capacité (solaire, cinétique, électromagnétique). De fait, il est impératif d'évaluer la consommation énergétique de ces systèmes et en amont, de concevoir des solutions économiques.

Les accès à la mémoire sont très gourmands en consommation d'énergie. Lors d'un accès à la mémoire, l'énergie est consommée d'un côté par les composants mis en œuvre (contrôleur de cache, contrôleur mémoire, bancs mémoire etc.), et d'un autre côté par le lien de communication.

2.3 Un accès transparent à la mémoire

La meilleure des solutions n'a aucun intérêt si les programmeurs ne sont pas capables de l'utiliser efficacement.

Les systèmes, composés d'une architecture matérielle et de couches logicielles, deviennent de plus en plus complexes à gérer par les programmeurs. Cette complexité a deux

origines orthogonales : l'avènement de la programmation parallèle et la montée en abstraction des vues du système. Toute solution concernant l'accès aux données doit prendre en compte cette complexité et ne pas en ajouter davantage.

2.3.1 Complexité des systèmes multiprocesseurs

La tendance durable est l'intégration d'un nombre croissant de processeurs dans le système. Les écueils de la programmation d'un système multiprocesseur sont multiples :

- Paralléliser efficacement une ou plusieurs tâches. Certains algorithmes sont très difficiles à paralléliser.
- Synchroniser l'exécution parallèle des différentes tâches en évitant les interblocages, les zones critiques non protégées et toute situation de compétition au résultat imprévisible.
- Communiquer efficacement entre les différentes tâches et ne pas créer des points de congestion.
- Déboguer des erreurs qui ont bien (trop) souvent une reproductibilité aléatoire.

De plus, si la plateforme est hétérogène et possède de nombreux types de nœuds de calculs, l'exploitation efficace du parallélisme s'avère encore plus difficile.

Tout ceci rend la programmation parallèle efficace complexe. Toutefois, l'effort à fournir pour développer une application parallèle efficace reste néanmoins payant au vu du gain en performance généralement obtenu.

2.3.2 Abstraction des détails de l'architecture

De nos jours, afin de simplifier la programmation et d'améliorer la portabilité des applications, les développeurs et concepteurs ont recours à des solutions qui permettent d'abstraire les détails de l'architecture. Cela va de l'usage de langages de haut niveau et de systèmes d'exploitations, à la virtualisation des systèmes ou l'utilisation de machines virtuelles.

Le revers de la médaille est que le programmeur n'est plus en mesure de maîtriser l'ensemble des composants, outils et services mis en jeu. Il n'est plus en mesure d'appréhender finement l'interaction entre le logiciel et le matériel. Par voie de conséquence, il n'est plus en mesure d'exploiter pleinement les mécanismes offerts par l'architecture afin de produire une application pleinement optimisée.

Par exemple considérons une application décrite dans un langage interprété par une machine virtuelle. Cette application s'exécute par ailleurs au sein d'un canevas complexe de composants aux interactions multiples.

Comment le programmeur peut-il garantir dans la pratique l'utilisation à bon escient de la mémoire ? Comment peut-il s'assurer de l'efficacité des transferts de données et messages, de l'usage des contrôleurs DMA ? A-t-il un moyen de placer les données de façon à en optimiser l'accès, diminuer les défauts de pages, la distance vis-à-vis du nœud de calcul ? Ce sont des questions auxquelles il n'est pas simple de répondre et auxquelles beaucoup de programmeurs ne désirent pas répondre. En général l'objectif à atteindre est d'avoir un système fonctionnel à moindre coût de développement logiciel. La performance du système est léguée aux autres acteurs : les concepteurs du système, des composants utilisés et de l'architecture matérielle.

Les concepteurs de systèmes embarqués, à quelques exceptions près, ne sont plus épargnés par ce problème. Les appareils multimédia grand public tels que les PDA, téléphones mobiles, baladeurs mp3 ou encore les consoles de jeux vidéo, n'échappent pas au problème de la complexité.

2.3.3 Accès transparent

Dans ce contexte il est donc impératif de ne pas reporter sur le programmeur le devoir d'exploiter des caractéristiques précises de l'architecture afin d'optimiser les accès mémoire. Nous définissons un accès transparent aux données comme la possibilité de manipuler efficacement les données d'un système sans avoir à prendre en compte les détails de l'architecture.

Ceci peut être fourni par certaines couches logicielles, ou bien, par le matériel. Cette dernière solution permet de garantir l'usage effectif de la solution mise en œuvre à tous les niveaux du logiciel : système d'exploitation, *middleware*, application.

L'enjeu de toute nouvelle solution permettant d'améliorer les accès mémoire est donc multiple, il doit réduire la latence moyenne des accès, être économe en bande passante, être aussi simple que possible à exploiter et avoir un impact positif ou négligeable sur la consommation d'énergie.

2.4 Le contexte d'étude

Maintenant que les tendances d'évolution ont été montrées en introduction, que les contraintes auxquelles nous nous intéressons sont définies, nous pouvons précisément décrire notre contexte d'étude.

Nos travaux se concentrent sur des architectures multiprocesseurs, flexibles, potentiellement embarquées et simples à programmer. Par conséquent elles ont les caractéristiques suivantes :

Homogène : elles intègrent des processeurs simples et génériques (ARM, SPARC, MIPS, MicroBlaze, etc).

Caches : elles possèdent des petits caches qui sont une solution efficace et dont l'usage est transparent pour le programmeur. Leur taille est de quelques kilo-octets.

Cohérence : la cohérence des données est maintenue par le matériel, son coût d'implantation est justifié par une transparence des accès à la mémoire partagée.

Mémoire embarquée : en vue d'une intégration enfouie, toute ou partie de la mémoire est embarquée dans la puce. Elle est distribuée en bancs dans l'ensemble du système pour facilité l'intégration et limiter la congestion des accès.

Mémoire partagée : l'ensemble de l'espace d'adressage est partagé par tous les processeurs du système pour simplifier la programmation de l'ensemble.

Réseau d'interconnexion : afin de permettre un passage à l'échelle et de pouvoir intégrer plusieurs dizaines de processeurs et bancs mémoire, l'interconnexion est réalisée par un réseau point-à-point embarqué. Un tel réseau est un NoC[1][GG00].

Nœuds de calcul : les composants seront organisés en nœuds de cacul comprenant au moins une mémoire et un processeur. Ces composants communiquent au travers d'un sous-réseau local de type crossbar.

Ci-dessous, nous aborderons cette problématique de façon détaillée sous trois angles différents mais complémentaires.

[1] *Network on Chip*

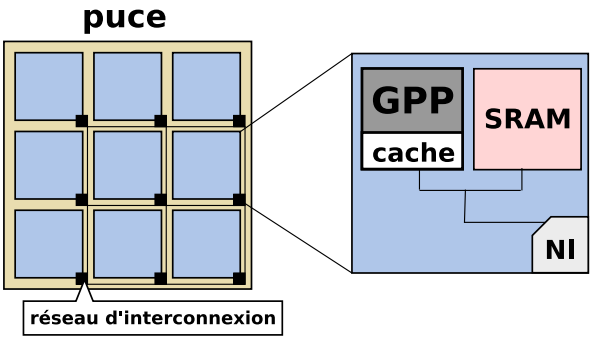


FIG. 2.3 – Architecture à processeurs homogènes embarquant de la mémoire partagée.

2.5 Caches et cohérence : de nouvelles contraintes

Dans le contexte de notre étude, nous utilisons de la mémoire cache. Nous présentons ci-après les différents problèmes liés aux nouvelles contraintes des systèmes embarqués.

2.5.1 Caches : simplicité et efficacité

Bien que les caches occupent une partie substantielle de la surface disponible dans une puce, leurs performances combinées à leur simplicité d'utilisation en font une des solutions privilégiées pour avoir des accès transparents et efficaces à la mémoire. L'utilisation des mémoires cache remonte à 1962 [BCP62], ils sont largement utilisées dans les architectures actuelles. L'usage des caches dans un système multiprocesseur où la mémoire est partagée implique la présence potentielle de copies des données. La cohérence des données est un problème majeur des systèmes multiprocesseurs.

2.5.2 Cohérence des données

La cohérence des données peut être gérée de façon logicielle à la charge du programmeur qui, quand c'est nécessaire, vide les mémoires caches afin de garantir une mise à jour cohérente des données. Néanmoins, il est possible de garantir la cohérence des données avec des mécanismes matériels. Ainsi, le programmeur n'a plus à se soucier de la gestion des caches.

Le coût en surface du maintient de la cohérence par la matériel est entièrement justifié par son efficacité et par sa simplicité d'usage vis-à-vis de la programmation logicielle.

La plupart des systèmes multiprocesseurs non embarqués intègrent un protocole de cohérence gérée par le matériel et implanté au niveau des contrôleurs de caches et mémoires.

Comme nous le verrons dans le chapitre suivant, il existe une multitude de protocoles de cohérence. Néanmoins, l'accès aux données est un point critique du système et l'évolution des technologies remet sans cesse en cause les choix et les solutions des concepteurs. Les protocoles de cohérence implémentés en matériel n'échappent pas à cette évolution.

2.5.3 Systèmes embarqués et nouvelles contraintes

Les technologies actuelles et à venir, permettent d'intégrer un grand nombre de processeurs dans une même puce. Pour les systèmes enfouis ou embarqués, les contraintes sur la surface et la consommation requièrent l'intégration de la plupart des composants et des périphériques au sein de la puce. Nous y retrouvons ainsi des modules mémoire, périphériques d'entrée-sortie, contrôleurs DMA, composants dédiés et plusieurs processeurs.

L'usage d'un NoC est une des solutions privilégiées pour assurer l'interconnexion de tous ces composants. L'avantage de ce type d'interconnexion est qu'il offre une bande passante cumulée presque illimitée. Puisque les mémoires sont intégrées à la puce, les transferts de données entre les caches et la mémoire bénéficient pleinement de la bande passante offerte par le NoC. De fait, celle-ci n'est plus un élément décisif lors de la conception d'un contrôleur de cache, d'autres éléments comme l'efficacité, la simplicité ou la consommation d'énergie deviennent prépondérants.

Au vu de ces nouvelles contraintes, ou plutôt, de la disparition d'une partie d'entre-elles, il convient de ré-évaluer certaines solutions au maintien de la cohérence mémoire.

Plus précisément, les protocoles de mise à jour mémoire de type *write-through* ont été très vite écartés à cause du trafic qu'ils généraient sur les liens d'interconnexion partagés (bus) ou sortant de la puce (multi-carte jusqu'alors).

Le problème auquel nous tenterons d'apporter une réponse est : est-il possible d'envisager une solution simple mais consommatrice de bande passante à la cohérence des caches ?

2.6 Évaluation et comparaison des protocoles de cohérence mémoire

Nous avons dans la section précédente, soulevé un problème vis-à-vis de la meilleure solution à implémenter dans notre contexte d'étude. Il se pose alors un deuxième problème : comment comparer deux solutions entre elles ?

Il n'est pas envisageable d'implémenter en matériel tous les protocoles de cohérence concurrents. Cela aurait un coût prohibitif en temps et en argent. Par conséquent, les architectes ont recours à une évaluation théorique des protocoles ou à la simulation de modèles.

2.6.1 Modèles de simulation et niveaux d'abstraction

Grâce à la puissance de calcul des systèmes actuels, la simulation des protocoles de cohérence est la solution privilégiée de nos jours.

Néanmoins, le modèle de simulation pourra avoir une précision très variable. Il pourra être décrit très précisément au niveau du *bit* et du cycle d'horloge mais sa simulation sera lente, ou bien être abstrait permettant ainsi d'atteindre des vitesses de simulation élevées.

Il est très difficile d'abstraire un protocole de cohérence mémoire et d'obtenir des résultats précis qui prennent en compte toutes les implications des détails de l'architecture. En effet, les interactions entre le protocole de cohérence et l'accès aux données a des effets de bord non négligeables et difficiles à modéliser.

De plus, lorsque l'on compare des implantations de protocoles il est très difficile de savoir si la différence de performance observée est due aux caractéristiques intrinsèques du protocole, ou bien à la qualité de l'implantation qui en a été faite.

Utiliser des modèles de simulation permet de répondre en partie à la question posée. Néanmoins, il reste le problème suivant : comment comparer précisément des protocoles de cohérence, par nature très dynamiques, de façon à abstraire les détails de l'implantation mais tout en continuant de modéliser les effets de bord et les interactions ?

2.7 Migration des données dans la puce

L'usage de la mémoire cache ne répond pas à tous les problèmes liés aux accès mémoire et en pose de nouveaux.

2.7.1 La pénalité d'échec

Dans un système comprenant des dizaines de nœuds de calcul et bancs mémoire, la distance entre un nœud et la donnée sur laquelle il travaille peut être très grande.

L'usage des caches ne permet pas de diminuer la pénalité d'échec, mais seulement d'en réduire le nombre. Les échecs en cache ont une pénalité d'accès (latence) élevée. Tenter de réduire indéfiniment le taux d'échec est une mauvaise solution car elle implique une croissance de la taille des caches dont le coût en surface et en temps d'accès deviendrait prohibitif.

Au lieu de réduire le taux d'échec, il serait intéressant de réduire la pénalité d'échec. Une telle solution pourra être complémentaire à l'usage des caches.

2.7.2 La congestion sur un banc mémoire

Les caches permettent de réduire le nombre d'accès réalisés aux bancs mémoire. Néanmoins, un nombre élevé de processeurs accédant à des données se trouvant physiquement dans le même banc mémoire peut créer un point de congestion.

Un point de congestion aura des conséquences dramatiques sur l'ensemble du système. A l'instar de la congestion d'accès sur un bus, la congestion d'accès sur un banc mémoire augmente de façon dramatique la latence d'accès aux données.

Ce problème est d'autant plus pertinent que notre contexte d'étude se focalise sur les architectures multiprocesseurs à mémoire partagée. Il convient donc de trouver une solution à ce problème qui n'est résolu qu'en partie par l'usage de la mémoire cache.

2.7.3 Surface utile

L'usage des caches pose un nouveau problème : l'occupation de surface utile. En effet, l'utilisation massive des caches réduit la surface utile de la puce à cause de la duplication massive des données. La surface occupée par les caches pourrait être mise à profit pour stocker des données à exemplaire unique, avoir des processeurs supplémentaires ou encore réduire la taille de la puce. L'idéal serait un système offrant les même performances d'accès que les caches tout en maximisant la surface utile de la puce.

2.7.4 La solution aux problèmes ou les problèmes de la solution ?

Une solution qui permet de répondre à ces trois problèmes à la fois est de déplacer les données entre les différents bancs mémoire afin d'optimiser leur placement. Dans un système idéal, les données manipulées par une tâche sont placées dans le banc mémoire qui se trouve à proximité. Dans notre contexte d'étude, nous ne faisons aucune hypothèse sur le logiciel. Il est fort probable que les tâches en cours d'exécution ne soient pas confinées à un seul processeur, et qu'elles migrent en fonction de la charge des nœuds et des décisions du système d'exploitation.

Quelle que soit la solution envisagée, il faudra répondre à trois questions fondamentales : quand ? ou ? et comment ?

Quand : le déplacement des données peut se faire *a priori* (anticipation) ou *a posteriori* (réaction). Cette dernière solution est très certainement la plus simple à réaliser mais en contre partie, le déplacement des données risque d'être toujours en retard. Au contraire, si l'on est en mesure d'anticiper de façon juste le comportement dynamique de l'application, il sera possible de réagir au bon moment pour déplacer les données utilisées. Toute la difficulté est là, comment anticiper le comportement d'une tâche lorsque l'on n'a aucune hypothèse sur son implantation ?

Où : si dans le système les tâches travaillent uniquement sur des données privées, il est aisé de répondre à la question : les données doivent être placées au plus proche de la tâche qui les utilise. Dans un système multi-tâche et multiprocesseur, les données peuvent être partagées par plusieurs tâches s'exécutant sur plusieurs processeurs simultanément. Où doit-on placer les données afin d'améliorer les performances de l'ensemble du système ? Si l'on place systématiquement les données au « centre » de la puce, ne risque-t-on pas de créer des points de congestion ?

Comment : même si l'on arrive à répondre convenablement aux questions précédentes, comment réaliser le transfert des données ? Dois-t-on utiliser une solution logicielle ou bien une solution gérée par le matériel ? Comment anticiper ou détecter le besoin de migrer des données ?

2.8 Conclusion

Nous avons soulevé ici un grand nombre de problèmes liés à notre contexte d'étude. Dans le cadre des systèmes multiprocesseurs à mémoire partagée et distribuée, intégrée dans la puce, comment fournir un accès transparent et efficace aux données ? L'efficacité touche trois aspects : la latence d'accès, la simplicité d'usage et finalement l'énergie consommée lors des accès mémoire.

Ce problème global va être abordé par trois aspects disjoints mais complémentaires : l'étude des protocoles de cohérence des caches, une méthode de comparaison et d'évaluation des protocoles de cohérence, et enfin le placement ou déplacement des données dans le système.

Nous pouvons ainsi énumérer les principaux problèmes qu'il conviendra de résoudre :

- L'évolution des contraintes dans les systèmes embarqués remet-elle en cause les choix réalisés quand aux protocoles de cohérence des données à utiliser ?
- L'utilisation d'un protocole de cohérence simple, *write-through invalidate* est-elle envisageable dans les systèmes multiprocesseurs intégrés ?
- Comment comparer précisément des protocoles de cohérence de façon à abstraire les détails de l'implémentation mais tout en continuant de modéliser les effets de bord et les interactions ?
- Comment placer ou déplacer les données sur une puce afin d'améliorer l'efficacité des accès ?
- Comment réduire la pénalité d'échec d'accès aux caches ?

Première partie

Chapitre 3

Caches et cohérence des données

Dans ce chapitre nous aborderons les notions de base concernant les caches. Nous passerons en revue les différentes solutions proposées pour le maintien de la cohérence des données dans un système. Finalement, nous nous positionnerons vis-à-vis de ces solutions et des multiples travaux de la communauté scientifique sur le sujet.

3.1 Cache : minimiser la latence et la bande passante

La mémoire cache a été inventée en 1962 [BCP62] comme un moyen de diminuer efficacement la latence d'accès aux données. Elle est également utilisée [Goo83] pour diminuer la bande passante utilisée sur les liens de communication.

Un cache est une mémoire d'accès rapide (S-RAM par exemple) située à proximité de l'unité de calcul d'un processeur. Le cache contient un sous ensemble (copie) des données présentes en mémoire.

3.1.1 Accès au cache

Lorsque le processeur a besoin d'accéder à une donnée (lecture ou écriture) il envoie la requête au cache ❶ (cf. figure 3.1). Le contrôleur vérifie la présence de la donnée en cache à l'aide d'une comparaison ❷ avec l'étiquette[1] ou *TAG* de la ligne sélectionnée par l'index[2]. Il vérifie également l'état et la validité de la ligne à l'aide des bits de contrôle ❸. Si les tests réussissent, l'accès est un succès (*hit*) sinon, c'est un échec (*miss*).

En général (cela dépend de la politique de mise à jour mémoire), en cas de *hit* la requête est traitée localement. Si c'est une lecture, la donnée est fournie au processeur au cycle d'exécution suivant (quelques cycles pour les caches de grande taille). Si c'est une écriture, la donnée est mise à jour ainsi que les bits de contrôle.

Lors d'un *miss*, la requête est envoyée ❹ à la mémoire. Pour les lectures, la donnée reçue sera fournie au processeur mais elle sera également placée dans le cache. Les communications entre le cache et la mémoire se font à la granularité de la ligne. Ainsi, pour placer la nouvelle ligne en cache il faudra potentiellement évincer une donnée ❺ et mettre à jour le *TAG* et les bits de contrôle.

[1] bits de poids fort de l'adresse
[2] portion de bits contiguë à l'étiquette

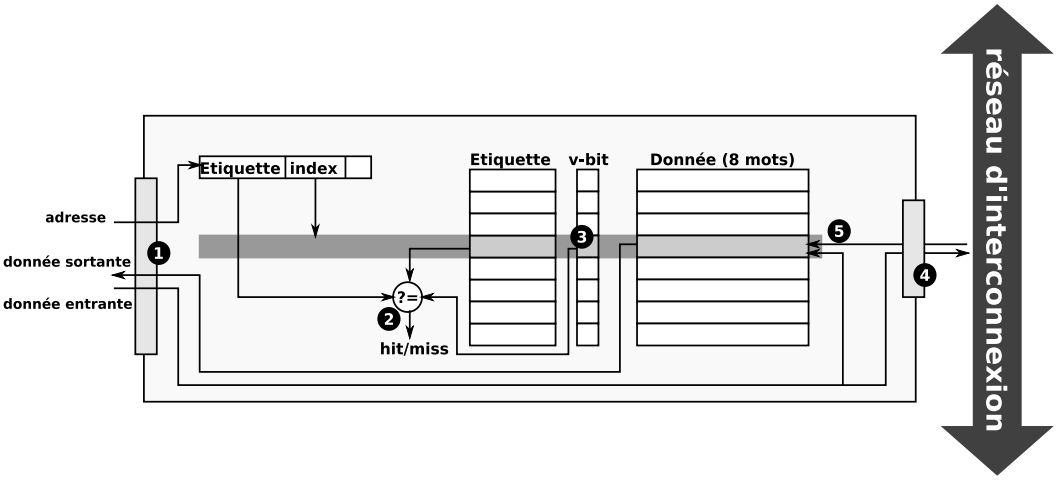


FIG. 3.1 – Différentes étapes de l'accès à un cache à correspondance directe.

3.1.2 Choix des données, associativité et ordre d'évincement

Lors d'un évincement, le choix de la ligne à remplacer dépend de l'algorithme choisi. Lorsqu'un cache est à correspondance directe (*direct mapped*) la question ne se pose pas et l'emplacement est choisi par son index (cf. figure 3.1).

Dans un cache associatif (partiellement ou totalement), le choix de la ligne peut être fait au hasard, dans un ordre FIFO[3] ou LRU[4]. Chacun de ces algorithmes a ses avantages et inconvénients, le plus performant mais également le plus coûteux à implanter est le LRU.

3.1.3 Le cache des instructions

Un processeur peut requérir deux types de données (au sens physique) : des instructions et des données (au sens programme). La plupart des architectures courantes sont de type Harvard où l'accès aux instructions et aux données se fait à des caches différents (figure 3.2) par des ports différents. Ceci permet à un processeur à plusieurs étages (*pipeline*) de ne pas avoir des conflits d'accès au cache par les étages responsables de la lecture d'une instruction et de l'accès aux données.

Dans ce type d'architecture, le cache des instructions et le cache des données sont séparés et fonctionnent de façon indépendante. Néanmoins, le contrôleur de cache peut n'avoir qu'une seule interface sur le réseau. En cas de conflit, les requêtes aux données sont en général prioritaires.

3.2 Politiques de mise à jour mémoire

En cas d'écriture, il existe deux politiques de mise à jour mémoire : *write-through* et *write-back*.

***Write-through* :** lors d'une écriture, la requête est toujours propagée à la mémoire. Ainsi, la mémoire possède à tout instant la dernière valeur écrite par le processeur. Si lors de l'écriture

[3] *first-in first-out* : premier arrivé, premier servi
[4] *Least Recently Used*, le moins récemment utilisé

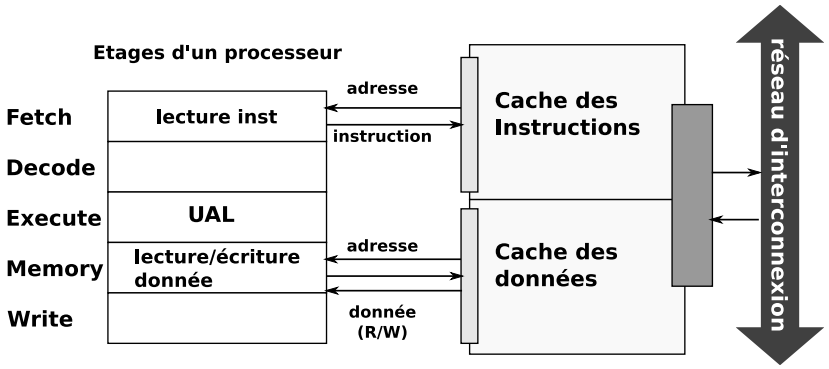


FIG. 3.2 – Architecture Harvard, les accès aux instructions et aux données se font indépendamment.

la donnée est en cache, celle-ci est également mise à jour.

Un avantage de cette politique c'est que lors d'un échec en lecture, la ligne de cache qui est évincée n'a pas besoin d'être recopiée en mémoire.

***Write-back* :** dans le cas d'une écriture, si la donnée est présente dans le cache alors la requête est traitée localement. Un bit de contrôle permet d'indiquer que cette ligne est modifiée (noté M ou D pour *Dirty*). Aucun accès à la mémoire n'est nécessaire, on économise ainsi de la bande passante sur le lien de communication.

Si la donnée n'est pas en cache, on applique en général une politique *write-allocate*. Dans ce cas, la donnée est préalablement lue en mémoire et mise en cache ; elle est ensuite modifiée localement.

Lors d'un échec en lecture (ou d'écriture avec allocation) il faut écrire la ligne évincée en mémoire si celle-ci est marquée comme étant modifiée. Cette écriture peut toutefois être réalisée après la lecture de la ligne afin de débloquer au plus tôt le processeur.

3.3 Cohérence des données

Comme nous l'avons vu dans le problématique (cf. section 2.5.1, page 10), un des problèmes majeur des systèmes multiprocesseurs est le maintien de la cohérence des données. Nous n'aborderons pas ici la cohérence des données faite de manière logicielle car celle-ci a été écartée de notre contexte d'étude. Afin de maintenir la cohérence des données dans un système, il y a deux façons de procéder :

Diffusion des écritures : lorsqu'un processeur émet une écriture, la valeur est diffusée dans le système et toutes les copies sont mises à jour. Cette technique est rarement utilisée car sa mise en œuvre est complexe à cause de certaines difficultés et limitations. En effet, la propagation des données à chaque écriture consomme de la bande passante. De plus, les conflits d'accès au cache par les requêtes provenant du processeur et celles reçues de l'extérieur peuvent nuire aux performances du système.

Invalidations lors des écritures : lorsqu'un processeur émet une écriture, toutes les copies du système sont invalidées. Nous sommes dans un schéma « plusieurs lecteurs, un seul écrivain ». Cette technique est de loin la plus utilisée de part sa simplicité de mise en œuvre et

d'un certain nombre d'avantages. Le premier avantage est une économie substantielle de la bande passante. En effet, seul un nombre restreint des données invalidées seront à nouveau accédées par les nœuds concernés (*miss* en cache). Deuxièmement, l'invalidation d'une ligne mémoire se fait très simplement par la mise à zéro des bits de contrôle du TAG. De plus, afin de minimiser l'interférence entre les accès du processeur local et les invalidations en provenance des autres caches, le contrôleur peut avoir un double du *TAG* pour traiter efficacement les invalidations (cf. figure 3.3).

Nous ne détaillerons pas davantage les tenants et aboutissants de ces deux façons de procéder. Dans nos travaux nous supposons l'usage d'un protocole à invalidations.

Nous présentons ici les différentes techniques gérées par le matériel pour maintenir la cohérence des données. Il existe deux grandes familles d'interconnexion, les bus et les autres (point-à-point). Pour chacune de ces familles il existe une technique appropriée.

3.3.1 Bus : espionnage

Les bus sont un moyen d'interconnexion simple et peu coûteux à mettre en œuvre. C'est un moyen de communication partagé par tout les intervenants, c'est-à-dire qu'il ne peut y avoir qu'un seul composant à la fois qui puisse émettre une requête.

Lorsqu'un composant émet une requête les autres peuvent néanmoins « l'observer » comme on peut le voir dans la figure 3.3. Cet espionnage (*snooping*) du bus est utilisé pour maintenir la cohérence des données. Ainsi, les caches vont continuellement espionner les requêtes qui transitent sur le bus. En cas d'une écriture ils peuvent décider d'invalider leur copie ou mettre à jour la valeur. En cas de lecture, ils peuvent changer l'état de partage de la copie ou encore émettre sur le bus la version modifiée qu'ils possèdent.

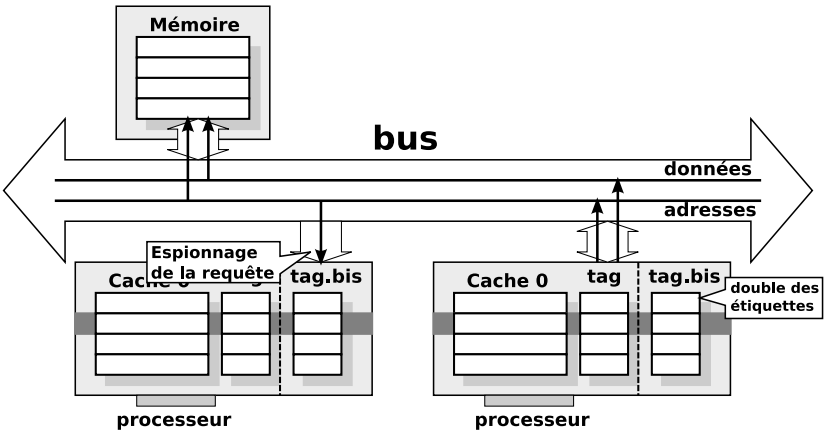


FIG. 3.3 – Espionnage des requêtes sur bus.

Le caractère « partagé » du bus a l'inconvénient qu'il limite énormément la bande passante disponible pour chacun des intervenants. Néanmoins il permet de propager très simplement les informations nécessaires à la cohérence des données à tous les intervenants.

Cette limitation en bande passante à poussé les concepteurs à privilégier les approches *write-back*. Les premiers protocoles implantés furent le `write-once` (1983) [Goo98], le protocole `Illinois` (1984) [PP84] et le protocole `Berkeley`(1985) [KEW+85]. Peu de temps après suivirent les protocoles `Dragon` (1987) [McC84], `Firefly` [TS87] et Synapse [Fra84].

Tout ces protocoles, basé sur l'espionnage d'un bus, utilisent (au moins partiellement) une politique de mise à jour mémoire de type *write-back*.

Un protocole de cohérence avec une politique *write-through* est le *write-through invalidate*. Ce protocole est tout simplement la combinaison d'une politique de mise à jour mémoire *write-through* avec un mécanisme d'invalidations lors des écritures.

Les protocoles qui utilisent une politique *write-through* sont pénalisés vis-à-vis des protocoles qui utilisent une politique de type *write-back*. Tout d'abord, ils sont très gourmands en bande passante alors que celle-ci est une ressource limitée sur un bus. De plus, dans un protocole de type write-back la lecture par un nœud d'une donnée modifiée dans un autre cache peut être réalisée à moindre coût. Dans ce cas, le nœud qui possède la donnée modifiée va, grâce à l'espionnage, pouvoir l'envoyer sur le bus afin de satisfaire la requête initiale. Au final, l'accès à la donnée aura eu une latence sensiblement égale à un accès direct à la mémoire.

En résumé, une politique de type *write-back* n'offre que des avantages sur un bus. Les diverses études comparatives sur le sujet [AB86, TM94, QY89] montrent effectivement que le protocole *write-through invalidate* offre de piètres performances dans une architecture à base de bus.

L.Benini a réalisé une étude comparative des différents protocoles de cohérence à base de bus pour les systèmes embarqués [LPB06]. Sa conclusion est la même, le protocole *write-trough invalidate* n'est pas compétitif sur un bus.

3.3.2 NoC : répertoires

Sur les réseaux d'interconnexion point-à-point comme les cross-bars, anneaux, mesh-2D, l'utilisation d'un répertoire pour le maintien de la cohérence est la meilleure solution.

Le répertoire est un ensemble de bits associé à un bloc qui indique quel cache possède une copie et dans quel état (cf. figure 3.4). Lorsqu'une requête atteint le banc mémoire ❶, le répertoire est interrogé pour connaître l'état de partage de cette ligne ❷. En fonction du protocole implanté des requêtes (invalidations) peuvent être envoyées aux copies du système ❸. La réception d'une invalidation oblige le contrôleur à modifier l'état de la ligne concernée et (en fonction du protocole) d'émettre une requête d'acquittement ou de mise à jour.

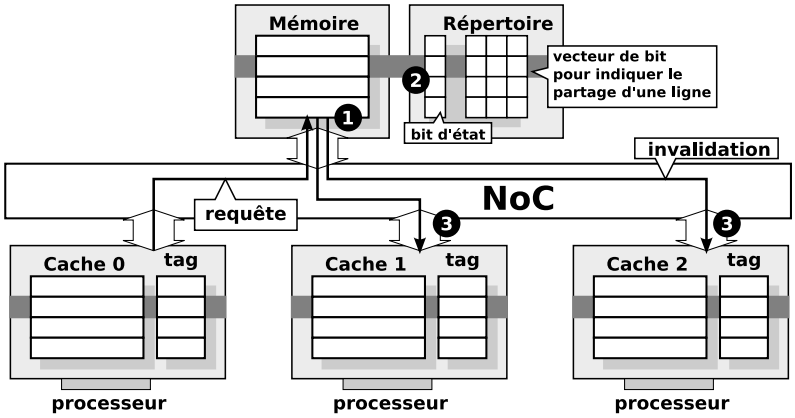


FIG. 3.4 – Principe des protocoles de cohérence faisant usage d'un répertoire.

L'arbitre du protocole n'est plus le bus qui sérialise les requêtes, mais le répertoire. Ici, une attention toute particulière doit être apportée à la gestion des acquittements et à la synchronisation des requêtes. Bien qu'en théorie l'implantation d'un protocole à base de répertoire puisse paraître aisée, dans la pratique elle est beaucoup plus complexe que celle d'un

protocole à espionnage de bus.

L'usage d'un répertoire est obligatoire dans la mesure où il n'est pas possible, à l'instar d'un bus, de propager toutes les requêtes à tous les intervenants du système.

La première réalisation a été faite par Tang en 1976 [Tan76]. Depuis, plusieurs solutions et variantes ont vu le jour [CF78, AB84]. Des études comparatives des différentes solutions ont été réalisées [TM94, ASHH88, Ste90].

Toutes les études et solutions proposées concernent des systèmes où la politique de mise à jour mémoire est *write-back*. Dans ces études, les architectures étudiées ne sont pas intégrées dans une puce, mais elles sont réparties sur un ensemble de puces, cartes ou machines.

3.3.3 Protocole write-through, positionnement

Dans notre étude, nous allons montrer la viabilité du protocole *write-through invalidate* sur un NoC. Une des hypothèses forte est que la mémoire partagée se trouve distribuée dans la puce, ainsi la bande passante disponible est très supérieure à celle que l'on trouve dans les solutions étudiées. Nous bénéficierons d'une grande simplicité d'implantation et d'une mise à jour constante des données en mémoire.

Chapitre 4

Protocoles de cohérence, mise à jour mémoire

L'intégration sur une même puce des processeurs et des mémoires, ainsi que l'utilisation d'un NoC au lieu d'un bus comme moyen de communication, redéfinissent les contraintes de la conception d'un système embarqué.

La bande passante offerte dans ce contexte n'est plus le point critique du système. Il convient donc de se concentrer sur les autres aspects de l'efficacité d'un protocole : la latence, la consommation et la complexité.

Nous présentons dans ce chapitre une étude comparative du protocole *write-through invalidate*, simple mais décrié par sa consommation en bande passante, il mérite d'être réévalué dans ce contexte précis.

4.1 Protocoles, caractéristiques et motivations

4.1.1 Le protocole *write-though invalidate* et ses avantages

Ce protocole est la combinaison d'une mise à jour mémoire *write-through* avec un protocole d'invalidations *write-invalidate*. Dans le contexte de nos travaux, le protocole d'invalidations est basé sur un répertoire (*directory based*). Nous en énumérons ci-dessous les différentes caractéristiques.

Toute écriture est propagée en mémoire. Cette caractéristique est la raison principale pour laquelle il n'est pas ou peu utilisé dans les systèmes multiprocesseurs. En effet, la plupart des systèmes multiprocesseurs utilisent un bus ou tout autre moyen d'interconnexion à bande passante limitée. Comme nous l'avons vu précédemment, l'usage de cette politique de mise à jour mémoire n'est pas envisageable sur un bus. Toutefois, nos travaux portent sur des plateformes multiprocesseurs sur puce faisant usage d'un NoC. Ce type d'interconnexion fournit une bande passante cumulée bien supérieure à celle d'un bus.

Dans un système avec des caches *write-through*, en moyenne une instruction sur 15 est une écriture et donc un accès à la mémoire (la plupart des lectures sont des succès en cache). Dans de telles conditions, il est difficile de saturer les liens de communication. Toutefois, la congestion peut apparaître sur le nœud mémoire s'il est accédé simultanément par un grand nombre de processeurs.

Enfin, afin d'amoindrir l'influence des écritures sur l'accessibilité des liens de communications, les caches posséderont un tampon d'écriture (*write-buffer*).

Une mémoire toujours cohérente. L'avantage inhérent des écritures propagées est le fait d'avoir des données toujours cohérentes en mémoire. De fait, la mémoire possède toujours la dernière copie d'une donnée ce qui simplifie le protocole de cohérence. Contrairement à un protocole de type *write-back*, il n'est pas nécessaire de récupérer dans un autre cache la dernière copie d'une donnée.

Dans une implantation autour d'un bus, ceci était une tâche aisée car l'espionnage des requêtes permettait au propriétaire de la donnée valide d'envoyer la donnée demandée. De plus, le temps de transfert d'un cache à un autre est sensiblement équivalent à celui de l'accès à une mémoire (intégrée à la puce).

L'utilisation d'un NoC rend impossible l'espionnage des requêtes, et donc un transfert de cache à cache d'une manière aussi simple. L'interrogation du répertoire afin de déterminer où se trouve la donnée valide est obligatoire. Par conséquent, récupérer la donnée depuis un autre cache de façon directe, ou bien à l'aide du contrôleur mémoire sera bien plus coûteux qu'une simple lecture en mémoire.

4.1.1.1 Simplicité d'implantation

Le protocole *write-through invalidate* est le plus simple des protocoles existant. Une complexité réduite se traduit par une économie de surface et une économie d'énergie au niveau de la gestion du protocole.

4.1.2 Protocoles comparés dans cette étude

Nous allons comparer ici deux protocoles. D'un côté le protocole *write-through invalidate* dont les principaux arguments ont été présentés ci-dessus, et de l'autre un protocole *write-back invalidate* dont nous détaillerons l'implantation ci-dessous.

4.1.2.1 Protocoles de haut niveau

***Write-trough invalidate (WTI)* :** la machine à états présentée dans la figure 4.1(a) montre les deux états possible d'une ligne de cache.

I : La ligne n'est pas en cache ou elle est marquée comme étant invalide.

V : La ligne est en cache. Les lectures sont traitées localement, les écritures sont propagées à la mémoire qui se chargera d'invalider les autres copies du système.

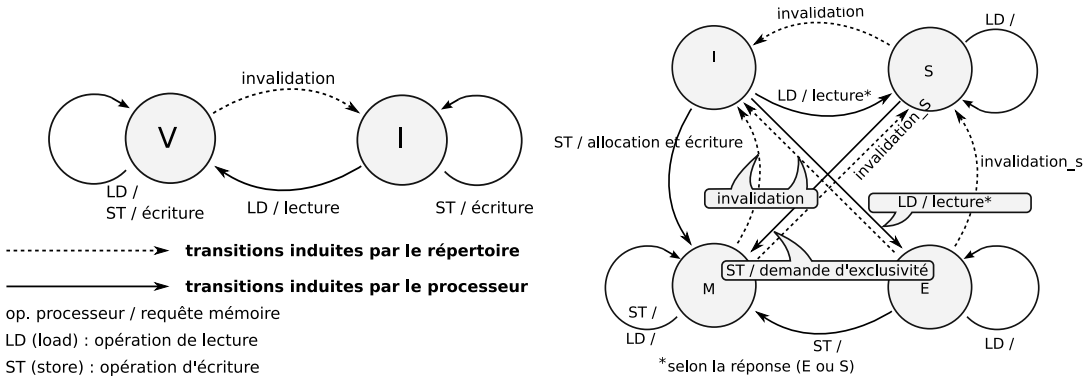
***Write-back invalidate (WB-MESI)* :** ce protocole est plus complexe, comme nous pouvons le voir dans la figure 4.1(b), une ligne de cache possède quatre états possibles :

I : La ligne n'est pas en cache ou elle est marquée comme étant invalide.

S : La ligne est en lecture seule, un ou plusieurs caches possèdent une même copie. Cet état est atteint lors d'un *miss* en lecture, ou d'une invalidation due à un *miss* en lecture d'un autre cache, obligeant la ligne à passer d'un état E ou M à S. Seul les lectures sont traitées localement, une écriture génère une demande d'exclusivité afin, d'une part invalider les autres copies du système et d'autre part, récupérer une copie à jour de la donnée si cela est nécessaire.

M : La ligne est modifiée en cache, toute lecture ou écriture est gérée localement.

E : Suite à un *miss* de lecture, si personne ne possède de copie de cette ligne, la réponse du contrôleur mémoire est marqué comme « exclusive » . Dans ce cas, cette copie est unique et toutes les requêtes du caches sont traitées localement (lecture, écriture).



(a) *Write-through invalidate : WTI* (b) *Write-back invalidate : WB-MESI*

FIG. 4.1 – Automates d'états des protocoles implantés

4.1.2.2 Implantation matérielle des protocoles

L'implantation d'un protocole quel qu'il soit laisse au concepteur une grande liberté quand aux choix à réaliser. Afin d'obtenir des résultats précis, à l'aide d'une plateforme de simulation CABA (cycle-accurate, bit-accurate), nous avons dû décrire et modéliser entièrement les composants mis en jeu (contrôleur de cache et contrôleur mémoire).

L'implantation réalisée influe sur les résultats obtenus, par conséquent nous allons détailler la façon dont sont gérées chacune des actions des protocoles. En fonction des implantations, un nombre différent d'actions et de messages peuvent circuler pour un même changement d'état dans le protocole de haut niveau.

La plus grande partie de la latence d'accès à la mémoire provient des parcours cache-mémoire des requêtes et réponses. Notons L_m la latence moyenne qui sépare deux nœuds du système.

Ci-dessous, nous allons détailler la manière dont les deux protocoles comparés vont implanter les différentes transitions d'états :

Requêtes de lecture : dans les deux implantations (WTI et WB-MESI), un échec en lecture a une latence d'au moins $2.L_m$, temps nécessaire à l'envoi d'une requête vers la mémoire et à la réception de la réponse.

Néanmoins, dans le protocole WB-MESI la ligne désirée peut être dans un état M dans un autre cache. En conséquence, il est nécessaire d'envoyer également une invalidation spécifique pour faire passer la ligne dans l'état S et récupérer par la même une copie à jour. Cette action prend $2.L_m$ cycles supplémentaires.

Requêtes d'écriture : dans le protocole WTI, les succès et les échecs d'écritures sont traités de la même façon : une mise à jour est envoyée à la mémoire.

Si après une interrogation du répertoire, il s'avère qu'il n'y pas d'autres copies dans le système, une réponse est immédiatement envoyée au contrôleur de cache. Dans ces conditions la requête a pris $2.L_m$ cycles d'horloge. Si une copie doit être invalidée, la durée de la requête est de $4.L_m$ cycles d'horloge. Néanmoins, l'usage d'un tampon d'écriture permet de ne pas bloquer le processeur tant qu'il ne subit pas un échec de lecture en cache avant que l'écriture soit achevée.

Dans le protocole WB-MESI, une écriture est un échec (ou interprétée comme tel) si la ligne est dans les états S ou I. Dans le premier cas, une copie à jour va être lue en mémoire ($2.L_m$) ou récupérée depuis un autre cache ($4.L_m$) par l'envoi d'invalidations.

Dans le deuxième cas, la requête peut prendre jusqu'à $6.L_m$ comme le montre la figure 4.2. En effet, pour l'allocation de la nouvelle ligne il peut être nécessaire de réaliser l'écriture de la ligne évincée (*write-back*). Malgré cela, le processeur n'est bloqué que pendant l'allocation de la ligne car l'écriture de la ligne évincée est effectuée après à l'aide d'un tampon d'écriture.

Finalement, si ligne est dans un état E ou M aucun envoi de requête n'est nécessaire, les écritures sont traitées localement en 1 cycle d'horloge.

Nous notons ici que dans la plupart des cas les écritures sont plus coûteuses en temps pour le protocole WB-MESI. Il est vrai que lorsque la ligne est dans les états M ou E l'accès en écriture est traité localement, mais par ailleurs, la présence d'un tampon d'écriture permet de masquer la latence aux processeurs dans un protocole WTI.

Le tableau 4.1 résume les caractéristiques des deux protocoles :

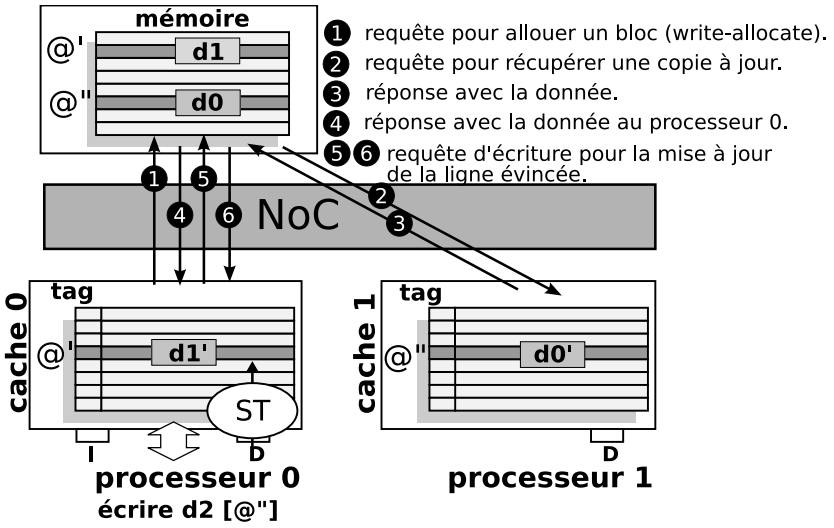


FIG. 4.2 – Exemple d'un accès qui coûte $6.L_m$ cycles d'horloges dans le protocole WB-MESI. Les adresses @′ et @" ciblent la même ligne dans le cache à correspondance directe.

Action processeur	WTI	WB-MESI
Lecture, *hit*	0	0
Lecture, *miss*	2 b.* *hops*	2 ou 4 *hops* b., (+2 n.b.**)
Écriture, *miss*	2,4 *hops* n.b.	2,4 b., (+2 n.b.)
Écriture, *hit*, état S	2,4 *hops* n.b.	2,4 *hops* b.
Écriture, *hit*, état E	-	0
Écriture, *hit*, état M	-	0

TAB. 4.1 – Coût en nœuds traversés dans chacun des protocoles

* accès non bloquant, mais peut pénaliser l'accès bloquant suivant pour garantir l'ordre des opérations

** accès bloquant, le processeur est en attente de la donnée

Les implantations qui ont été faites de nos protocoles ne sont probablement pas optimales. Nous aurions pu imaginer que l'acquittement des invalidations se fasse directement auprès des caches ayant effectué la requête, faisant chuter la latence d'accès de 4 à $3.L_m$.

Malgré cela, les principes suivis pour l'implantation ont été les mêmes, les protocoles restent donc comparables entre eux.

4.2 Expérimentations

Afin d'évaluer et comparer les protocoles WTI et WB-MESI nous avons réalisé leur implantation dans un environnement de simulation. Nous décrivons ci-dessous l'environnement de simulation utilisé, les architectures modélisées et la couche logicielle utilisée.

4.2.1 Environnement de simulation et architecture matérielle

Environnement de simulation

Nos plateformes de simulation ont été construites à l'aide de la bibliothèque de composants SoCLib [soc]. La plupart des composants sont décrits avec une précision au niveau du cycle d'horloge et du bit d'information CABA[1].

Certains composants sont décrits avec moins de précision. Ceci est notamment le cas du réseau d'interconnexion. Le composant utilisé « Generic Micro Network » (GMN) est décrit au niveau du cycle d'horloge, mais son implantation n'est pas réaliste. Il est modélisé comme un double crossbar (cf fig. 4.3) (pour les requêtes et les réponses) avec des files d'attentes. Ces dernières, de profondeur configurable, permettent de modéliser la latence minimum de traversée (file de délai) et la capacité du réseau (file de capacité).

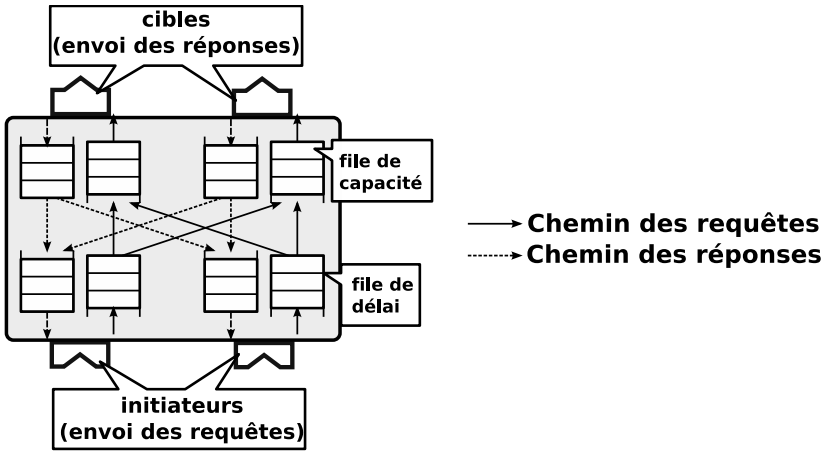


FIG. 4.3 – Architecture du GMN, la file de délai a un temps de traversée constant, la file de capacité peut être traversée en 1 cycle.

Ce composant a un comportement proche d'un NoC car il est peu sensible à la congestion et il est de type point-à-point. Son défaut majeur est que tous les composants sont virtuelle-

[1] Cycle-Accurate, Bit-Accurate

ment placés à la même distance les uns des autres, il ne permet donc pas de modéliser une topologie.

Malgré cela, le point important est que les deux protocoles implantés soient comparés sur la même base, l'erreur de précision obtenue sur le résultat absolu n'est pas gênante car nous nous concentrerons uniquement sur les comparaisons des résultats relatifs.

Cette bibliothèque nous permet de modéliser des plateformes contenant des dizaines de processeurs, mémoires et périphériques communiquant à l'aide du protocole VCI[2] [vci00].

Descriptif des architecture modélisées

Dans la figure 4.4 nous présentons les architectures modélisées. Nous avons utilisé des processeurs Sparc V8 d'architecture Harvard (cache instructions et données séparées). Notons que les deux caches partagent la même interface VCI sur le réseau afin de réduire sa surface.

Nous utilisons deux types d'architectures, l'une est très centralisée et l'autre au contraire très décentralisée. Les deux intègrent de 4 à 100 processeurs. L'architecture décentralisée possède un nombre variable de bancs mémoire.

Le tableau ci-dessous résume les caractéristiques des deux architectures.

Nombre de processeurs	$n = \{4, 16, 32, 64\}$
Nombre de bancs mémoire	$m = \{2, n + 3\}$
Processeur utilisé	SPARC-V8 et une FPU
Taille du cache des données	$4Kb$
Taille du cache des instructions	$4Kb$
Taille d'une ligne de cache	32 bytes
Politique d'évincement du cache	Direct-mapped
Taille du tampon d'écritures	8 words (32 bytes)
Configuration du GMN	Mesh - 2D
Latence du réseau	$3.\frac{2}{3}\sqrt{n+m+3}$

TAB. 4.2 – Caractéristiques des plateformes simulées

4.2.2 Applications

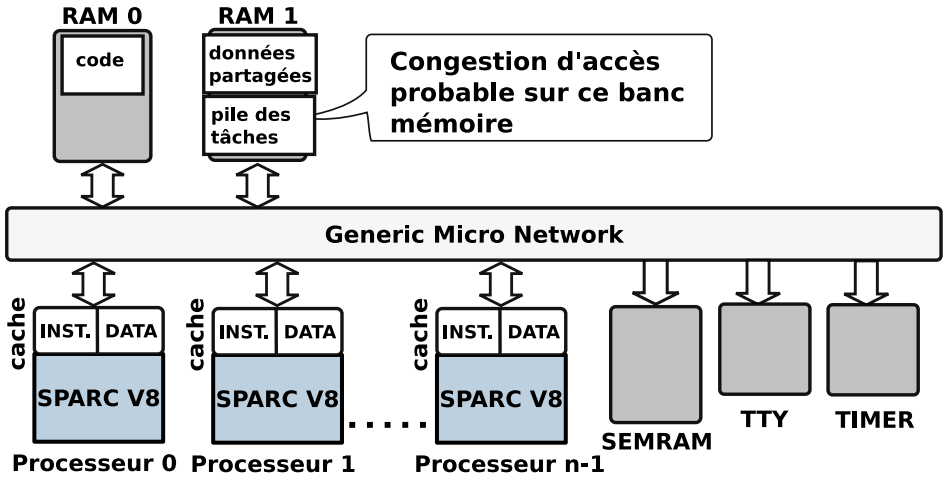
4.2.2.1 Application synthétique : simple_test

Afin de valider nos protocoles et d'observer certaines tendances, nous avons utilisé une application synthétique. Cette application crée un tableau de travail à deux dimensions. Cet ensemble est divisé de façon régulière entre les différents processeurs du système. Chaque processeur va réaliser un calcul simple sur le sous-ensemble qui lui est dédié et celui-ci, aura un taux de recouvrement avec les zones voisines paramétrable. Nous pouvons ainsi observer le comportement des différents protocoles en fonction du taux de partage des données.

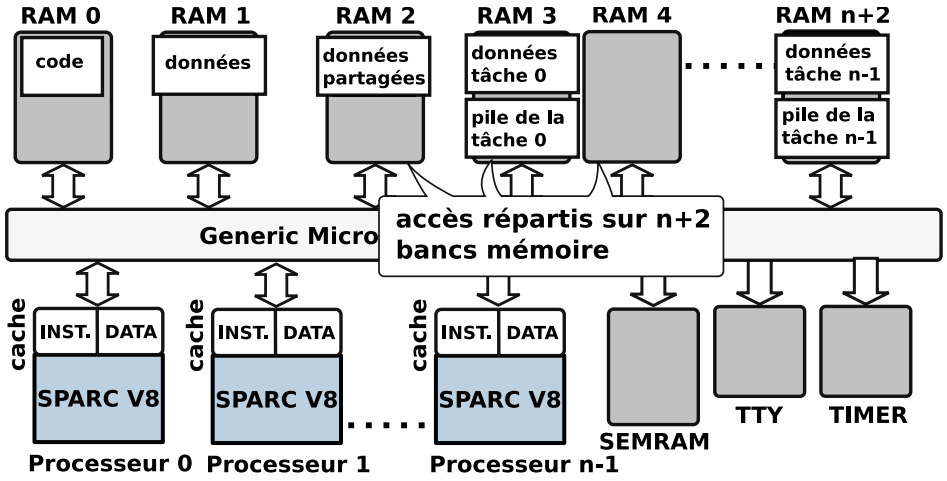
Dans la figure 4.5 nous présentons le placement des données de l'application en fonction du type de plateforme considérée. La distribution des données sur l'ensemble des bancs mémoire aura une influence déterminante sur le comportement des protocoles.

Cette application, très simple, ne requiert le support d'aucun système d'exploitation. Ainsi, l'ensemble des accès réalisés est entièrement déterminé par l'application.

[2] Virtual Component Interface



(a) Architecture centralisée



(b) Architecture décentralisée

FIG. 4.4 – Description des deux architectures utilisées

4.2.2.2 SPLASH-2

Nous avons utilisé 6 applications du jeu de test SPLASH-2. Dans [CME+95] nous pouvons trouver une description détaillée des applications SPLASH-2 utilisées (taux de partage des données, nombre de synchronisations, taille des problèmes, etc.). Ce jeu d'applications a été conçu pour évaluer les machines parallèles à mémoire partagée et distribuée. Bien que des jeux de test plus récents existent, ceux-ci sont toujours une référence et sont exécutables en un temps abordable sur nos plateformes de simulation.

Les applications fournies sont en général des applications parallèles de calcul intensif. Un ensemble de travail est divisé en n tâches réparties sur un ensemble de processeurs. Les tâches synchronisent régulièrement leurs résultats à l'aide de barrières de synchronisation.

Ces applications requièrent les services d'un système d'exploitation, de librairies standards (entrée/sortie essentiellement) et d'opérations de calcul à virgule flottante.

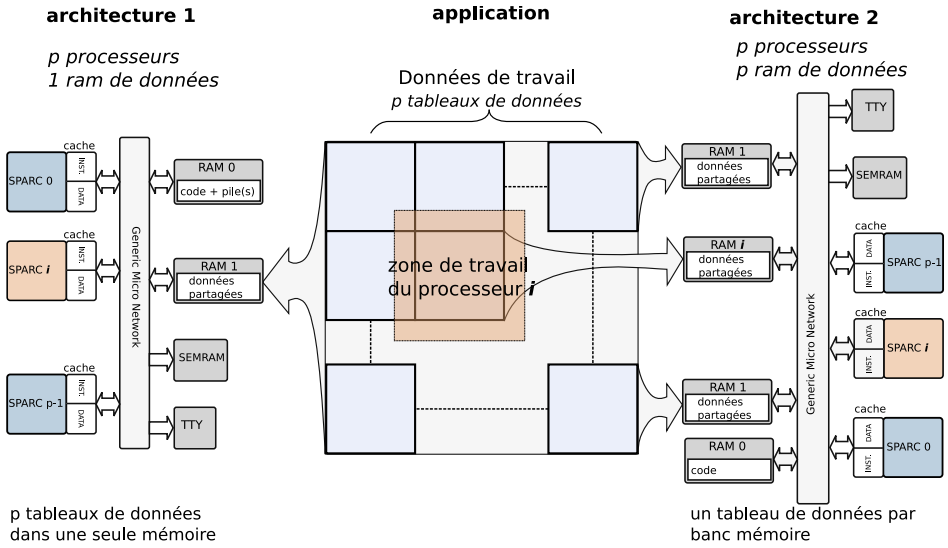


FIG. 4.5 – Application de test et validation `simple_test` : répartition des données en fonction de l'architecture considérée.

4.2.2.3 MJPEG

L'application de décodage vidéo Motion-JPEG est décrite par un ensemble de tâches communicantes à l'aide de files d'attentes (cf. figure 4.6).

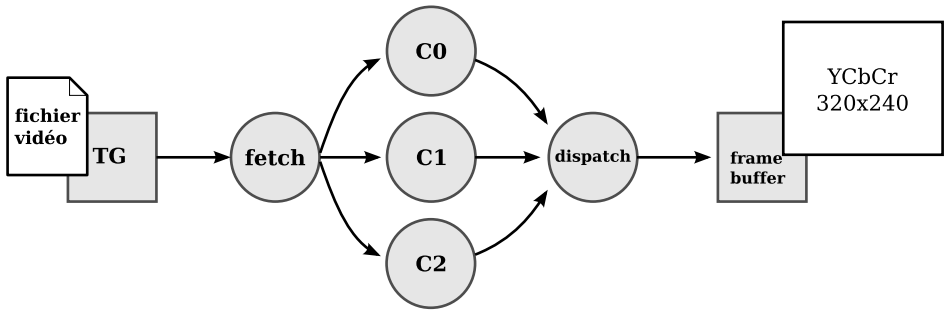


FIG. 4.6 – Application mjpeg décrite à l'aide de tâches communicantes

Contrairement aux applications précédentes, celle-ci traite un flux de données, elle est donc décrite en plusieurs étages (`fetch`, `compute`, `dispatch`). Le premier étage `fetch` lit le flux de données depuis un fichier que l'on accède au travers d'un module matériel TG. Il décompresse les données et les envoie à l'étage suivant.

L'étage `compute`, qui réalise le décodage des images, est parallélisable en plusieurs tâches. La configuration optimale dans notre contexte est de trois tâches (C0, C1 et C2). L'intérêt de cette application est que les tâches ont toutes une fonction spécifique et travaillent sur un flux de données. Le schéma des accès des processeurs à la mémoire varie donc en fonction de la tâche exécutée et au cours du temps.

Pour cette application nous mesurons le temps de décodage nécessaire à 4 images couleur d'une résolution de 320×240 pixels.

4.2.3 Système d'exploitation

Les applications (à l'exception de simple_test) requièrent des services qui sont fournis par un système d'exploitation. Nous utilisons ici `Mutek`[PG03] qui fourni un sous-ensemble de l'API POSIX et permet d'exécuter des tâches parallèles sur une plateformes multiprocesseurs. Ce système est configurable au niveau du placement des données et de la répartition des tâches :

Ordonnancement et données centralisées : dans cette configuration `Mutek` distribue l'exécution des tâches sur les processeurs disponibles avec une politique *premier arrivé, premier servi*.
Les tâches peuvent migrer entre les processeurs de façon aléatoire. Cette configuration n'est pas idéale pour les architectures de grande taille car l'accès à l'ordonnanceur centralisé devient un point de congestion.
De plus, les données, piles, tas et code sont centralisés sur deux bancs mémoire comme on peut le voir dans la figure (4.4).

Ordonnancement et données distribuées : dans cette configuration, les tâches peuvent être explicitement dédiées à un processeur. Ainsi, il est possible statiquement de distribuer les tâches sur les processeurs et d'en empêcher toute migration.
`Mutek` utilise un ordonnanceur par processeur afin d'éviter tout point de congestion. De plus, les piles d'exécution sont placées dans les mémoires locales des processeurs favorisant ainsi la distribution des accès à la mémoire.

Nous utilisons la configuration centralisée de `Mutek` sur l'architecture centralisée, et la configuration décentralisée sur l'architecture éponyme. Ainsi, nous avons deux plateformes de simulation.

4.3 Résultats et commentaires

Nous détaillons ici les résultats obtenus avec les différentes applications et les différentes plateformes. Les trois informations relevées sont le temps d'exécution, la quantité de trafic généré et la latence moyenne des accès.

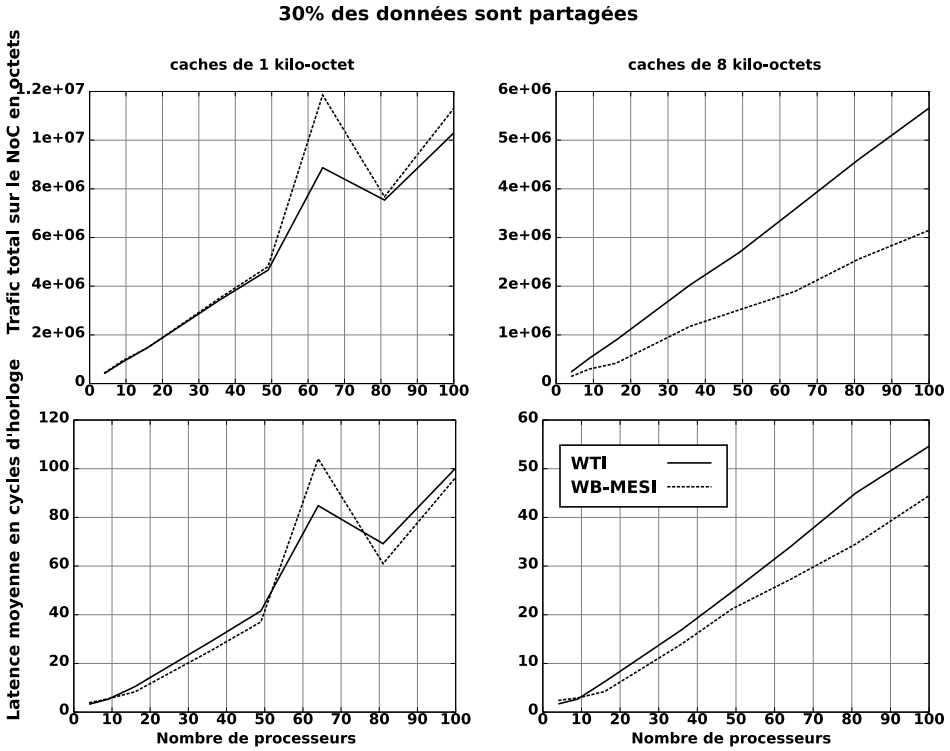
4.3.1 simple_test

Cette application a été exécutée sur des plateformes contenant de 4 à 100 processeurs. Les résultats sont présentés dans les figures 4.7 et 4.8.

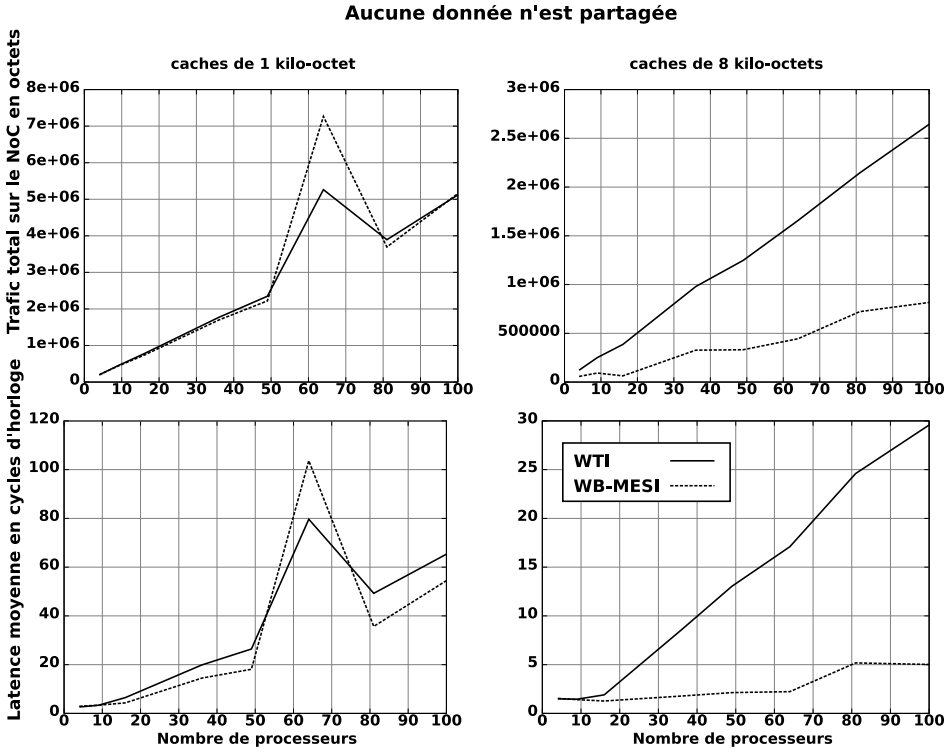
4.3.1.1 Résultats sur l'architecture centralisée

Latence d'accès : nous observons que le protocole WTI offre de moins bonnes latences d'accès aux données comparé au protocole WB-MESI. La diminution de la taille des caches ou l'augmentation du taux de partage des données réduit l'écart observé. Deux causes expliquent ce phénomène. D'une part l'accès à une donnée modifiée dans un autre cache est très coûteux en WB-MESI, d'autre part les échecs en écriture dans les caches induisent un coût en latence pour le protocole WB-MESI.

Trafic généré : le protocole WTI génère dans le pire cas à peu près 3,5 fois plus de trafic que le protocole WB-MESI. Le placement centralisé des données devient un point de congestion qui explique les latences élevées des accès aux données. Ce résultat est très dépendant de l'application utilisée et ne peut donc pas être généralisable. Néanmoins ce résultat est

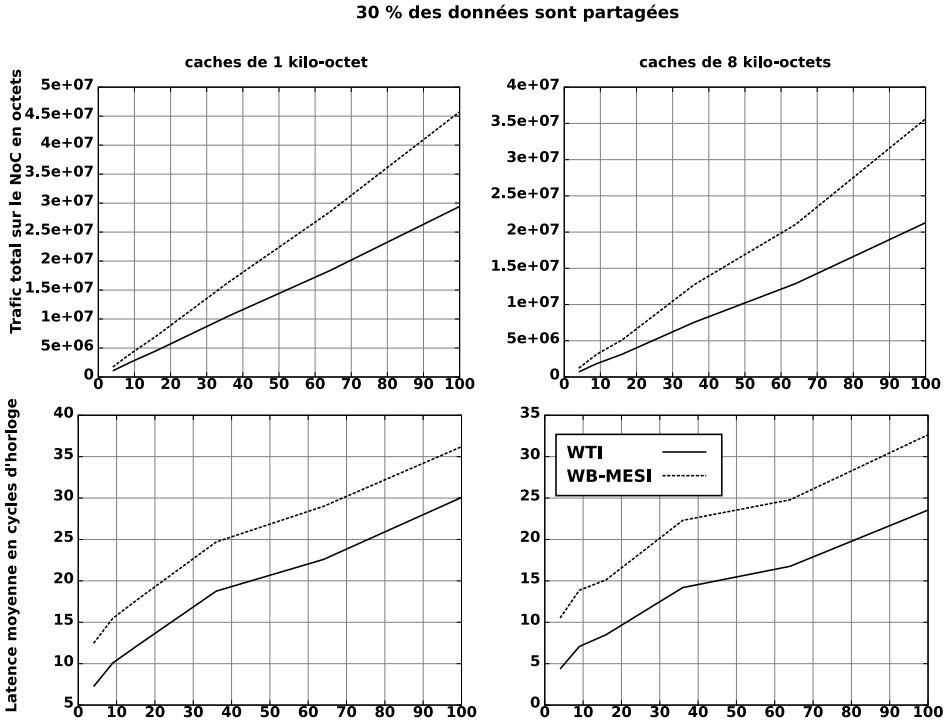


(a)

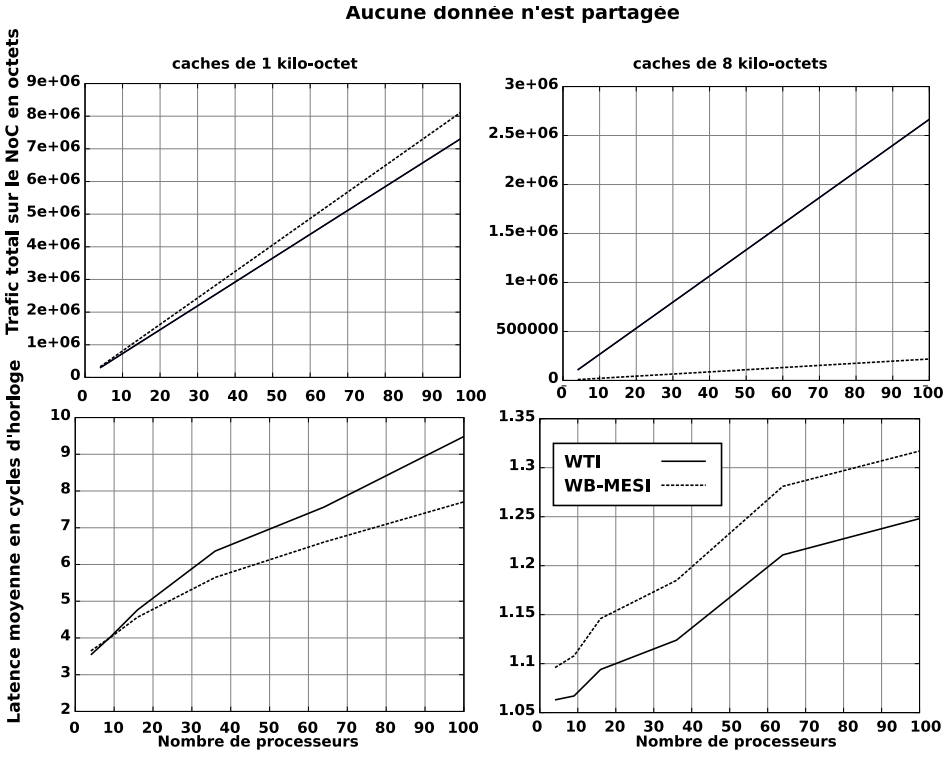


(b)

FIG. 4.7 – Trafic et latence moyenne pour l'application `simple_test` sur l'architecture centralisée.



(a)



(b)

FIG. 4.8 – Trafic et latence moyenne pour l'application `simple_test` sur l'architecture décentralisée.

conforme à nos attentes. L'évolution du trafic généré en fonction des paramètres (taille de cache, taux de partage des données) va dans le même sens que l'évolution de latence d'accès, l'écart entre WTI et WB-MESI diminue et devient marginal.

4.3.1.2 Résultats sur l'architecture décentralisée

Latence d'accès : d'une manière générale les résultats sont meilleurs sur l'architecture décentralisée. Ceci est dû en partie à la meilleure distribution des données qui contribue à diminuer la latence des accès de façon considérable. Ceci permet également de mettre en évidence l'existence d'une congestion d'accès dans l'architecture précédente.

L'usage de caches de 8 kilo-octets permet d'avoir un ensemble de travail entièrement contenu en cache. Nous observons donc ainsi l'influence du partage des données sur les latences d'accès.

Le partage des données donne un avantage certain au protocole WTI qui offre ainsi une plus faible latence d'accès. Lorsque les données ne sont pas partagées et que le cache est suffisamment grand, alors le protocole WB-MESI prend l'avantage. Puisqu'il n'y a pas de congestion d'accès, le problème est lié à l'efficacité de l'implantation du tampon d'écritures. Les lectures après écritures doivent attendre l'acquittement de la première afin de garantir la consistance des données. Le protocole WB-MESI a ici un avantage.

Le partage des données permet au protocole WTI de profiter des mises à jour continues de la mémoire ce qui réduit de moitié la pénalité des échecs en mémoire.

Trafic généré : le protocole WTI génère à peu près 5 fois plus de trafic que le protocole WB-MESI comme dans l'architecture précédente. En effet, la répartition des données a une légère influence sur le trafic généré. généré.

Les différences que l'on peut observer avec la première architecture sont dues à des conflits d'adresse dans le cache. Celui-ci est à correspondance directe et est donc sensible au placement des données. Ceci explique certainement le point discordant que l'on peut observer avec 64 processeurs (sur l'architecture centralisée).

Notons que la bande passante cumulée disponible sur le réseau croît avec le nombre de nœuds connectés.

4.3.1.3 Conclusions sur l'application `simple_test`

Cette application nous a permis de mettre en évidence certains avantages et inconvénients du protocole *write-through*. Comme nous pouvions l'imaginer, le trafic généré n'est pas un problème pour le réseau. Néanmoins, l'accès à un banc mémoire en particulier peut devenir un point de congestion. Les expérimentations avec l'architecture centralisée montrent bien ce phénomène : le protocole WTI offre de moins bonnes performances dans toutes les situations.

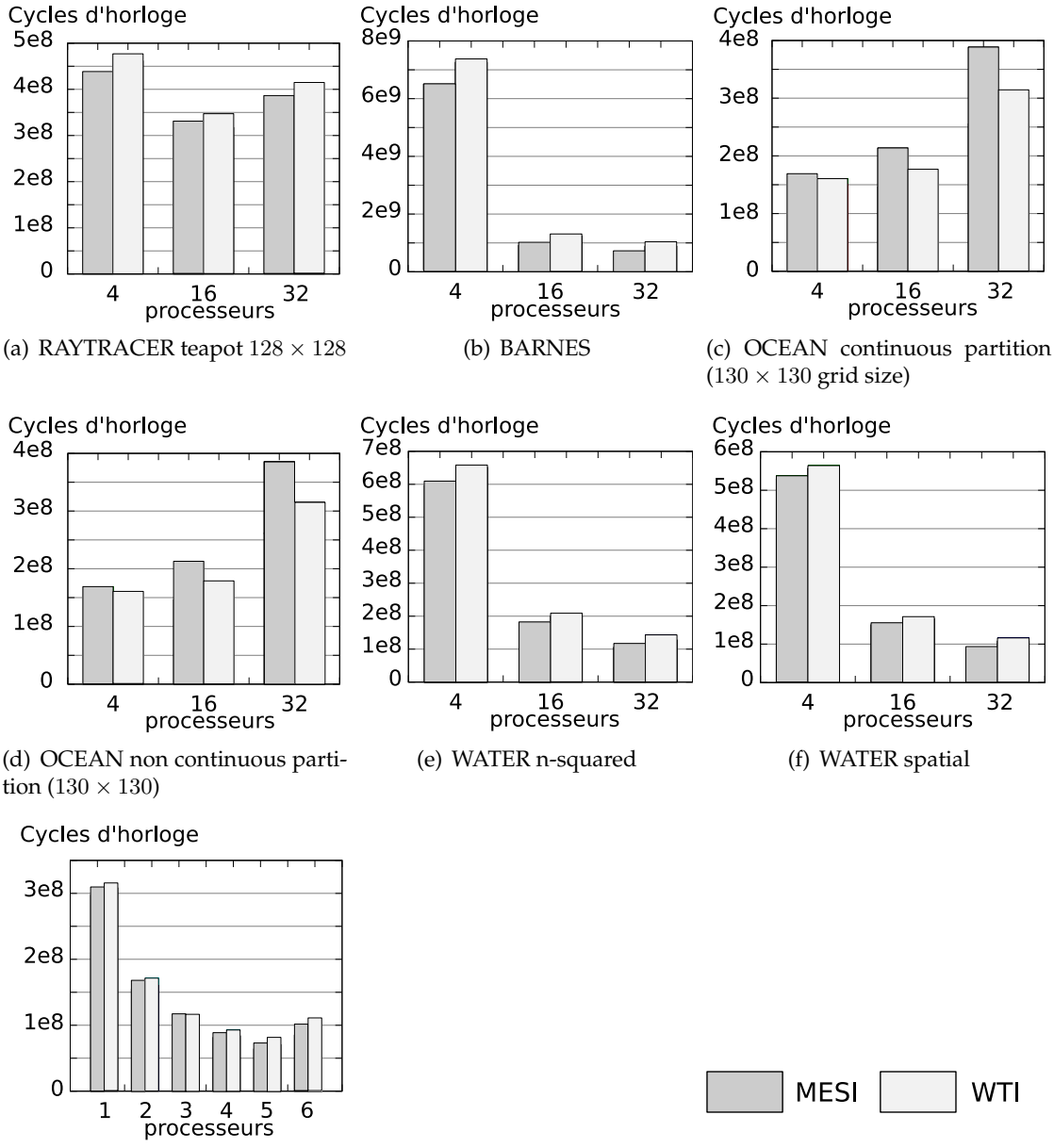
La distribution des données permet d'éviter les points de congestion et donne un avantage au protocole WTI. Celui-ci tire profit d'une mémoire toujours à jour et par conséquent, le maintien de la cohérence des données a une influence sur les performances moins prononcée que pour le WB-MESI.

Nous observons donc les tendances suivantes :

- Le partage des données favorise le protocole WTI, essentiellement grâce à une pénalité d'échec moins élevée en moyenne.
- Une bonne distribution des données est nécessaire pour éviter des points de congestion avec le protocole WTI.

– Une taille de cache importante favorise le protocole WB-MESI qui bénéficie d'écritures locale en un cycle après avoir alloué la donnée en cache.

4.3.2 Applications SPLASH-2 et MJPEG



(a) RAYTRACER teapot 128 × 128

(b) BARNES

(c) OCEAN continuous partition (130 × 130 grid size)

(d) OCEAN non continuous partition (130 × 130)

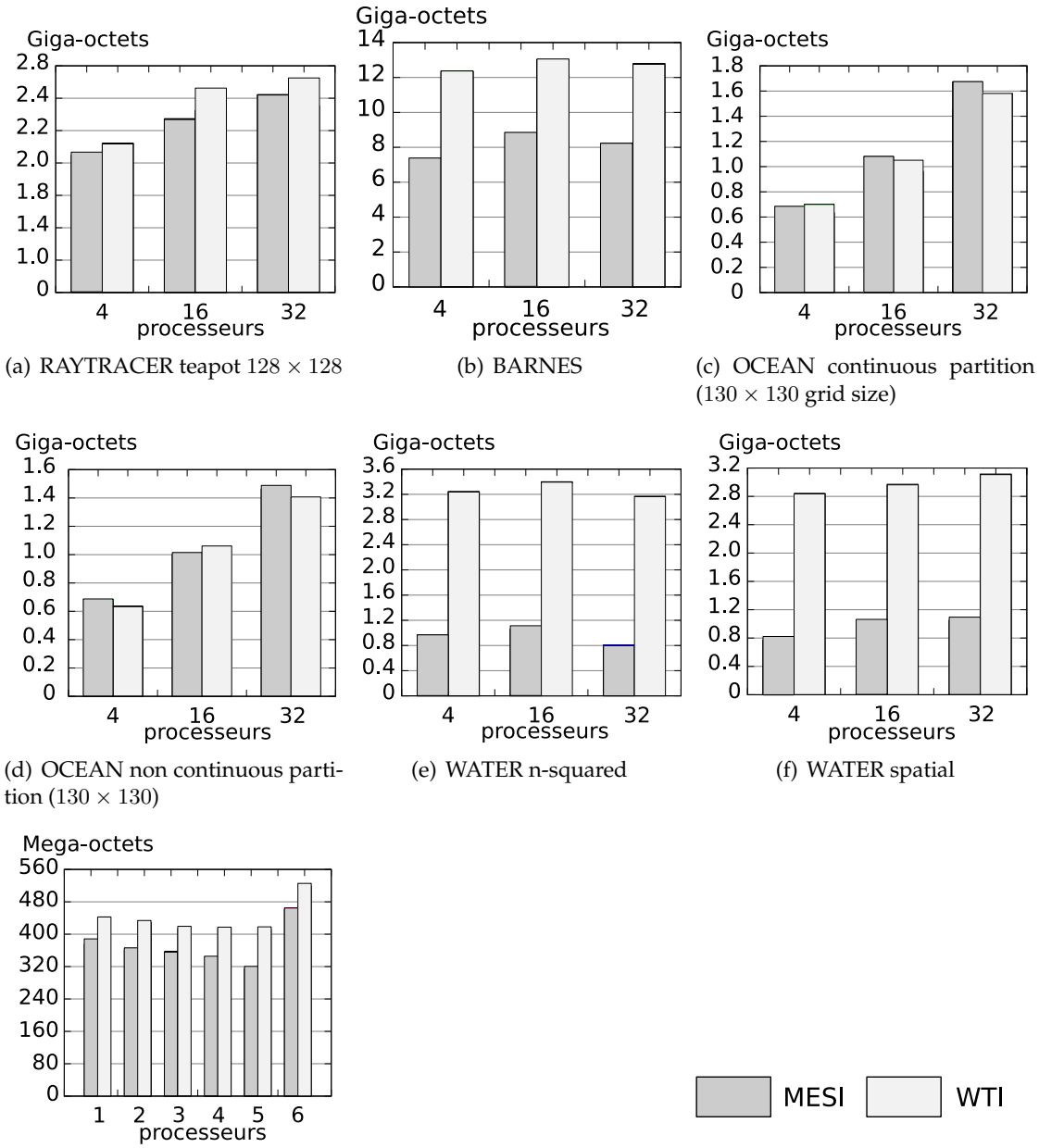
(e) WATER n-squared

(f) WATER spatial

(g) Motion-JPEG : 256x144 pixels, 4 images

FIG. 4.9 – Temps d'exécution des applications

Pour ces applications, afin d'éviter les problèmes de congestion que nous avons précédemment montré, nous n'avons réalisé des simulations que sur l'architecture décentralisée. Les résultats obtenus sont présentés dans les figures 4.9 et 4.10.



(a) RAYTRACER teapot 128 × 128

(b) BARNES

(c) OCEAN continuous partition (130 × 130 grid size)

(d) OCEAN non continuous partition (130 × 130)

(e) WATER n-squared

(f) WATER spatial

(g) Motion-JPEG : 256x144 pixels, 4 images

FIG. 4.10 – Cumul du trafic généré, il inclue toutes les requêtes et toutes les réponses émises sur le réseau d'interconnexion.

4.3.2.1 Analyse des résultats

Temps d'exécution : contrairement à nos attentes, le protocole WTI offre des moins bonnes performances que le protocole WB-MESI dans la plupart des applications testées. Néanmoins, la différence de performance ne dépasse pas les 20% et est en général inférieure à 10%.

D'une manière similaire, le protocole WTI offre un gain faible en temps d'exécution pour les applications `ocean-c` et `ocean-nc`.

Trafic généré : le trafic généré est, dans son ensemble, similaire pour les deux protocoles. D'après la littérature et les tests préliminaires avec l'application `simple_test`, nous devrions avoir une quantité de trafic générée bien supérieure pour le protocole WTI que pour le protocole WB-MESI. Néanmoins, nous observons que la différence de trafic est faible pour `raytracer`, `ocean-nc`, `ocean-c` et `mjpeg`. Cette différence ne dépasse pas les 20%.

La plus grosse différence observée en terme de trafic est pour l'application `water-ns` où le protocole WTI génère 4 fois plus de trafic que le protocole WB-MESI. Notons par ailleurs que dans certains cas le protocole WTI génère moins de trafic que le protocole WB-MESI, c'est le cas pour `ocean-c`, `ocean-nc` et `raytracer` (32 processeurs).

4.3.2.2 Conclusions sur les applications

Nous avons constaté que :
- Le protocole WTI offre des performances similaires au protocole WB-MESI bien qu'étant un peu en retrait.
- Le trafic généré par le protocole WTI est jusqu'à 4 fois supérieur à celui généré par le protocole WB-MESI

Il convient de relativiser ces résultats, qui au premier abord, ne semblent pas très bons. Rappelons tout d'abord qu'un des principaux avantages du protocole WTI est sa simplicité de mise en œuvre.

Deuxièmement, nous avons montré ici que ce protocole était tout à fait comparable à notre implantation WB-MESI en terme de performances contrairement à ce que présageaient les différentes études mentionnées.

4.4 Conclusion de l'étude

Nous avons réalisé tout au long de ce chapitre une étude comparative des protocoles *write-through invalidate (WTI)* et *write-back invalidate (WB-MESI)*. Nous en avons réalisé des implantation précises au niveau CABA, que nous avons comparé à l'aide de la simulation d'architectures multiprocesseurs à mémoire partagée et distribuée.

L'utilisation d'une application de test contrôlée et synthétique (sans système d'exploitation) nous a permis de montrer un gain en performances dans certaines situations. Dans les systèmes où les données sont suffisamment distribuées, le protocole *write-through invalidate* a l'avantage sur le protocole *write-back invalidate* car il permet de diminuer la latence moyenne des accès. Par ailleurs, le trafic généré est certes plus important, mais il n'est pas suffisant pour créer de la congestion sur un NoC.

L'exécution d'applications plus complexes et réalistes avec un système d'exploitation montre des résultats plus mitigés. Le protocole *write-through invalidate* s'avère légèrement moins performant que le protocole *write-back invalidate* avec une augmentation du temps d'exécution de l'ordre de 10%. En revanche, le trafic généré est moins important que prévu. Dans la plupart des applications il est du même ordre de grandeur que celui du protocole *write-back invalidate*.

Nous avons finalement montré qu'un protocole de cohérence mémoire très simple comme le *write-through invalidate* offre des performances très comparables à un protocole beaucoup plus complexe. De plus, dès lors que les accès sont distribués sur plusieurs bancs mémoires, le trafic généré ne pénalise pas les performances du système.

4.5 Limitations de l'étude

L'étude réalisé s'est basée sur l'utilisation d'un simulateur précis. Ceci permet d'évaluer le protocole de cohérence en prenant en compte le système dans son intégralité : schémas d'accès aux données, détails de la micro-architecture, partage des données et du code au niveau du système d'exploitation, des bibliothèques et de l'application.

En contre partie, nous avons évalué et comparé le protocole à travers son implantation. Il en découle une question fondamentale : qu'avons-nous évalué, le protocole ou bien son implantation ? Dans la suite de nos travaux, nous apporterons une réponse à cette question.

Chapitre 5

Méthodes d'évaluation et de comparaison des protocoles de cohérence

Nous présentons dans ce chapitre une description non exhaustive des différentes techniques d'évaluation des protocoles de cohérence mémoire.

5.1 Modèles théoriques

Jusqu'au début des années 90, la puissance de calcul nécessaire à la simulation d'architectures matérielles complètes (processeurs, mémoires, réseaux d'interconnexion, etc.) n'était pas disponible. C'est pour cela que les premières études [AAHV91, QY89, CF78, PP84] sur les protocoles de cohérence ont été réalisées à l'aide de méthodes analytiques et statistiques.

Afin de simplifier le problème, ces analyses reposent sur des hypothèses qui sont très contraignantes et peu réalistes dans la pratique. Les modèles utilisés pour représenter la distribution des accès à la mémoire sont bien souvent peu représentatifs du comportement d'un programme. De même, ces modèles négligent souvent l'accès aux instructions.

Ces méthodes ne permettent pas d'évaluer l'influence des détails de l'architecture et donnent une vue très abstraite d'un système. Néanmoins, la comparaison des protocoles se faisant avec les mêmes hypothèses et contraintes, elle permettait de dégager des tendances et des rapports de performance.

5.2 Simulations dirigées par les traces *(trace driven)*

Dès le début des années 90 on vit apparaître les premiers modèles de simulation. Ceux-ci n'exécutaient pas du code à la volée, mais rejouaient les traces d'exécutions obtenues sur une vraie machine. Ces simulateurs modélisent précisément la hiérarchie mémoire et permettent d'évaluer le coût d'exécution en cycles d'horloge de chaque instruction.

5.2.1 Architectures monoprocesseurs

Ces simulateurs ne feraient pas d'erreurs d'appréciation si le déroulement de l'exécution dépendait uniquement du programme exécuté. Néanmoins, certains évènements indépendants du programme, comme les interruptions, sont difficiles à modéliser et font perdre de

la précision au simulateur. Malgré cela, ces modèles de simulation sont très précis pour les architectures à un seul processeur.

Des études [YRC+00, GH93] montrent les limites de cette approche, en particulier pour les plateformes multiprocesseurs.

5.2.2 Architectures multiprocesseurs et « effet papillon »

Dans une application une partie du contrôle dépend des données. Ceci est un véritable problème pour l'évaluation des architectures multiprocesseurs.

En effet, si le chemin des instructions est défini par la valeur d'une donnée qui est modifiée par un autre processeur, alors la date de modification de la donnée aura une influence sur le chemin suivi.

Dans la figure 5.1 un branchement dépend de l'évaluation d'un test sur une variable. Le chemin b sera exécuté sur le processeur 1 seulement si la variable v est modifiée par le processeur numéro 2 avant que le test ne soit effectué.

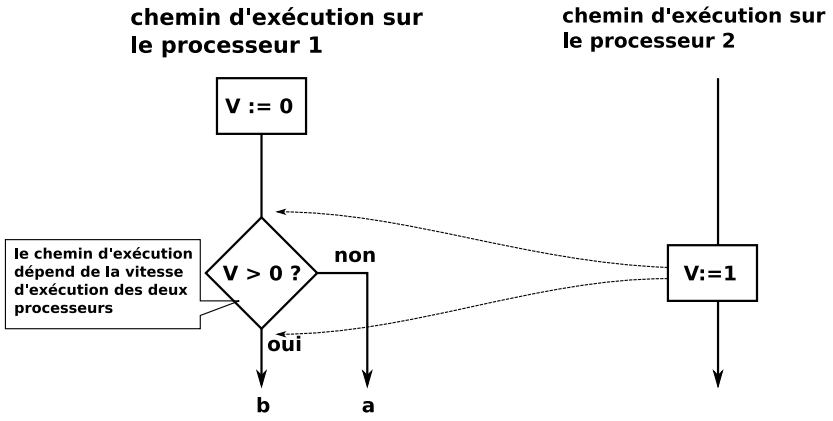


FIG. 5.1 – Exemple où le contrôle du chemin d'exécution dépend des données.

L'évaluation d'une trace d'exécution correspond à un unique chemin quelles que soient les vitesse d'exécution de chacun des chemins exécutés en parallèle sur différents processeurs.

5.3 Simulations dirigées par les évènements *(event driven)*

Afin d'obtenir une précision acceptable lors de la simulation des architectures multiprocesseurs, les modèles de simulation sont devenus dynamiques. Les différents processeurs sont simulés en parallèle et les chemins d'exécution sont évalués dynamiquement en fonction de l'arrivée des évènements.

Deux simulations où les paramètres de la plateforme ont été modifiés peuvent produire des résultats différents. La précision des résultats obtenus dépend directement de la précision avec laquelle les différents composants font évoluer leur état dans le temps. Cette précision peut être au niveau du cycle d'horloge, de l'instruction processeur, de la transaction mémoire, etc...

Les travaux récents et actuels d'évaluation des protocoles de cohérence ou d'architectures mémoire font tous (ou presque) usage de tels simulateurs.

5.3.1 Précision et niveau d'abstraction

La précision est fortement corrélée au niveau d'abstraction du simulateur (CABA, TLM, PV) clairement décrits dans [JBP06]. Si le niveau d'abstraction de la modélisation est haut, les vitesses de simulation seront très élevées mais les résultats obtenus auront une mauvaise précision.

Inversement, un niveau d'abstraction faible permet de modéliser finement tous les détails de l'architecture et de la micro-architecture. Par conséquent, les vitesses de simulations seront très lentes mais la précision obtenue sera très bonne.

Néanmoins, comme il est expliqué dans [PMT04], il est très difficile de modéliser de façon fidèle un protocole de cohérence et les erreurs de modélisation peuvent fausser les résultats. Dans certains cas, l'erreur de précision lors de la comparaison de deux solutions est supérieure à leur différence de performance supposée [GKO+00].

Toutefois, quelle que soit la précision du modèle de simulation, les résultats sont exploitables tant que les performances relatives des solutions comparées sont réalistes.

5.4 Conception des systèmes et modélisation

Dans les étapes préliminaires de la conception, les architectes essaient d'évaluer et de valider le plus tôt possible leurs solutions.

Pour cela ils favorisent les modèles de simulation abstraits, en effet pour obtenir un modèle fonctionnel le temps de développement est court et aucun détail de l'implantation n'est modélisé.

Afin d'évaluer précisément certains aspect de leur architecture ils peuvent opter pour des modèles hybrides [BSS+06, OCC08] qui permettent de modéliser à différents niveaux d'abstraction les différents composants.

Cette évaluation se base sur des comparaisons avec d'autres solutions elles aussi modélisées, ou bien si la précision du modèle le permet sur les résultats intrinsèques obtenus (estimation de performance).

Aucune de ces méthodes (à notre connaissance) ne permet au concepteur d'évaluer la qualité de son implantation. En effet, quels que soient les résultats obtenus il est difficile de savoir s'ils sont dus aux caractéristiques intrinsèques de la solution ou à l'implantation qui en a été réalisée. Pour cela, dans le contexte de l'évaluation des protocoles de cohérence, il faut être en mesure d'obtenir une borne inférieure sur les latences d'accès et le trafic généré.

Dans [MSB+05] et [MHW02] les auteurs présentent un environnement de simulation où le modèle fonctionnel est découplé du modèle temporel. Le modèle temporel modélise très précisément le réseau d'interconnexion, la hiérarchie mémoire et le protocole de cohérence. L'utilisateur a la possibilité de spécifier les valeurs du modèle temporel du protocole de cohérence. Toutefois, l'objectif de cette solution n'est pas l'obtention d'une borne inférieure. De plus, le modèle temporel a une flexibilité limitée et certaines de ses caractéristiques ne sont pas modifiables.

Chapitre 6

Méthode d'évaluation des protocoles mémoire : un cache omniscient

Dans le chapitre 4 nous avons comparé deux protocoles de cohérence mémoire. Pour cela, nous avons entièrement modélisé chacun des protocoles et nous avons réalisé des simulations d'architectures multiprocesseurs. Les résultats obtenus soulèvent néanmoins quelques questions. Quelle est l'influence de la qualité d'implantation des protocoles sur les résultats obtenus ? S'il est possible d'optimiser l'un ou l'autre des protocoles, quelle amélioration des performances sommes-nous en mesure d'envisager ?

Dans ce chapitre nous aborderons le problème de l'évaluation et de la comparaison des protocoles. Pour cela, nous avons créé une famille de protocoles que nous appelons « omniscients » que nous détaillerons ici.

6.1 Protocole omniscient : obtenir un meilleur cas d'évaluation

La dénomination « cache omniscient » est assez obscure au premier abord. Nous allons ici exposer le principe général d'un tel cache avant d'en présenter tous les détails. L'idée première est liée à l'optimisation d'une hiérarchie mémoire : quel est le gain en performances qu'il est possible d'espérer d'un nouveau protocole de cohérence et de son implantation ?

L'omniscience est la capacité à tout observer en même temps, d'être partout à la fois et de pouvoir agir en temps nul. Notre but est d'implanter un protocole de cohérence possédant ces caractéristiques dans un modèle de simulation. Un tel modèle offrirait une grande précision dans la modélisation de toutes les actions du système sauf pour celles liées à la cohérence des données. Ces dernières seront réalisées en un temps nul, partout où cela est nécessaire dans le système.

L'intérêt majeur de notre méthode est de fournir un « meilleur cas » aux performances de l'implantation d'un protocole. Il permet ainsi de borner le gain en performances qu'il est possible d'obtenir en optimisant un protocole complexe et son implantation.

Pour cela, nous avons deux façons de procéder :

1. Modifier le modèle de simulation d'un cache existant pour en enlever toute la logique liée au protocole de cohérence puis, implanter la cohérence omnisciente.
2. Modifier le modèle de simulation d'un cache qui ne possède aucun mécanisme de cohérence, afin d'y ajouter la cohérence omnisciente.

Nous sommes convaincus que la première solution n'est pas viable : modifier un modèle de simulation précis et complexe est très difficile à cause des adhérences et de l'intégration profonde des mécanismes liés au protocole.

La deuxième solution est, comme nous le verrons plus tard (cf. figure 6.4.2), beaucoup plus accessible. En conséquence, on pourrait argumenter sur le fait que cette méthode implique d'être en possession d'un modèle de simulation n'offrant pas le maintien de la cohérence. Cette hypothèse est vraie, et nous présentons donc ci-après les situations concernées par notre méthode.

1. Il existe un modèle de simulation pour un système complet offrant de la cohérence mémoire et les concepteurs désirent comparer l'impact de la cohérence des données sur les performances.
2. Pareil que précédemment, mais les concepteurs désirent évaluer un autre protocole. Il est probable qu'ils possèdent déjà un modèle de simulation du système sans maintien de la cohérence.
3. Les concepteurs possèdent un modèle monoprocesseur, et ils désirent implanter à bas coût un mécanisme matériel du maintien de la cohérence pour faire des expériences multiprocesseurs.
4. Les concepteurs possèdent uniquement le modèle d'une plateforme multiprocesseur qui maintient la cohérence de façon logicielle. Ils désirent évaluer cette solution avec une plateforme ayant la cohérence gérée de façon matérielle.

Cette méthode permet d'obtenir des résultats dans le meilleur cas d'exécution. Toutefois, il ne garantit pas que cette limite soit atteignable, ni à combien il est possible de s'en approcher. Cette limitation nous empêche de comparer des protocoles omniscients entre eux car les résultats ne seront pas exploitables. Malgré cette limitation, nous présentons dans la figure 6.1 un ensemble de situations où cette méthode est très utile.

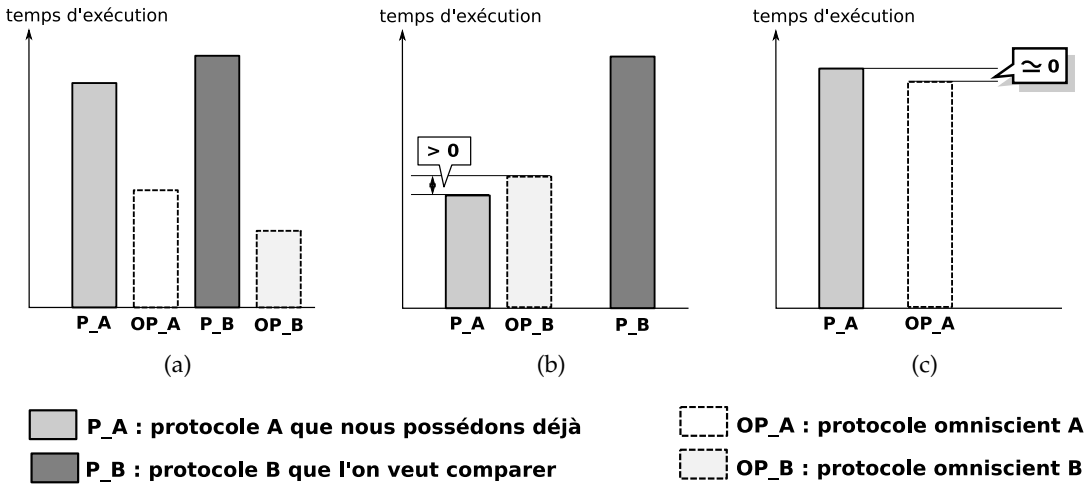


FIG. 6.1 – Trois *scenarii* de comparaison possibles

Dans la figure 6.1(a), nous présentons un premier scénario possible : les protocoles omniscients A et B présentent de meilleurs résultats que le protocole initial A. Ces résultats ne nous permettent pas de prédire la performance d'un protocole B, en effet comparer OP_A avec OP_B n'a pas de sens.

Dans la figure 6.1(b), nous comparons un protocole *omniscient* B à un protocole A. Ici, le protocole P_A a de meilleurs résultats que le protocole OP_B. Nous pouvons donc prédire que le protocole A offrira des meilleures performances que le protocole B et donc qu'il n'est pas nécessaire de dépenser une énergie considérable pour son implantation.

Dans la dernière figure (6.1(c)), un protocole A obtient presque les mêmes performances que son implantation omnisciente. Nous pouvons donc estimer qu'il est inutile d'essayer d'optimiser le protocole A car ses performances sont très proches de l'optimal.

6.2 Protocole omniscient, détails de la solution

6.2.1 Actions élémentaires d'un protocole

Un protocole de cohérence peut être décrit à un certain niveau d'abstraction en un ensemble d'actions et de transactions. Néanmoins, dans une implantation réelle, ces transactions ne peuvent pas être réalisées de manière atomique. De fait, elles sont décomposées en une succession d'étapes qui prennent plusieurs cycles d'horloge. Par exemple, envoyer une requête sur le réseau, interroger le répertoire ou encore mettre à jour l'état de la ligne de cache sont des exemples d'étapes élémentaires.

Chacune de ces étapes a une influence directe ou indirecte sur le comportement du système et de ses performances. Pour traiter une requête d'invalidation il faut de la logique supplémentaire. De plus, ces invalidations vont contribuer au congestionnement des liens de communication et donc avoir une influence sur la bande passante consommée et sur la latence des accès. De même, le contrôleur mémoire peut avoir à interroger un répertoire et donc avoir une pénalité d'accès accrue de quelques cycles d'horloge. Nous pouvons classer l'influence de ces actions en trois domaines :

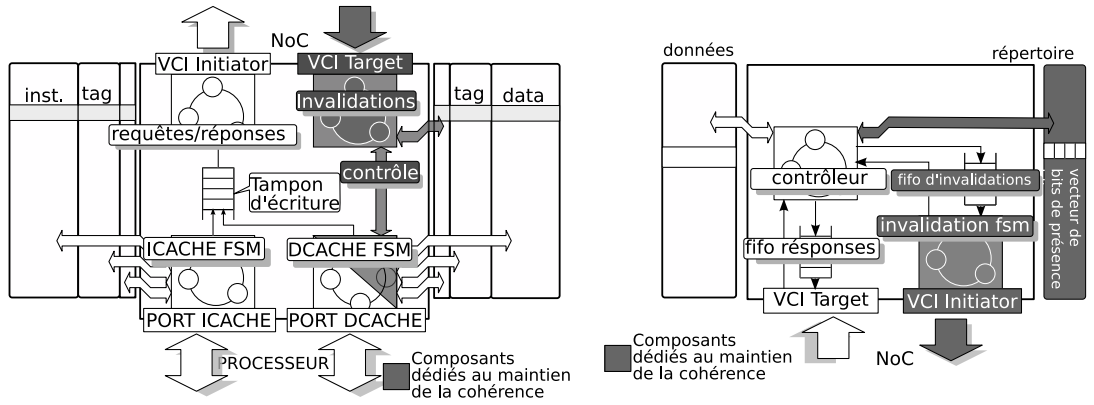
Nombre de transistors : chacune de ces actions requiert directement ou indirectement quelques portes logiques afin d'être traitée. En conséquence nous pouvons potentiellement observer un allongement du chemin critique, l'apparition de points de congestion, ou un surcoût en surface. Par exemple, sur un protocole de cohérence implanté autour d'un NoC, il peut être nécessaire d'avoir une interface supplémentaire (esclave) sur les contrôleurs de caches pour qu'ils puissent recevoir des invalidations. Ceci a une influence directe sur la taille du réseau et potentiellement sur les temps d'accès. Sur les figures 6.2(a) et 6.2(b) nous observons la logique de contrôle nécessaire au traitement des invalidations dans l'implantation d'un protocole de cohérence.

Latence : afin de maintenir la cohérence des données ; des requêtes supplémentaires (invalidations) devront être émises sur le réseau. Celles-ci peuvent être bloquantes pour le déroulement du protocole et donc avoir une influence directe sur les temps d'accès. Par exemple, comme nous l'avons vu (cf. 4.1.2.2 page 25) certaines requêtes peuvent avoir une latence plus longue à cause du maintien de la cohérence.

Bande passante : de la même manière, toutes les requêtes nécessaires au maintien de la cohérence consomment de la bande passante, dont une partie indirectement. En effet, l'invalidation des données crée potentiellement des échecs de cache qui vont engendrer de nouveaux accès à la mémoire.

Dans la figure 6.3(a) nous présentons une architecture contenant deux processeurs et un banc mémoire. La cohérence est maintenue par un protocole *write-through invalidate* à base de répertoire. Ci-après, nous illustrons une succession d'étapes générées par une instruction de rangement en mémoire (ST).

1. Le cache initiateur envoie une requête d'écriture vers le nœud mémoire : L_m cycles.
2. Le contrôleur mémoire :



(a) Architecture d'un contrôleur de cache. L'interface esclave permet de recevoir des invalidations

(b) Architecture d'un contrôleur mémoire. Le protocole de cohérence utilise un répertoire, l'interface maître permet d'envoyer des invalidations.

FIG. 6.2 – Architecture des contrôleurs de cache et mémoire. En noir, les infrastructures nécessaires au maintien de la cohérence.

   (a) met à jour la ligne : 2 cycles
   (b) interroge le répertoire : 1 cycle
   (c) génère une requête d'invalidation : 4 cycles
   (d) envoie la requête d'invalidation : L_m cycles
3. Le contrôleur de cache distant :
   (a) invalide sa copie locale : 4 cycles
   (b) envoie une réponse : L_m cycles
4. Le contrôleur mémoire :
   (a) génère un paquet réponse : 3 cycles
   (b) envoi le paquet : L_m cycles

Dans ce cas, effectuer la requête de rangement prend $2.L_m + 14$ cycles même si le processeur n'est pas bloqué (*stalled*). Si les requêtes et les réponses ont une taille d'un mot et les lignes mémoire de 8 mots, alors l'opération dans son ensemble génère 18 mots de trafic.

Pour un même protocole, les différentes implantations possibles auront un coût par étape différent, et un nombre d'étapes différent. Cette influence sur le coût peut être due à des caractéristiques intrinsèques du protocole comme le taux d'échec. Il peut être dû à des choix d'implantation à gros grains tels que le fait de permettre des transferts entre caches sans passer par le contrôleur mémoire[1]. Enfin, les choix d'implantation sur les détails de l'architecture ont également leur importance, la taille du tampon d'écriture, ou encore l'envoi asynchrone des invalidations ont tous une influence sur les performances du système.

D'autres optimisations sont également possibles, mais dans ce cas : comment prédire le gain maximum en bande passante consommée et en latence d'accès qu'il est possible d'obtenir sans concevoir une nouvelle version du protocole ?

Malheureusement, évaluer précisément l'influence directe ou indirecte des choix de conception lors de l'implantation d'un protocole est très difficile. Nous proposons donc une

[1]Transfert 3 étapes, une économie de L_m cycles dans notre cas

solution qui permet d'évaluer l'influence maximum de ces choix en réalisant certaines transactions d'une manière omnisciente. Nous apportons ainsi, des résultats dans le meilleur cas de la latence d'accès et de la bande passante consommée.

6.2.2 Transaction omnisciente d'un protocole

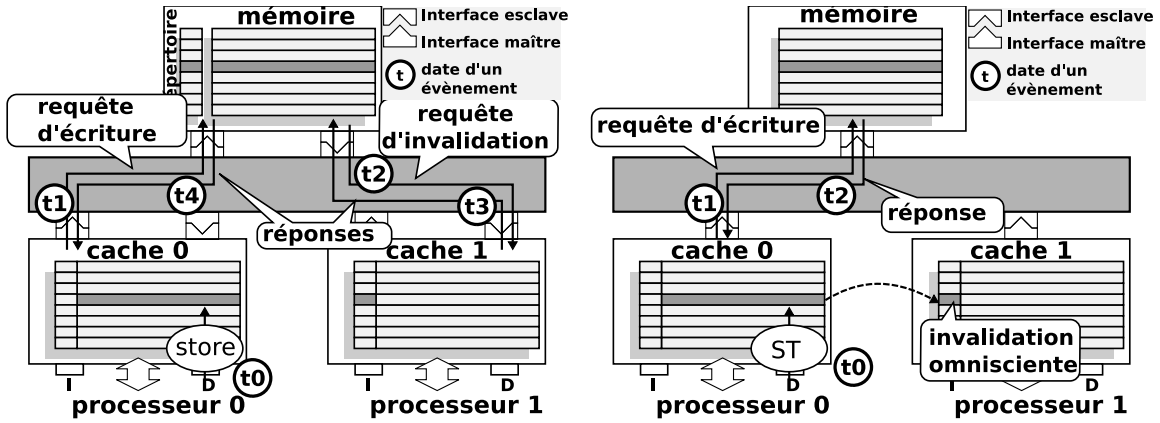
L'originalité de notre proposition est d'utiliser au sein d'un environnement de simulation précis, des actions instantanées et omniscientes. De telles actions modifient l'état d'un composant et par la même son comportement dynamique. Son implantation ne doit pas interférer avec l'architecture du composant.

Afin de pouvoir réaliser de telles actions, nous devons être capables d'interférer avec le modèle de simulation, d'accéder et modifier les états internes des composants simulés. Par exemple, dans un simulateur RTL, les invalidations vont être réalisées de manière omnisciente en modifiant le bit de validité des lignes de caches. Ceci peut être fait en appelant la fonction adéquate quelque part dans le modèle de simulation du cache ou de la mémoire.

Pour évaluer l'influence du protocole de cohérence, nous effectuons toutes les actions de cohérence de manière omnisciente. Nous appelons un tel protocole, un protocole omniscient. En conséquence, toutes les actions restantes, induites par une instruction de chargement/rangement du processeur, sont exécutées avec toute la précision du simulateur. Ainsi, la congestion du réseau, les latences d'accès et les synchronisations fines des évènements sont entièrement modélisées.

Chaque contrôleur agit comme s'il ne possédait aucun mécanisme de cohérence mais, l'architecture fournit des données cohérentes. Il en découle qu'un protocole de Cache à Cohérence Omnisciente (CCO) fournit la meilleure implantation possible d'un protocole de cohérence spécifique. Cette implantation est bien sûre non synthétisable, mais elle fournit un meilleur cas d'exécution pour toutes les implantations possibles d'un protocole donné.

Avec l'usage d'invalidations omniscientes ajoutées à un protocole de cohérence *write-through invalidate*, il est possible de réaliser une écriture et les invalidations nécessaires en $2.L'_m + 2$ cycles comme nous pouvons le voir dans la figure 6.3(b).



(a) Write-invalidate à base de répertoires : l'ensemble des étapes prend $4.L_m + 14$ cycles

(b) cohérence omnisciente : l'ensemble des étapes prend $2.L_m + 2$ cycles

FIG. 6.3 – Une instruction de rangement traitée par une architecture *write-through* avec deux protocoles de cohérence différents.

6.2.3 Implémentation et situations de compétition

Une solution naïve : une solution de mise en œuvre serait d'exécuter à l'aide de fonctions les étapes omniscientes d'une invalidation. Ces fonctions seraient appelées depuis des points pertinents du modèle d'un contrôleur de cache ou mémoire. Cette solution ne peut pas fonctionner correctement à cause des situations de compétition inhérentes à l'architecture modélisée.

Situations de compétition : Dans la figure 6.4 nous présentons une situation de compétition possible. Dans le schéma 6.4(a) au temps t_0 le processeur 0 émet une requête de lecture. Cette requête subit un échec de cache et elle est émise vers la mémoire à t_1. Le nœud mémoire envoie une réponse à t_2 qui va mettre un temps Δt avant d'arriver au cache.

Dans le schéma 6.4(b) à t_3 le processeur 1 émet une requête d'écriture à la même adresse, elle est émise vers le nœud mémoire pour mettre à jour la donnée (politique *write-through*). A cet instant, une action du protocole omniscient a lieu pour invalider les copies du système, et donc du processeur/cache 2. Puisque la réponse qu'attend le cache 0 ne lui est pas encore parvenue, l'invalidation ne sera pas prise en compte. Par conséquent, lorsque la réponse sera reçue par le cache 0, le contrôleur placera le bit de validité et mettra à jour la ligne de cache avec une donnée incohérente.

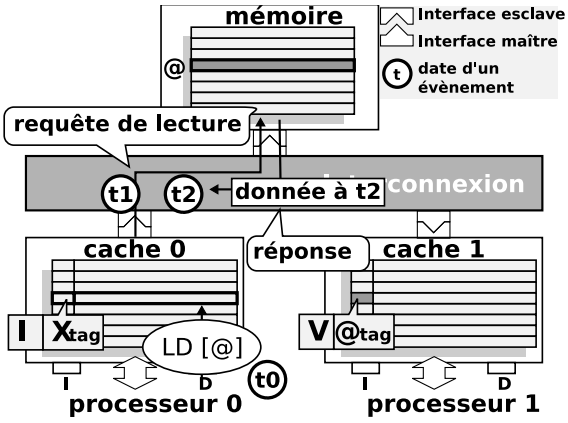
Dans la figure 6.5, nous présentons une compétition sur une requête d'écriture. Cette fois-ci, la donnée lue est incohérente car la requête de lecture a atteint la mémoire avant la requête d'écriture.

Dans un protocole de cohérence, les actions et les étapes sont synchronisées ou ordonnées afin d'éviter les problèmes de compétition d'accès et assurer la cohérence des données. Par exemple, dans le cas de la figure 6.4 le contrôleur de cache 0 aurait dû prendre en compte l'invalidation de la ligne et ne pas prendre en compte la réponse à la requête qu'il aurait détecté comme étant périmée. Comme notre approche omnisciente ne modifie pas le comportement de l'architecture pour synchroniser ou ordonner les requêtes, la solution est de rendre certaines actions atomiques. L'atomicité fournira l'ordonnancement et la synchronisation requise pour éviter les situations de compétition.

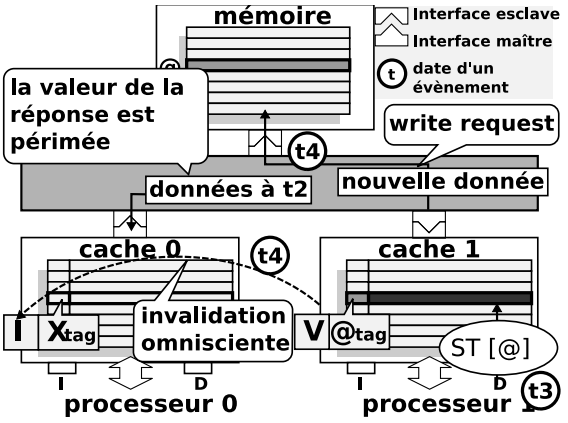
Solution : nous devons réaliser les lectures et les écritures comme des requêtes atomiques. Pour parvenir à cette fin sans modifier le comportement du composant, certaines actions vont être réalisées à la fois par le simulateur CABA et par des actions omniscientes. Ainsi, nous aurons d'un côté des temps réalistes puisque l'accès sera modélisé avec une précision CABA, mais également de la cohérence car la donnée effectivement lue ou écrite aura été traitée par une action atomique.

Deux fonctions sont nécessaires pour réaliser les actions omniscientes atomiques :

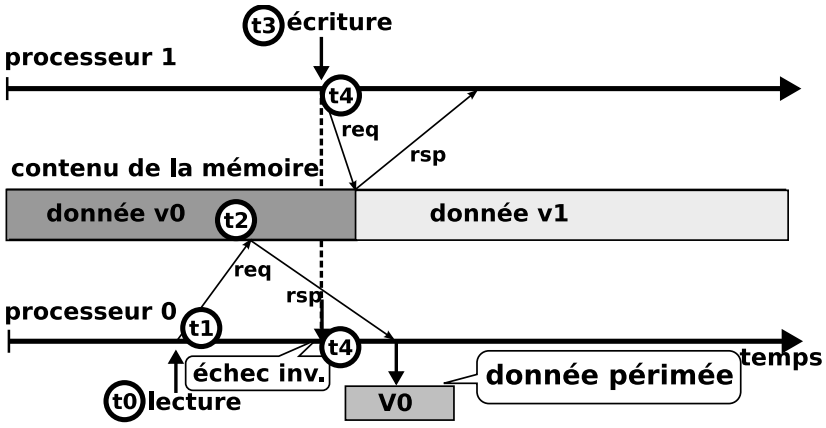
- **`ecriture_coherente_atm` :** Cette fonction réalise la mise à jour de la mémoire et les invalidations des autres copies de la donnée dans le système. De cette façon, la donnée est écrite en mémoire et invalidée dans le système de manière atomique (0 cycle). Par exemple, dans la compétition décrite précédemment (fig. 6.5), la copie en mémoire aurait été mise à jour à t_0 et par conséquent la donnée lue par le processeur 0 aurait été cohérente.
  Notons que lorsque l'écriture réalisé au niveau CABA atteint le nœud mémoire, la donnée est ignorée.
- **`lecture_coherente_atm` :** Cette fonction est appelée par le modèle du contrôleur de cache lorsqu'il reçoit la réponse à une requête de lecture. Cette fonction lit direc-



(a) Le processeur 0 réalise une lecture, la donnée est lue en mémoire.



(b) Le processeur 1 effectue une écriture, une invalidation omnisciente des copies est réalisée par le contrôleur mémoire. La réponse à la requête de lecture contient une donnée incohérente

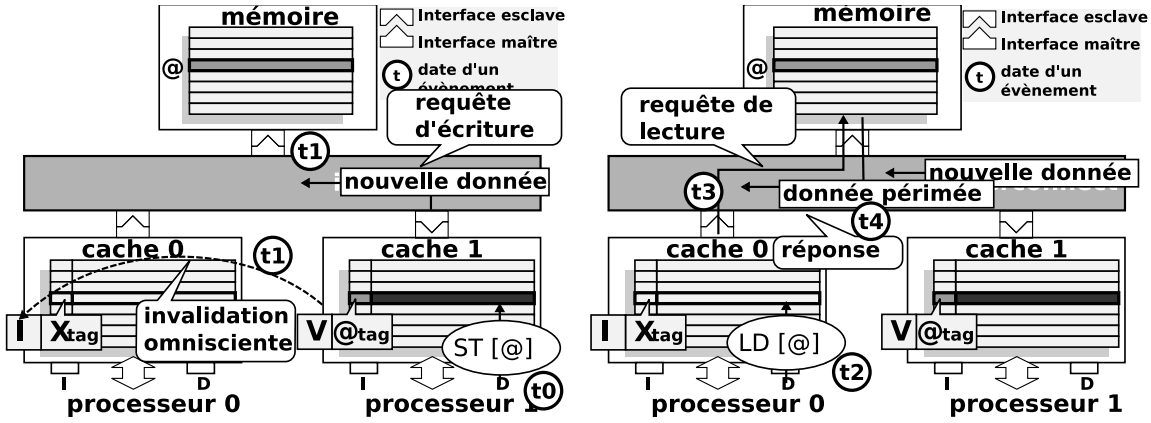


(c) Résumé des requêtes et des réponses

FIG. 6.4 – Exemple d'une condition de compétition sur une lecture, la donnée lue en mémoire est incohérente à t_4

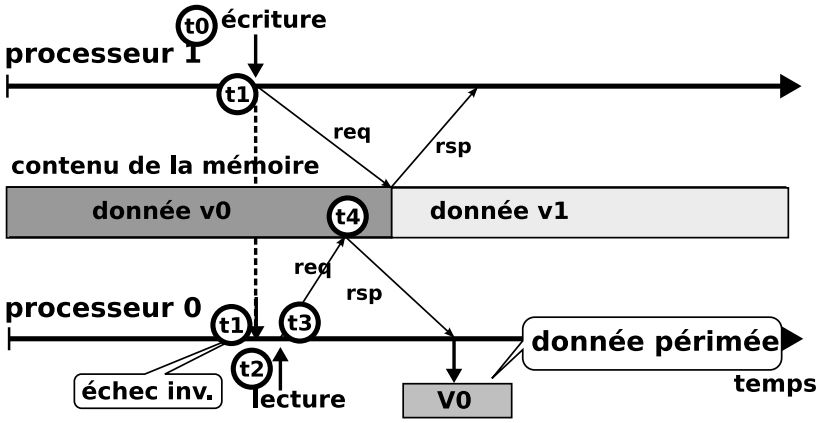
tement la valeur donnée présente en mémoire au cycle d'horloge courant. La réponse obtenue via la modélisation CABA n'est pas prise en compte.
Cette fonction permet d'assurer la cohérence de la donnée, une compétition comme dans le cas de la figure 6.4 n'est plus possible. Lorsque la donnée arrive en $t_2 + \Delta t$, la ligne de cache est mise à jour avec la donnée présente en mémoire à cet instant. Néanmoins, vis-à-vis du processeur, l'accès aura tout de même été effectué en $t_2 + \Delta t$ cycles.

Si, dans le même cycle d'horloge plusieurs composants font appel à l'une de ces fonctions, le résultat dépend de l'ordre d'évaluation des composants. Comme nous pouvons le voir dans la figure 6.6, les données sont lues et écrites de façon effective uniquement par ces fonctions. De fait, les lectures et écritures sont ordonnées et aucune compétition d'accès n'est possible.



(a) Le processeur 1 effectue une écriture, les copies vont être invalidées.

(b) Le processeur 0 réalise une lecture, la donnée obtenue est incohérente.



(c) Résumé des requêtes et des réponses

FIG. 6.5 – Exemple d'une compétition d'accès lors d'une écriture.

6.3 Concernant la consistance mémoire

Lorsque l'on conçoit une architecture mémoire multiprocesseur, une attention particulière est portée au modèle de consistance mémoire. En utilisant de la consistance séquentielle (SC) nous obtenons un modèle de programmation très simple. En utilisant un modèle de consistance plus relâché, nous obtiendrons de meilleures performances, mais des contraintes de programmation vont être reportées sur le programmeur. Par conséquent, notre approche *CCO* doit s'assurer de ne pas modifier les propriétés du modèle de consistance mis en œuvre dans l'architecture.

Nous allons montrer que l'utilisation (à bon escient) de cette solution ne remet pas en cause le modèle de consistance mémoire choisi. Pour cela, nous allons nous inspirer de la méthode présentée dans [CHPS99]. La méthode va être appliquée à notre implantation des protocoles *write-invalidate* (*write-back et write-through*).

Le modèle de consistance mémoire est défini par les contraintes imposées à l'ordre dans lequel les requêtes mémoires sont traitées vis-à-vis de leur ordre d'apparition dans le programme exécuté. Les opérations mémoire sont dans notre cas des lectures et des écritures qui appartiennent respectivement aux ensembles E_{LD} et E_{ST}. Ces opérations peuvent être

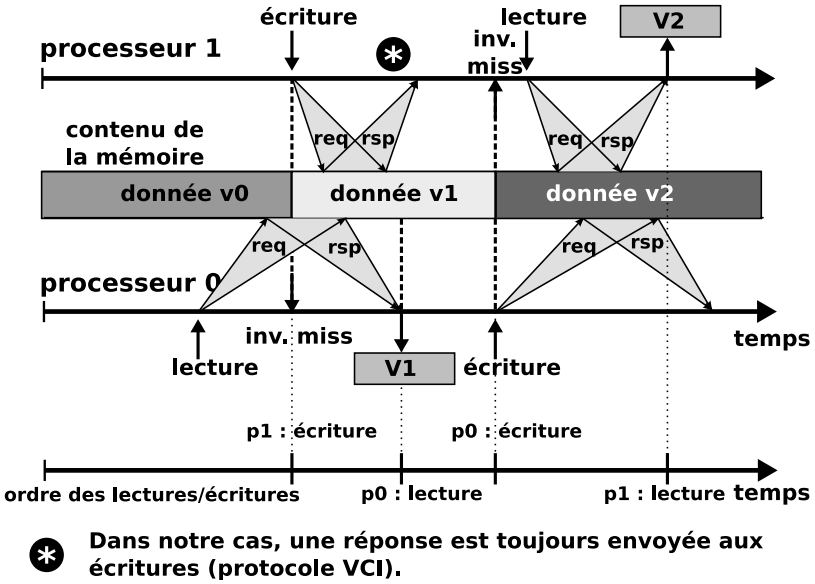


FIG. 6.6 – Application de notre solution pour les compétitions présentées dans les figures 6.4 et 6.5.

datées afin de définir un ordre total sur les opérations mémoire, à l'aide d'horloges de Lamport [Lam78]. Comme dans [CHPS99] nous notons $<_p$ l'ordre dans lequel les opérations apparaissent dans le programme. Par rapport à la mémoire cet ordre est total pour un système monoprocesseur, mais partiel pour un système multiprocesseur. Nous notons $<_m$ l'ordre total sur les opérations vues par la mémoire. C'est dans cet ordre que les écritures prennent effet. Cet ordre peut être différent de l'ordre du programme, ceci dépend du modèle de consistance mémoire présent dans l'architecture.

Dans la suite, ori et om font référence à toute relation concernant le protocole « originel » et sa version « omnisciente » respectivement. Dans notre modèle de consistance mémoire, tous les accès à la mémoire sont effectués dans l'ordre d'apparition du programme :

$$ori : \forall X, Y \in E_{LD} \bigcup E_{ST}, (X <_p Y \implies X <_m Y),$$

Pour qu'un protocole omniscient soit comparable à son homologue réaliste, il doit garantir que les conditions et les contraintes sur l'ordre des accès mémoire soient toujours valables. Le fait de réaliser certaines actions en un temps nul peut ajouter de nouvelles propriétés au modèle de consistance.

Soient X, Y deux opérations mémoire. Si dans l'implantation réaliste $X <_p Y$, alors dans l'implantation omnisciente $X <_p Y$. Ceci est vrai car l'ordre du programme dépend exclusivement du programme et du processeur, et qu'ils n'ont pas été modifiés. Nous notons cette implication : $ori : X <_p Y \implies om : X <_p Y$. Notons que c'est même une équivalence, mais nous nous intéressons uniquement à l'implication.

Nous voulons montrer que :

$$\forall X, Y \in E_{LD} \bigcup E_{ST}\ ori : (X <_p Y \implies X <_m Y) \implies om : (X <_p Y \implies X <_m Y)$$

Afin de montrer la validité de cette implication, nous allons diviser en 4 l'espace des solutions :

I1 $\forall X, X' \in E_{LD}\ ori : (X <_p X' \implies X <_m X') \implies om : (X <_p X' \implies X <_m X')$

I2 $\forall X \in E_{LD},\ Y \in E_{ST}\ ori : (X <_p Y \implies X <_m Y) \implies om : (X <_p Y \implies X <_m Y)$

I3 $\forall X \in E_{LD},\ Y \in E_{ST}\ ori : (Y <_p X \implies Y <_m X) \implies om : (Y <_p X \implies Y <_m X)$

I4 $\forall Y, Y' \in E_{ST}\ ori : (Y <_p Y' \implies Y <_m Y') \implies om : (Y <_p Y' \implies Y <_m Y')$

I1 : Cette implication est vraie car les étapes nécessaires à la réalisation d'une lecture dans chacun des protocoles ont les mêmes contraintes. En effet, une lecture ne peut être traitée que si la lecture précédent précédente a déjà été traitée par le contrôleur mémoire.

I2 : Cette implication est maintenue parce que dans nos deux implantations une instruction d'écriture ne peut être traitée que si le contrôleur mémoire a acquitté l'instruction de lecture précédente.

I3 : Dans notre implantation *write-through* (voir la section 6.4), une opération de lecture ne peut être envoyée qu'après l'acquittement par le contrôleur mémoire de l'opération d'écriture. Par conséquent, puisque notre implantation omnisciente ne modifie pas les étapes de traitement des requêtes de lecture/écriture, la propriété est maintenue.

Dans notre implantation *write-back*, nous datons l'opération mémoire au moment de la mise à jour de la donnée en cache. Aucune opération de lecture ne peut être réalisée avant cette étape précise, garantissant de fait l'implication précédente.

I4 : Dans notre implantation *write-through*, des écritures consécutives peuvent être envoyées en rafale au contrôleur mémoire. De plus, les écritures sont traitées dans l'ordre par le contrôleur et notre réseau d'interconnexion garantit l'ordre des paquets sur un même chemin de communication[2]. Puisque ces conditions sont maintenues dans notre implantation omnisciente, l'implication est vraie.

Dans l'implantation *write-back*, cette condition est aussi valable car les succès d'écritures sont traités dans l'ordre et que les échecs ne sont traités qu'après l'acquittement des invalidations des copies du système. Ceci est vrai que les invalidations soient effectuées de manière omnisciente ou pas.

Dans notre implantation, les propriétés de notre modèle de consistance mémoire restent valable dans notre implantation omnisciente. Néanmoins, cette nouvelle implantation possède une nouvelle propriété : chaque processeur voit les effets d'une écriture au même moment que la mémoire. Cette propriété n'était pas vraie dans l'implantation originelle.

La méthode que nous proposons ne garantit pas la justesse du modèle de consistance mémoire, ni le maintien de ses propriétés. C'est au concepteur d'assurer que son implantation omnisciente ne va pas modifier les propriétés, ce qui, bien que n'étant pas automatique est possible.

6.4 Détails de l'implantation des actions omniscientes pour des protocoles *write-through* et *write-back invalidate*.

Nous présentons ci-après, l'implantation d'un protocole de cohérence omniscient à partir de deux plateformes existantes : l'une *write-trhough* et l'autre *write-back*. Ces plateformes ne possèdent aucun protocole de cohérence.

[2]Le routage est déterministe

6.4.1 Implémentation des actions omniscientes

Les fonctions `coherent_read` et `coherent_write` sont implantées dans un module nommé `omniscient_register` (figures 6.7 et 6.8). Dans notre cas ces modules sont de simples classes C++ intégrées au modèle de simulation (SystemC CABA). L'`omniscient_register` contient une liste de chacun des composants cache ou mémoire du système. Ainsi, il est en mesure d'accéder directement à leurs structures de données (données et états).

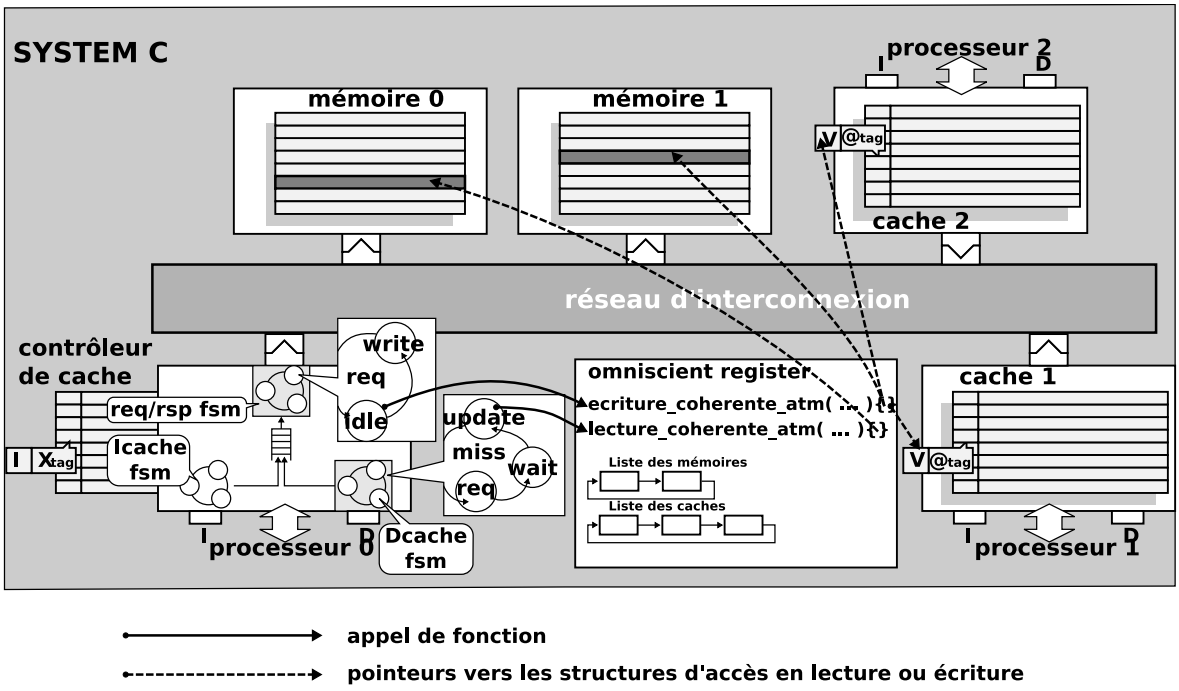


FIG. 6.7 – Vue de l'architecture d'une plateforme pourvue de la cohérence omnisciente.

Write-through : pour cette implantation, la fonction `coherent_write` met à jour la mémoire et invalide les copies du système en 0 cycle d'horloge. Cette fonction est appelée au moment où le contrôleur de cache émet la requête sur le réseau d'interconnexion. Très précisément lorsque la machine à états dédiée au traitement des requêtes est oisive et qu'une requête est présente dans le tampon d'écriture. Comme nous pouvons le voir sur la figure 6.7, le contrôleur de cache appelle la fonction `coherent_write` ❷. Cette fonction parcourt la liste des caches et invalide les copies de la donnée ❹ en comparant l'adresse d'écriture (`@2` sur la figure 6.8) avec le TAG de la ligne de cache.

La fonction `coherent_read` est appelée lorsque le contrôle de cache reçoit la réponse à sa requête de lecture ❶. Cette fonction parcourt les composants mémoire et récupère la donnée qui s'y trouve ❸.

Write-back : pour l'implantation *write-back*, nous utilisons le même module `omniscient_register` que l'implantation *write-through*. La différence réside dans l'emplacement des appels à `coherent_read/write`.

Comme pour l'implantation *write-through*, la fonction `coherent_read` est appelée lorsque la donnée lue en mémoire parvient au cache (cas d'un échec en lecture ou d'une

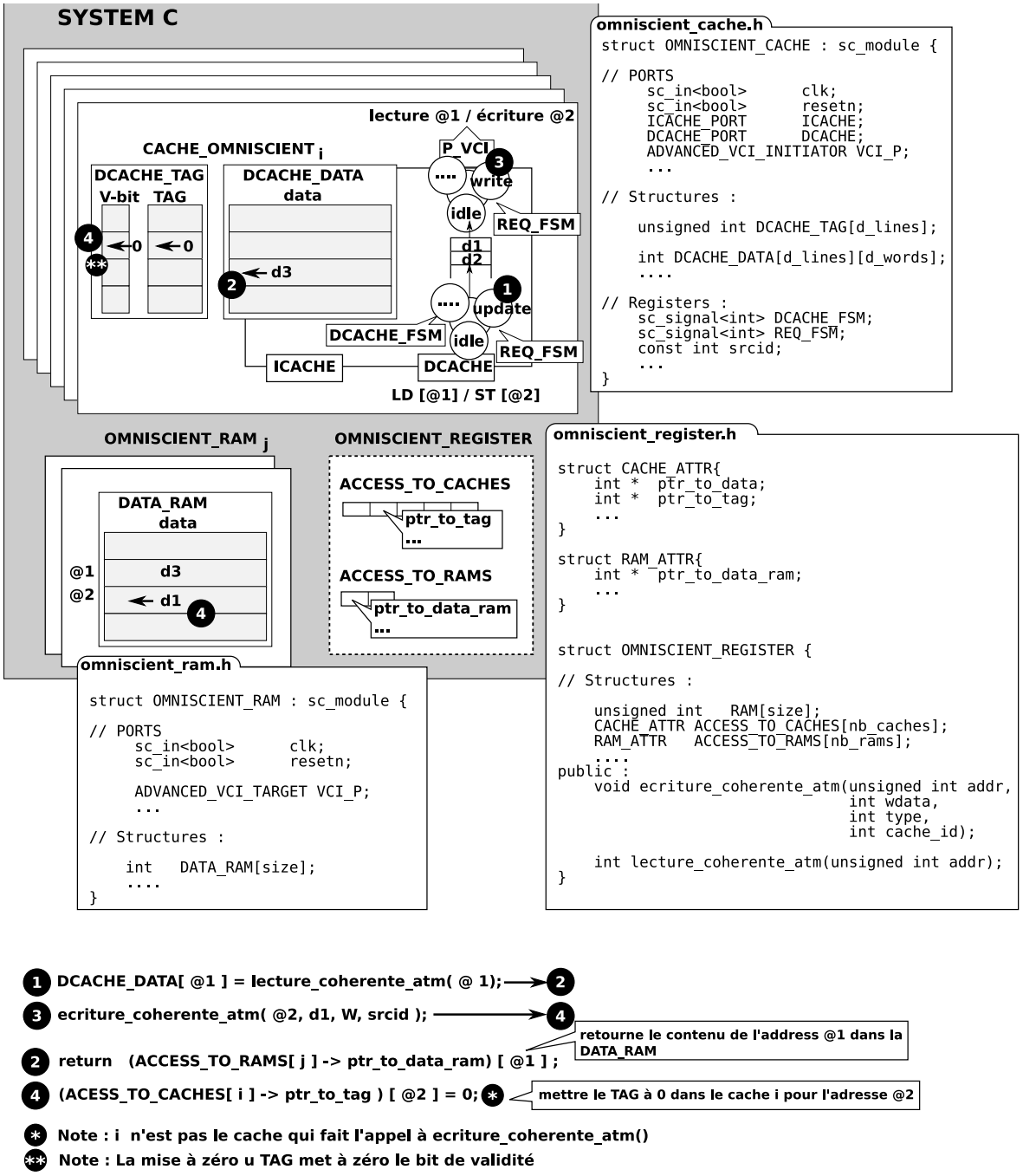


FIG. 6.8 – Structure de l'implantation d'un protocole de cohérence omniscient dans un cache *write-through*

allocation sur écriture « *write-allocate* »).

Dans cette implantation, il n'est plus possible d'appeler `coherent_write` lorsqu'une requête d'écriture est présente dans le tampon d'écriture car aucune requête n'est émise lors d'un succès en écriture (*write hit*).

Par conséquent le point d'appel a été déplacé, l'action omnisciente est réalisée lorsque la donnée est modifiée dans le cache. Ceci est le cas pour toutes les écritures (un échec en

écriture est quand même traitée *in fine* dans le cache).

Comme pour l'implantation *write-through*, toutes les copies de la donnée sont invalidées dans le système et la donnée présente en mémoire est mise à jour. Ce dernier point peut paraître étrange pour un protocole de type *write-back* mais, dans un protocole de cohérence, une lecture fournit toujours une valeur à jour de la donnée. Ici, la lecture d'une donnée va fournir la copie présente en mémoire dont la cohérence est maintenue par la fonction *coherent_write*.

Ce protocole omniscient à politique de mise à jour mémoire *write-back* n'a pas pour but d'avoir le même comportement qu'un protocole spécifique de la famille des *write-back invalidate*. Il permet d'avoir une borne inférieure, ou un meilleur cas d'exécution par rapport à tous ces protocoles.

6.4.2 Coût d'implantation de la méthode

La méthode que nous avons présenté n'a un intérêt que si l'effort à fournir pour sa mise en œuvre est très inférieur à celui qu'il faudrait pour implanter une version détaillée du protocole étudié. Nous présentons dans la table 6.1 le nombre de lignes C++ nécessaires à la description des modèles des caches, mémoires et modules de cohérence omnisciente (version *write-through*).

TAB. 6.1 – Nombre de lignes de code de chacun des composants

	cache	mémoire	autres	total
Sans cohérence	1413	395		1808
Avec cohérence omnisciente	1473	384	166	2023
Avec cohérence	1852	996		2848

Comme nous pouvons le constater, l'intégration des actions omniscientes dans des composants ne fournissant aucun mécanisme de cohérence représente 154 lignes de codes. A ceci il faut ajouter les 320 lignes de code du composant `omniscient_register`. En conclusion, le coût d'implantation du protocole omniscient est très faible pour les concepteurs des contrôleurs de cache et mémoire.

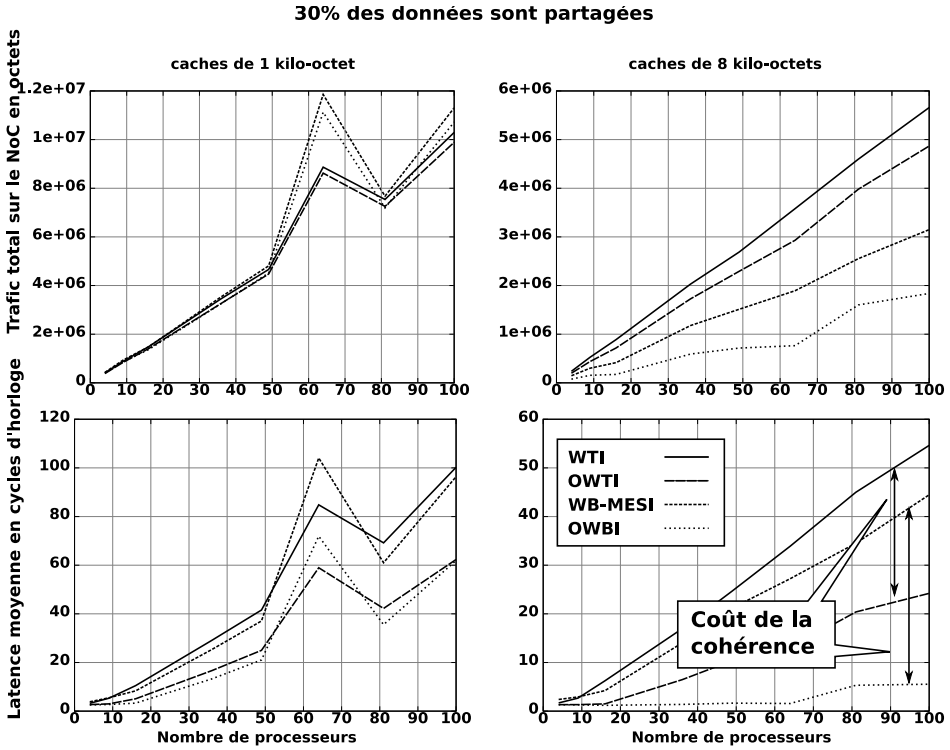
6.5 Expérimentations, résultats et commentaires

Pour évaluer notre proposition, nous allons comparer deux protocoles de cohérence et leurs homologues omniscients. Nous allons par conséquent comparer quatre protocoles, aux deux protocoles comparés dans le chapitre 4 nous ajoutons : *omniscient write-through invalidate* (OWTI) et *omniscient write-back invalidate* (OWBI).

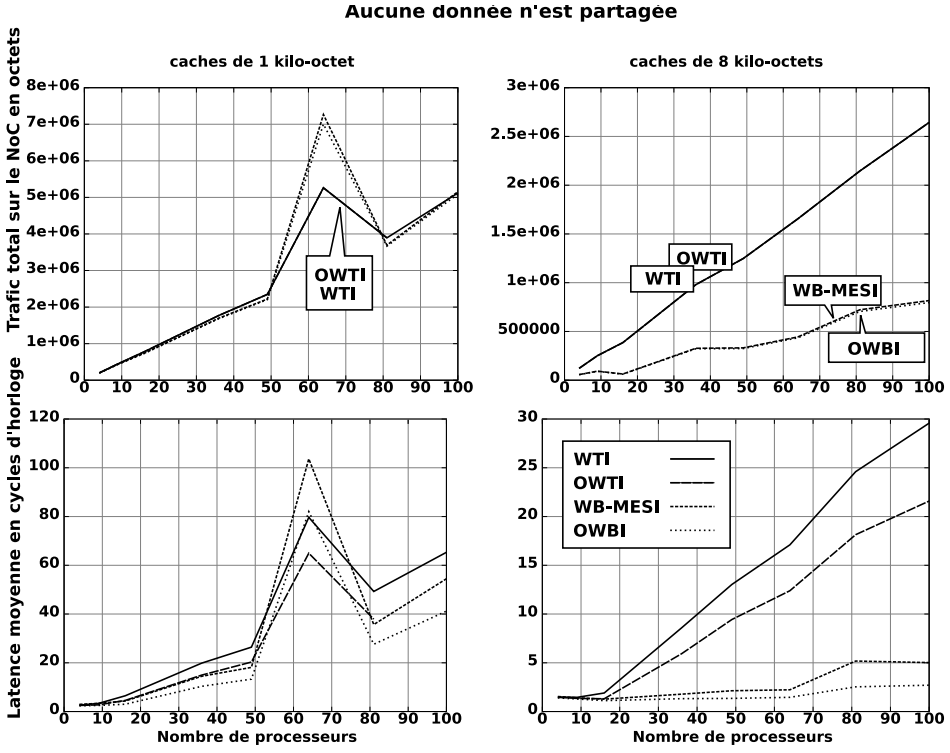
Nous reprenons l'environnement de simulation, les architectures et les applications déjà présentées dans le 4^{e}chapitre.

6.5.1 simple_test

Comme précédemment cette application a été exécutée sur des plateformes contenant de 4 à 100 processeurs. Les résultats sont présentés dans les figures 6.9 et 6.10.

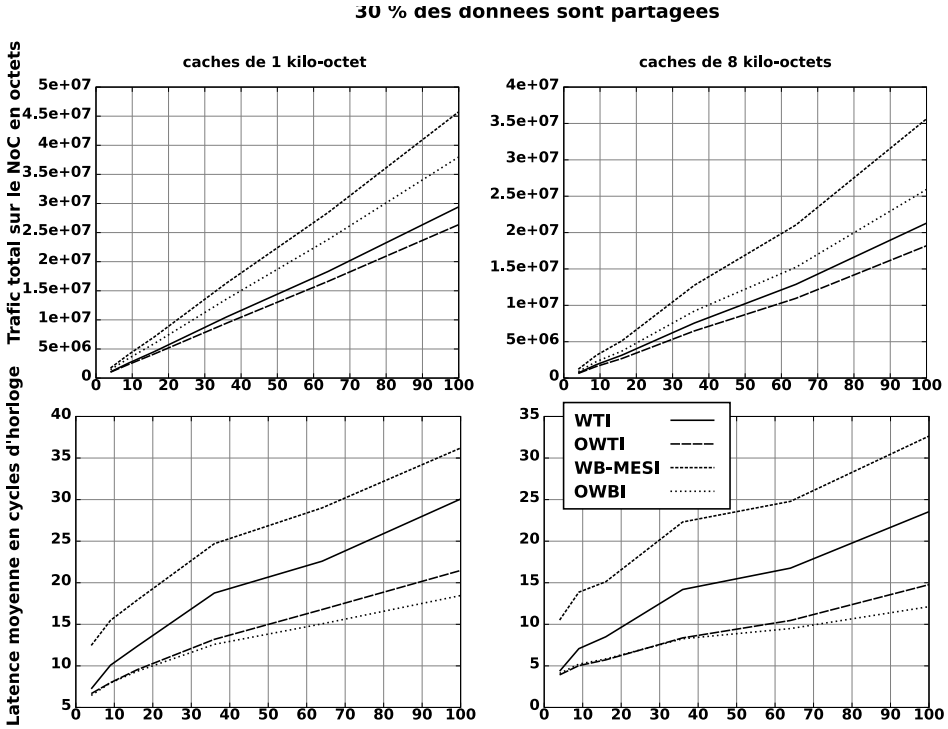


(a)

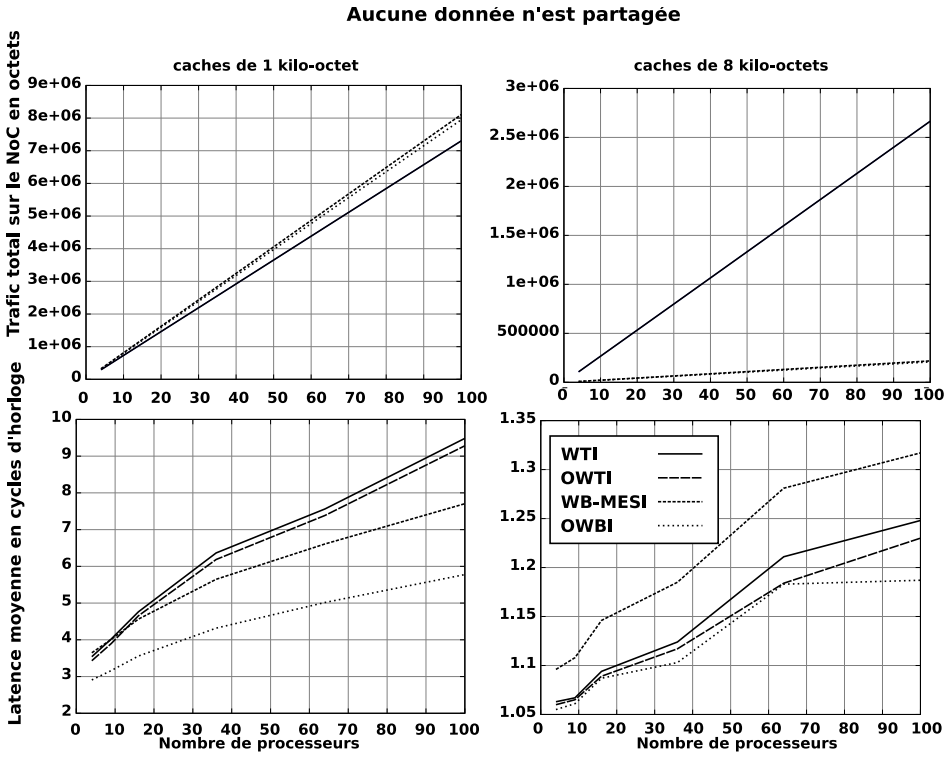


(b)

FIG. 6.9 – Trafic et latence moyenne pour l'application `simple_test` sur l'architecture centralisée.



(a)



(b)

FIG. 6.10 – Trafic et latence moyenne pour l'application `simple_test` sur l'architecture décentralisée.

6.5.1.1 Résultats sur l'architecture centralisée

Latence d'accès : les différences observées entre un protocole et son homologue omniscient correspondent au coût en latence du maintien de la cohérence.

Nous remarquons que ce coût est très élevé pour les caches de 1 kilo-octet et peut atteindre les 50% de la latence moyenne des accès. Avec des caches de plus grande taille le coût est plus faible, ceci est un effet de bord d'un nombre d'accès moins élevé à la mémoire. En effet, l'envoi des requêtes d'invalidations par le contrôleur mémoire et l'attente des réponses diminue le débit maximum des requêtes que peut traiter le contrôleur. Par conséquent, si un banc mémoire ne peut traiter toute les requêtes qui lui sont envoyées, l'ajout des délais d'invalidations contribue à faire croître de façon démesurée la latence des accès. Nous observons ici le même phénomène que sur un réseau de communication qui arrive à saturation. Nous observons également que lorsqu'il n'y a pas de partage de données, le protocole omniscient n'offre pas exactement les mêmes performances que son homologue. Ceci est dû à certains détails de la micro-architecture qui ne sont pas identiques dans les deux protocoles. Par exemple, l'interrogation du répertoire (réalisée à chaque accès) par le contrôleur mémoire prend un cycle d'horloge.

Comme escompté, les protocoles omniscients fournissent une borne inférieure des latences d'accès à la mémoire (à l'exception du point discordant à 64 processeurs).

Trafic généré : le trafic de cohérence est plus élevé pour le protocole WB-MESI que pour le protocole WTI. L'explication est simple et sans surprise, l'invalidation d'une ligne génère moins de trafic que sa recopie en mémoire et son allocation dans un autre cache (celui qui a réalisé l'écriture).

Lorsqu'aucune donnée n'est partagée, aucun trafic de cohérence n'est généré pour les protocoles WTI/OWTI. Cependant, pour les protocoles WB-MESI/OWBI nous observons une légère différence pour la plateforme à 64 processeurs.

Cette différence s'explique par le fait que dans le protocole WB-MESI, lors d'un échec de lecture en cache la donnée est allouée dans un état potentiellement *exclusif* (E, cf. figure 4.1(b)). Ceci est également vrai pour les instructions, bien que leur cohérence ne soit pas maintenue dans notre architecture, le répertoire ne fait pas la différence avec les autres données. Si un autre processeur accède à la même donnée, ou plus exactement à la même ligne de cache, alors une requête d'invalidation spécifique sera envoyée pour informer le propriétaire *exclusif* que l'état de la ligne doit passer à *partagé* (S).

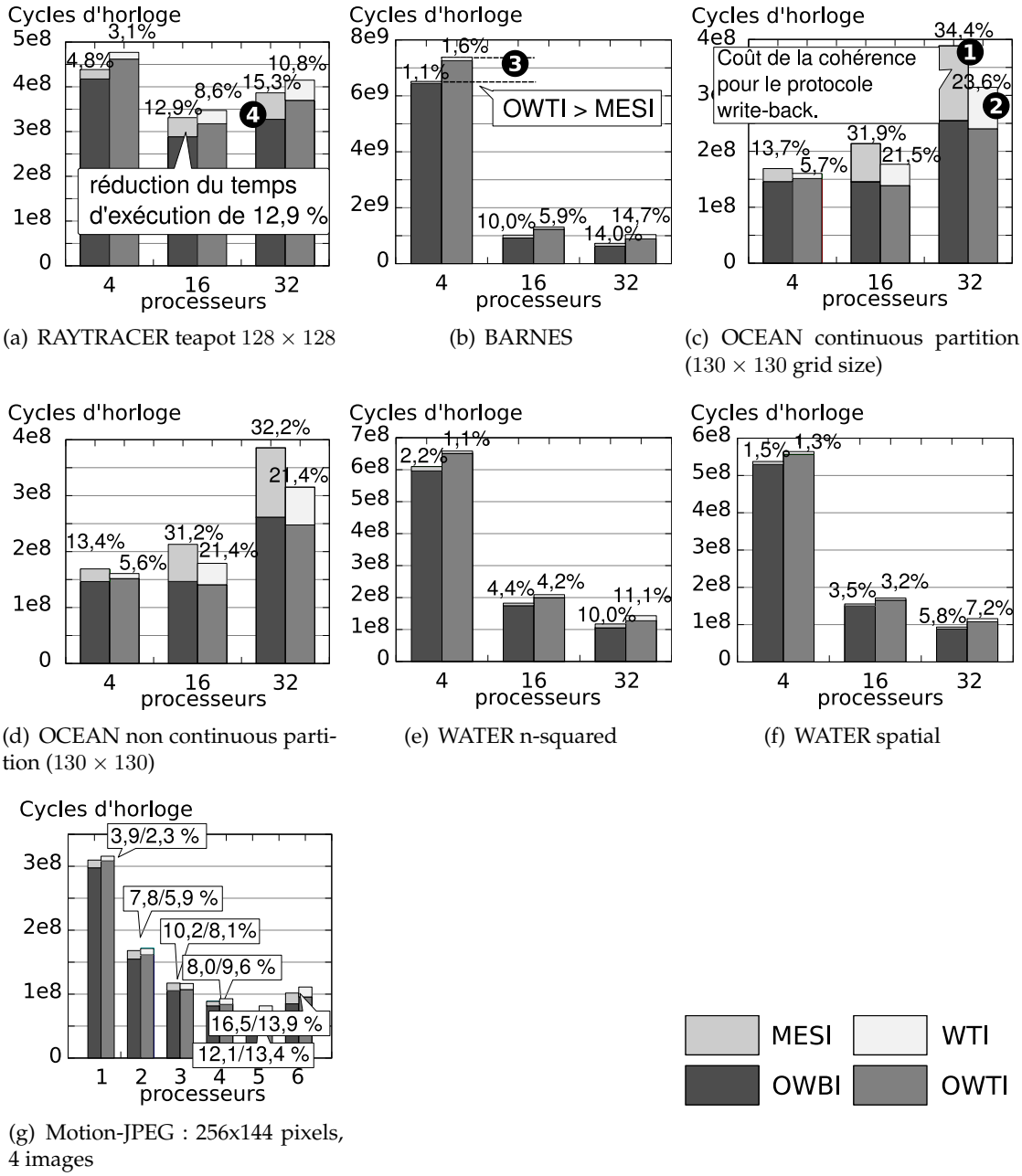
6.5.1.2 Résultats sur l'architecture décentralisée

Le placement des bornes inférieures obtenues ne nous permettent pas de conclure sur la performance relative des protocoles, les résultats obtenus correspondent au cas « a » que nous avons présenté dans la figure 6.1(a).

Toutefois, les protocoles omniscients fournissent bien une borne inférieure sur les latences et le trafic générés.

6.5.2 Applications SPLASH-2 et MJPEG

Nous présentons dans les figures 6.12 et 6.11 les résultats obtenus avec les différents protocoles de cohérence.



(a) RAYTRACER teapot 128 × 128

(b) BARNES

(c) OCEAN continuous partition (130 × 130 grid size)

(d) OCEAN non continuous partition (130 × 130)

(e) WATER n-squared

(f) WATER spatial

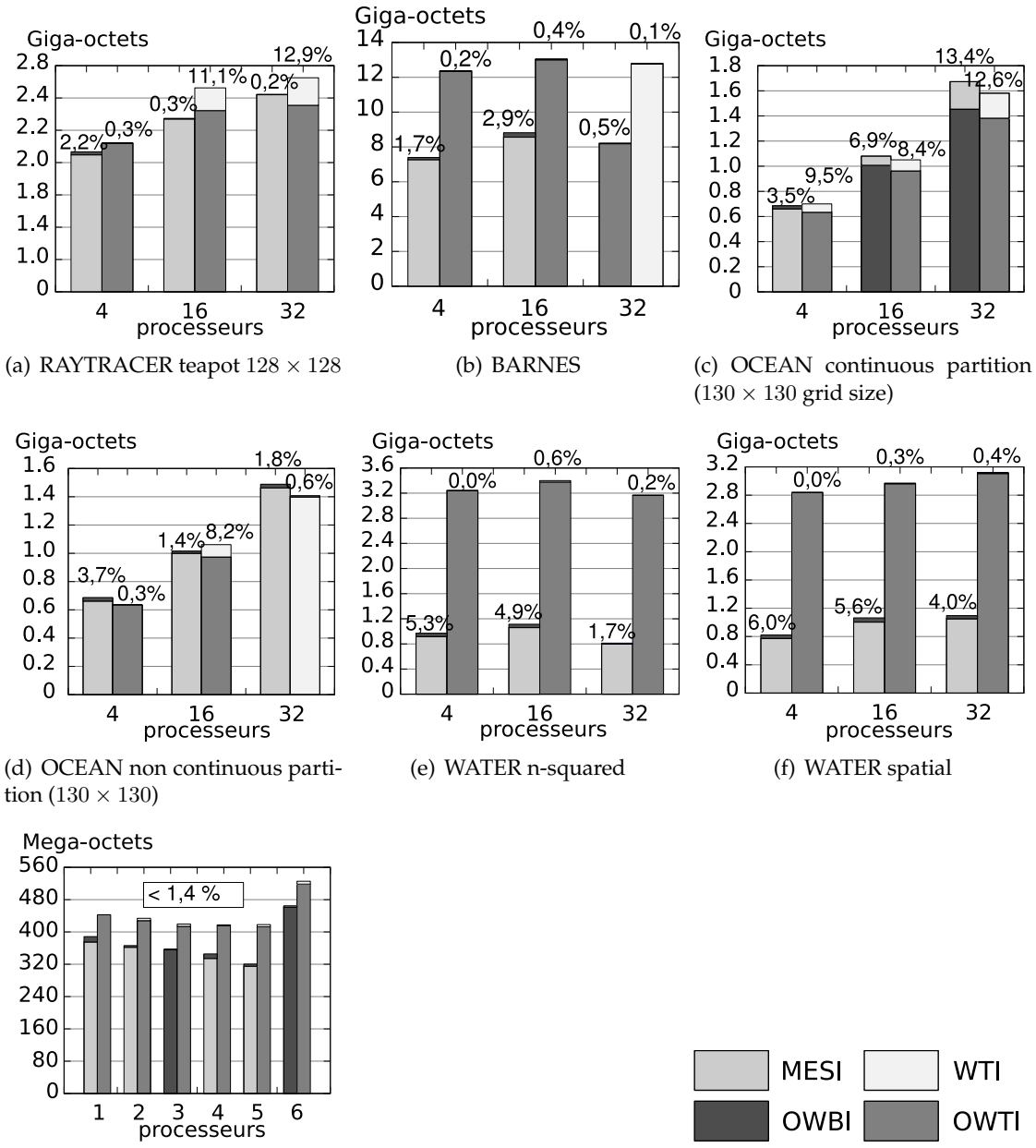
(g) Motion-JPEG : 256x144 pixels, 4 images

FIG. 6.11 – Temps d'exécution total des applications.

6.5.2.1 Temps d'exécution :

Nous observons pour l'ensemble des applications que le coût du maintien de la cohérence en WB-MESI est supérieur à celui du protocole WTI. Ces résultats confirment un des avantages du protocole WTI que nous avions montré précédemment.

Pour l'application `barnes` nous avons des résultats très intéressants : le protocole OWTI offre de moins bonnes performances que le protocole WB-MESI ❶(cf. figure 6.11(b)). Ceci prouve qu'il n'est pas possible d'optimiser le protocole WTI de façon à ce qu'il surpasse le protocole WB-MESI pour cette application.



(a) RAYTRACER teapot 128 × 128

(b) BARNES

(c) OCEAN continuous partition (130 × 130 grid size)

(d) OCEAN non continuous partition (130 × 130)

(e) WATER n-squared

(f) WATER spatial

(g) Motion-JPEG : 256x144 pixels, 4 images

FIG. 6.12 – Cumul du trafic généré, il inclue toutes les requêtes et toutes les réponses émises sur le réseau d'interconnexion.

6.5.2.2 Trafic généré :

Pour la plupart des applications nous observons qu'il y a très peu de trafic lié au maintien de la cohérence des données. Nous rappelons que le trafic mesuré inclut toutes les requêtes qui passent sur le NoC dont les instructions. Ceci contribue à minimiser l'impact du trafic lié à la cohérence, même pour une application très dynamique comme le MJPEG. En effet, mis à par l'application de décodage vidéo, les tâches ne migrent pas entre les processeurs et les structures de données du système d'exploitation ne sont pas partagées par les différents

processeurs.

Les protocoles omniscients doivent fournir une borne inférieure des coûts d'implantation, et pourtant, nous remarquons que sur plusieurs applications le protocole OWBI génère plus de trafic que le protocole WB-MESI. Dans le cas de l'application `water spatial` cette différence est importante et atteint les 6%. Nous analyserons et détaillerons les causes de ce phénomène dans la section suivante qui présente les limitations de cette technique.

6.6 Limitations : une borne inférieure « approchée »

L'approche que nous avons proposé fournit une borne inférieure sur le coût en performances de l'implantation sous certaines contraintes que nous détaillons ci-dessous.

6.6.0.3 L'implantation des étapes communes d'un accès à la mémoire doivent être équivalentes

Considérons par exemple la charge de trafic, la logique voudrait que le protocole qui n'émet pas de requêtes de cohérence sur le réseau d'interconnexion (protocole omniscient) génère moins de trafic qu'un protocole standard. Toutefois ceci peut ne pas être le cas comme l'attestent nos résultats sur le protocole WB-MESI.

Dans notre implantation du cache *write-back invalidate* sans maintien de la cohérence des données il n'existe que deux types de requêtes : lecture ou écriture d'une ligne mémoire. Dans le cas d'un échec d'écriture en cache, une ligne est allouée à l'aide d'une lecture. Celle-ci induit l'envoi d'une requête de 8x4 octets (lignes de 8 mots) et de la réponse associée de 32 octets également, soit un total de 64 octets. Dans notre implantation du protocole WB-MESI la requête correspondante à un échec en écriture ou sur une donnée partagée (S) fait 1x4 octets. Par conséquent, le protocole OWBI va générer plus de trafic que le protocole WB-MESI. Ceci explique les résultats obtenus dans les figures 6.12(a),6.12(b)(4,16 processors), 6.12(c),6.12(d),6.12(e),6.12(f),6.12(g).

Cette optimisation au niveau des accès à la mémoire a été possible pour deux raisons, la première est le standard de communication utilisé (BVCI ou AVCI[3]) et la deuxième est que le contrôleur mémoire fait la différence entre la lecture de lignes et la demande d'exclusivité sur une ligne (nécessaire à une écriture).

Afin d'éviter ce problème nous aurions pu appliquer la même optimisation au protocole OWBI, néanmoins cela aurait demandé des modifications dans la façon de traiter les requêtes qui ne sont pas liées au protocole de cohérence. Le but de ces travaux est également de montrer les limites de cette approche.

Finalement, lors de l'implantation d'un protocole omniscient il faut s'assurer que le traitement des actions communes aux deux protocoles d'accès à la mémoire sont implantés de façon identique.

6.6.1 Le chemin d'exécution peut être modifié

Lorsque l'on compare deux architectures multiprocesseurs légèrement différentes, il faut être attentif au problème de « l'effet papillon » (cf. figure 5.1). Dans un environnement concurrentiel où le chemin d'exécution dépend de données partagées, il est impossible de garantir que deux architectures différentes produiront le même chemin d'exécution. Ainsi, le retard de quelques cycles d'horloge d'une requête mémoire peut provoquer l'échec d'une prise de verrou, une commutation de contexte ou encore la migration d'une tâche.

[3] AVCI autorise un nombre différent de *flits* dans une requête et la réponse associée

Les performances que l'on compare subissent ces aléas et peuvent parfois produire des résultats inattendus. Pour comparer nos protocoles omniscients nous avons délibérément empêché la migration des tâches afin de minimiser ce problème. Toutefois, bien que son impact soit moindre il est toujours présent sur toute prise de décision qui dépend directement ou indirectement d'une donnée partagée.

6.7 Conclusion de l'étude

Nous avons présenté dans ce chapitre une nouvelle solution permettant d'évaluer le coût d'implantation d'un protocole de cohérence mémoire. Ce coût est mesuré en comparant des architectures qui utilisent soit un protocole réel, soit un protocole omniscient. Ce dernier réalise les actions de cohérence en un temps nul et permet de garantir un accès cohérent aux données.

Un protocole omniscient permet d'obtenir une borne inférieure sur le coût d'implantation d'un protocole. Il permet d'évaluer le gain maximum en performance qu'il est potentiellement possible d'obtenir en optimisant le protocole existant. Il permet donc d'évaluer la pertinence d'une coûteuse recherche d'optimisation d'un protocole réel.

Cette approche permet également de comparer à moindre coût des nouveaux protocoles de cohérence, ou bien encore, de fournir la cohérence mémoire à une architecture qui en est dépourvue. Nous avons également soulevé certaines limitations de l'approche qui a pour conséquence de fausser l'obtention de la borne inférieure. Néanmoins, cette approche nous a permis de mettre en évidence des situations où le coût de cohérence est marginal, ou bien où la borne inférieure est un mauvais résultat.

La simulation basée sur une approche omnisciente est un outil puissant, utilisable uniquement en simulation, mais qui pourrait être utilisée pour l'évaluation de choix d'architecture ou micro-architecture autres que les protocoles de cohérence mémoire.

Deuxième partie

Chapitre 7

Mémoire, placement et gestion des données

Dans la problématique, nous avons exposé le problème de l'accès aux données. Nous avons également montré qu'un des points majeur est la pénalité d'accès lors d'un échec en cache. Dans ce chapitre nous allons présenter les différentes techniques et mécanismes existants pour placer, déplacer et gérer les données.

L'enjeu principal est d'améliorer le temps d'accès et donc, par voie de conséquence, les performances du système. Toutefois, certaines solutions présentées ici sont dédiées à améliorer la gestion de l'énergie.

Nous considérerons deux catégories d'architectures, celles qui sont distribuées sur plusieurs puces, cartes ou machines ; et celles qui sont intégrées dans une même puce. Les architectures distribuées existent depuis les débuts de l'informatique et sont à l'origine du calcul parallèle.

7.1 Architectures distribuées

Les architectures distribuées sont constituées par un ensemble de nœuds qui contiennent un ou deux processeurs, de la mémoire cache et une mémoire partagée par l'ensemble du système. Cette mémoire contient un sous-ensemble de l'espace d'adressage. La cohérence des caches est en général maintenue par le matériel à l'aide de répertoires. Cette famille d'architectures est classée dans de nombreux travaux sous le nom d'architecture cc-NUMA (cache coherent Non Uniform Memory Access).

Comme nous l'avons présenté dans la figure 7.1, les composants sont distribués sur un ensemble de puces ou cartes interconnectées entre-elles. Le temps d'accès à la mémoire d'un nœud voisin est bien supérieur au temps d'accès à une mémoire locale. Dans la machine Wildfire ce temps est 6 fois supérieur[NvdP99]. Par conséquent, le placement des données dans le système a une influence considérable sur les performances. De fait, une multitude de solutions ont été proposées afin de maximiser la localité des données et réduire la latence des accès.

Nous présentons ci-dessous les différents solutions existantes liées au placement et déplacement des données dont certaines sont appliquées à la gestion de la consommation d'énergie.

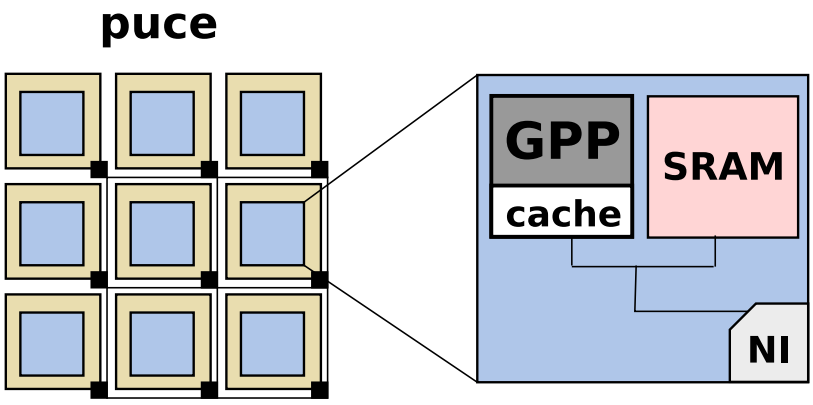


FIG. 7.1 – Exemple d'architecture distribuée comprenant quatre nœuds.

7.1.1 Placement et déplacement des données

Dans un système où les tâches et processus n'ont pas la possibilité de migrer en dehors du nœud (potentiellement multiprocesseur) sur lequel ils ont été placés, le placement initial des données est déterminant. Celui-ci est réalisé par le système d'exploitation. Bien souvent, ces systèmes possèdent un mécanisme de mémoire virtuelle et le système a la charge d'allouer les pages physiques associées.

Afin d'optimiser le placement des données, les systèmes allouent les pages physiques sur le banc mémoire local au nœud qui en a fait la demande.

L'accès aux données devient problématique lorsqu'elles sont partagées par des tâches se trouvant sur des nœuds différents, ou bien, lorsque les tâches peuvent migrer entre les différents nœuds. Afin de résoudre ce problème, les architectes et concepteurs des systèmes ont proposés des solutions permettant de déplacer ou de dupliquer les données entre les différents nœuds du système. Certaines solutions sont implantées en matériel et sont transparentes à l'utilisateur, d'autres sont intégrées au système d'exploitation.

7.1.1.1 Solutions implantées en matériel

Une solution simple est l'usage de caches. Ceux-ci, permettent de placer à proximité des processeurs les données les plus utilisées et permettent ainsi de diminuer la latence moyenne des accès.

La mémoire cache a la particularité d'être transparente à l'utilisation, et de gérer les données à un grain très fin (l'unité étant de quelques mots). La mémoire locale est adressable par l'utilisateur et a une capacité bien plus grande. Les architectes ont donc proposé une solution qui permet de combiner les avantages des deux solutions, les architectures COMA dont [LH91] est une des premières implantations.

L'idée est d'utiliser une partie de la mémoire locale comme un cache de données. Pour cela, le système d'exploitation ne pagine pas l'ensemble de la mémoire physique disponible afin de conserver un espace pour des données dupliquées.

Cette mémoire dénommée *Attraction Memory* possède, à l'instar d'un cache, un TAG qui permet d'identifier de façon unique chacune des lignes mémoire. Chaque ligne possède une adresse de résidence fixée. Dans l'implémentation [SJG92] flat-coma, lorsqu'une donnée n'est pas présente en cache, elle est recherchée dans son nœud de résidence. Si elle ne s'y

trouve pas, ce dernier connaît son adresse et propage la requête. Lorsqu'elle est trouvée, une copie est placée dans le cache et dans la mémoire locale.

La difficulté majeure d'une telle solution est d'être en mesure de garantir l'existence d'au moins une copie de chaque donnée dans le système. Par conséquent, l'évincement d'une ligne est une opération complexe. De plus, le surcoût lié à la réplication est non négligeable.

Une autre solution matérielle est l'usage d'un cache de troisième niveau qui n'est interrogé que si il y a eu un échec de cache et que l'adresse n'est pas contenue dans la mémoire locale au nœud. Ce cache implémenté en D-RAM est appelé *Remote Cache* et a été utilisé dans les machines Sting [LC96] et S3.mp [NAB+95]. Ces solutions ont été comparées dans [ZT97] et donnent l'avantage à la deuxième solution par sa performance et son faible coût d'implantation.

7.1.1.2 Solutions logicielles ou hybrides

Afin de réduire le coût d'implantation des architectures COMA, les solutions Simple-COMA [SWCL95] puis MS-COMA [BT98] ont déporté une partie du protocole au niveau logiciel. Lors d'un échec de données dans la mémoire locale (*Attraction Memory*), une exception est levée et une page est allouée par la MMU. Les données sont copiées ou déplacées ligne par ligne vers cette page par le matériel. Puisque l'adressage des données est garanti par la MMU, la présence du TAG n'est plus nécessaire.

Une proposition hybride permettant d'exploiter les qualités des différentes solutions matérielles et logicielles (S-COMA et cc-NUMA) a été explorée [FW97, LF00].

D'autres solutions logicielles visent à déplacer ou dupliquer des pages entre les mémoires du système. D'une manière générale, des compteurs d'accès sur les pages, TLB ou échecs de caches sont utilisés pour identifier les pages candidates à la migration ou à la réplication. Ces compteurs peuvent déclencher une interruption (SGI Origin [LL97]) ou bien, être relevés régulièrement par une tâche spécifique du système d'exploitation (Wildfire [NvdP99]). Plusieurs études sur l'influence de la migration et/ou réplication des pages ont été menées et comparées aux solutions existantes [il99, CDV+94, VDGR96] et [CML03]

Ces études montrent que la migration et la duplication des pages permettent de résoudre le problème du placement des données partagées et de la migration des tâches dans le système. Ces études montrent également que l'usage de la migration comme seule solution (sans duplication) ne donne pas de très bons résultats. En effet, la migration a un coût supérieur à la duplication et peut être préjudiciable pour des données partagées. Notons que le placement des pages se fait dans tous les cas sur le nœud qui en fait la demande, créant ainsi potentiellement des effets *ping-pong*. L'étude [CML03] de la machine SGI Origin 2000 montre également un des défauts des solutions gérées par le système d'exploitation : le manque de réactivité. En effet, si les tâches migrent de façon trop rapide, la migration des données est toujours en retard et dégrade les performances du système.

En partant du constat que la migration des pages est probablement plus efficace si elle est réalisée par le matériel, F.Bellosa [Bel04] propose de réaliser l'allocation des pages et la migration par un composant spécifique. Celui-ci est en réalité un petit processeur exécutant un micro-OS qui reçoit et traite les requêtes du système. Bien que la solution ait l'inconvénient majeur d'être centralisée, elle propose l'idée intéressante d'ajouter un deuxième niveau d'adressage afin de placer et déplacer les données à l'insu du système.

La migration des données peut être également reportée sur l'application comme le propose la solution [NPP+00]. L'argument est que l'application connaît mieux que le système ses besoins en terme de placement de données. Ceci peut être géré au niveau d'un *runtime* à l'aide d'une API fournie par l'OS. Le *runtime* peut être intégré au processus de compilation

afin de ne pas être une charge pour le programmeur. Une telle solution s'applique à chaque application individuellement à condition qu'elles fassent usage de ce mécanisme.

7.1.2 Gestion de l'énergie

Bien que le placement et déplacement intelligent des données permettent d'obtenir un gain en performance d'exécution, ils permettent également de réduire la consommation d'énergie d'un système.

Les problèmes de consommation d'énergie sont bien connus dans les systèmes embarqués où l'autonomie est un enjeu de premier ordre. Ils sont également présents dans les systèmes haute performance où la réduction de la consommation se traduit par une baisse des coûts d'exploitation. L'énergie consommée par ces systèmes a un coût direct (chaleur dégagée) mais également un coût indirect : usure prématurée des composants et évacuation des calories.

Contrairement à la recherche de performances d'exécution où les données sont dispersées dans le système et sont placées de manière à réduire les latences d'accès ; la diminution de la consommation requiert de conserver en activité un nombre minimum de bancs mémoire. Les autres bancs sont placés en mode basse consommation et ne sont réactivés que si c'est nécessaire.

L'idée principale est donc de réaliser, si possible, les allocations dans un même banc mémoire. Puis, de s'adapter au comportement du système afin d'agglomérer dynamiquement les données actives du système. Dans [LFZE00] les auteurs s'inspirent pour cela des techniques coopératives entre le logiciel et le matériel (compteurs d'accès) afin de réduire la consommation d'énergie du système.

Des solutions plus performantes, mais également plus complexes ont été proposées [HPS03, MGP+03]

7.2 Architectures intégrées

La plupart des architectures intégrées sont comparables à celle que nous présentons dans la figure 7.2.

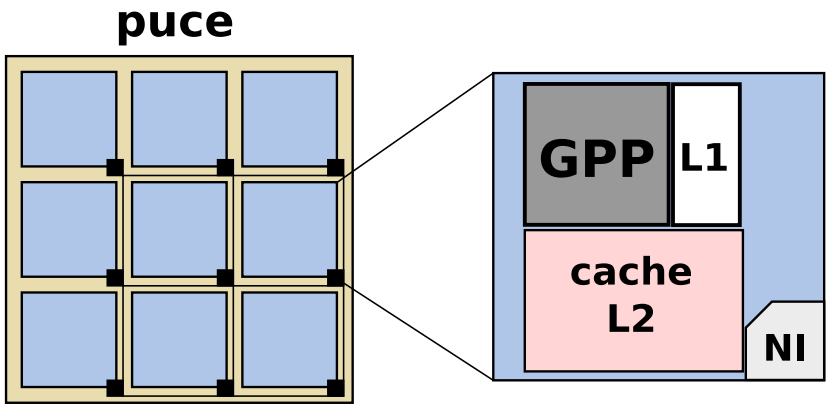


FIG. 7.2 – Exemple d'architecture intégrée comprenant quatre nœuds.

Ces architectures embarquent une grande quantité de mémoire cache et sont orientées vers le calcul haute-performance. La cohérence mémoire est maintenue à l'aide de répertoires distribués sur l'ensemble des caches de second niveau (L2).

Le placement et le déplacement des données sur ces architectures est un problème complexe dont les solutions sont complémentaires à celles présentées précédemment. Ces solutions méritent d'être examinées car notre contexte d'étude est à mi-chemin entre ces deux types d'architectures.

Néanmoins, nous faisons l'impasse dans cette étude des architectures intégrées ne possédant pas de caches, dont les problématiques sont bien différentes de celles qui nous concernent.

7.2.1 placement et déplacement des données

La mémoire cache a la propriété de fournir un temps d'accès très réduit. Cette affirmation devient toute relative dans le contexte actuel. En effet, les capacités d'intégration permettent d'intégrer dans la puce plusieurs méga-octets de mémoire cache.

Dans un petit cache le temps d'accès est constant, le chemin critique du délai de transmission pour la donnée la plus éloignée permet de maintenir la fréquence de fonctionnement identique à celle du processeur. Dans un cache de grande taille, les délais de transmission ne permettent plus de maintenir une fréquence de fonctionnement élevée à moins de réaliser les accès en plusieurs cycles d'horloge.

Une solution est d'avoir un temps d'accès variable à l'ensemble des données en fonction de leur placement . Afin d'améliorer la latence moyenne des accès des politiques de placement et migration de données sont mises en œuvre [KBK02, CPV05] pour optimiser le placement des données fréquemment accédées.

Dans les architectures multiprocesseurs, ce problème prend une autre dimension. Le cache L2 peut être partagé par les différents processeurs, ou bien, être divisé en parties de tailles égales et privées à chaque processeur. La première solution permet d'avoir une capacité de stockage sur la puce plus élevée et donc, de minimiser les accès à la mémoire externe. La seconde solution permet d'avoir des caches plus petits et donc d'avoir des latences d'accès plus faibles. En contrepartie, la capacité de stockage est réduite et un plus grand nombre d'accès externes vont être réalisés. L'application des solutions précédentes a pour effet de placer les données au centre de la puce et donc d'avoir des performances médiocres [BW04].

Les études montrent qu'avec un nombre de processeur croissant l'usage d'un cache L2 partagé n'est pas envisageable. Les architectes ont donc développé des techniques de coopération entre les caches privés de second niveau. Cette coopération permet de déplacer les données en fonction de leur fréquence d'accès[CS06], de contrôler le taux de duplication des données partagées [BMW06], ou bien encore de moduler la capacité des différents caches L2 en fonction des besoins des nœuds [CPV05].

Dans le domaine des caches, des études concernent le placement efficace des données dans les architectures multiprocesseurs. Dans [MH07] les auteurs présentent une solution où le protocole de cohérence est divisé en deux niveaux hiérarchiques afin de créer des zones de partage optimisées sur la puce. Chaque zone va regrouper des données potentiellement partagées appartenant à des tâches d'une même application. L'usage de structures spécifiques et de la duplication des répertoires permet au protocole de premier niveau d'assurer localement la cohérence des données et donc de réduire le coût du maintien de la cohérence. Notons que cette solution requiert un support du système d'exploitation.

Dans [HCM09] nous trouvons des travaux qui présentent une solution très similaire à la notre. Ces travaux ont été publiés après notre étude prospective et ce n'est que très récemment que nous en avons pris connaissance. Il convient toutefois de préciser que le contexte

d'étude n'est pas le même, notre solution ne s'applique pas à des caches de second niveau mais à de la mémoire partagée et distribuée dans la puce. Ceci implique des contraintes et des difficultés spécifiques au niveau de l'adressage des données.

7.3 Positionnement de nos travaux

Nos travaux se positionnent à mi-chemin entre les techniques de migration de données entre les caches, et les techniques de migration de données entre les mémoires.

A l'instar des solutions que l'on trouve dans les caches, nous réalisons le placement et le transfert des données par le matériel. Aucun support du système d'exploitation n'est requis. L'avantage est d'être plus réactif qu'une solution logicielle et d'être applicable indépendamment du logiciel et du système d'exploitation.

Par ailleurs, nous allons transférer les données par pages comme nous l'avons vu dans les solutions logicielles. De plus, notre solution concerne le placement et la migration des données adressables qui, à l'exception des architectures COMA, sont toujours gérées au niveau logiciel.

Chapitre 8

Migration des données et choix du placement idéal

Dans ce chapitre nous allons aborder les problèmes du placement des données d'une manière indépendante du système ou de l'application. Nous désignons par donnée un élément de la mémoire qui peut contenir aussi bien des instructions que des données au sens logiciel. Ces dernières seront désignées comme « données logicielles ». La taille de la donnée n'est pas définie ici, une donnée peut donc contenir des données logicielles et des instructions.

8.1 Le partage des données

Si l'on regarde les données au sens logiciel manipulées par un système, seul un petit nombre d'entre elles sont effectivement partagées par plusieurs tâches. Néanmoins, les données au sens général ont un taux de partage bien supérieur. En effet, il y a partage de données dès lors que deux copies d'un même emplacement mémoire existent dans le système. Nous détaillons ci-dessous les différents types de partage des données.

8.1.1 Le partage des instructions

Les instructions d'une application multi-tâche sont partagées dès lors que plusieurs tâches vont faire appel à une même fonction (cf. figure 8.1 ❶). Dans une application de type scientifique où n tâches vont exécuter le même travail, le code est massivement partagé. À cela, nous devons ajouter le partage du code du noyau et des librairies partagées.

L'influence du partage des instructions est moindre que celui des données logicielles pour deux raisons :

1. Le code ne requiert que très rarement[1] le maintien de la cohérence.
2. Les instructions ont une localité spatiale et temporelle bien supérieure à celle des données. Les taux d'échec en cache sont donc très inférieurs à ceux des données logicielles et sont globalement très faible. En conséquence, les architectes interviennent essentiellement à l'optimisation de l'accès aux données logicielles.

Néanmoins, même avec un taux d'échec de l'ordre de 1%, une pénalité d'échec de 100 cycles fait doubler la latence moyenne des accès aux instructions.

[1] Code automodifiant, édition de liens dynamique

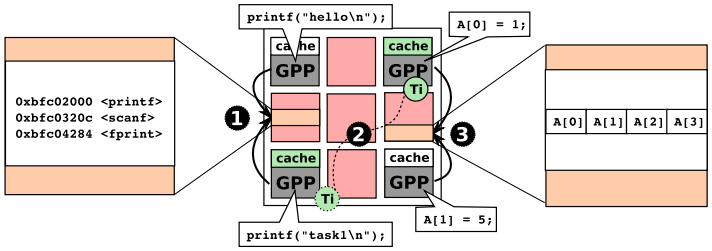


FIG. 8.1 – Illustration des différents types de partage des données

8.1.2 Persistance des données en cache

La persistance des données en cache provoque du partage de données dès qu'une tâche migre et change de processeur (cf. figure 8.1 ❷). Si le système d'exploitation ne vide pas les caches[2], alors les données seront partagées par les deux caches le temps qu'elles soient invalidées ou évincées du premier. Si le système d'exploitation vide les caches, il se peut que les répertoires de la mémoire conservent une information de partage[3]. Cette situation est également temporaire et peu à peu les données vont être invalidées.

8.1.3 Le faux partage des données

Si l'on considère qu'une donnée a une taille supérieure à un mot machine, allant de la ligne de cache à une page mémoire, il est très probable d'avoir du faux partage par *voisinage*. Pour cela il suffit que deux processeurs travaillent sur des données au sens logiciel qui se trouvent dans le même emplacement mémoire (cf. figure 8.1 ❷). Le cas d'école le plus dramatique est celui d'un tableau de données partagé par plusieurs tâches. Chaque tâche va travailler sur un sous-ensemble du tableau, mais l'organisation du tableau en mémoire va provoquer le partage des lignes de cache par les différents processeurs.

En prolongeant le raisonnement à des structures plus grandes, nous pouvons avoir des structures de données privées qui se retrouvent dans le même emplacement mémoire. Celui-ci est donc effectivement partagé par plusieurs tâches. De même, l'allocateur mémoire peut allouer côte-à-côte des données d'applications différentes.

8.1.4 Les données : un « tout »

Afin d'améliorer les accès à la mémoire nous devons considérer les données comme un tout. Nous chercherons ainsi à améliorer le placement de tous les types de données, mais également à prendre en compte tous les types de partage indistinctement.

Le partage des données a une importance primordiale quand au choix du placement. Il s'agit d'améliorer les performances de l'ensemble du système en plaçant les données au mieux pour l'ensemble des processeurs qui se les partagent.

8.2 CPI et placement des données

Vis-à-vis d'un nœud de calcul en particulier, le placement optimal des données est celui qui permettra de minimiser le nombre moyen de cycles d'horloge par instruction (CPI). Le

[2]La cohérence des données est maintenue par le matériel
[3]L'invalidation d'une ligne n'est pas communiquée au répertoire

nombre de cycles nécessaire à l'exécution d'une instruction dépend de deux facteurs :

- Le nombre de cycles d'horloge minimum nécessaire à l'exécution d'une instruction.
- Le temps en cycles d'horloge pendant lequel le processeur aura été bloqué à cause d'un échec d'accès aux données ou aux instructions.

Le premier facteur est connu et déterminé par l'implantation matérielle du jeu d'instructions. Nous appellerons $CPIopt$ la valeur obtenue en ne prenant en compte que ce premier facteur (tous les accès à la mémoire sont réalisés en un temps nul).

Le deuxième facteur est déterminé par l'architecture matérielle (pénalité d'un échec de cache, parallélisme d'accès,...) et à l'ordre des accès qui déterminent le taux d'échec en cache. Ces facteurs varient également à cause des interactions et effets de bords (congestion sur le réseau, invalidation de données, dépendances des données, etc...).

Néanmoins, si nous considérons des processeurs simples (exécution ordonnée, caches bloquants en lecture, un seul fil d'exécution), une bonne approximation de cette valeur est définie par le taux d'échec (noté Te) d'accès au cache et de sa pénalité (notée Pe). De plus, dans une architecture avec des accès uniformes à la mémoire, nous pouvons considérer que s'il n'y a pas de congestion d'accès (nous considérons un système monoprocesseur) la pénalité d'échec est constante.

$$CPI \approx CPIopt + Te.Pe \tag{8.1}$$

Le placement des données n'a d'influence que sur la pénalité d'échec qu'il convient de réduire au minimum.

Dans la suite de ce chapitre, nous utiliserons le symbole «˜» pour indiquer les valeurs moyennes. Dans un système à N processeurs, la valeur moyenne des CPI est :

$$\widetilde{CPI} = \frac{\sum_{i=0}^{i<N} CPI_i}{N} \tag{8.2}$$

Pour améliorer les performances du système il faut minimiser le CPI des processeurs qui exécutent du code utile, certains processeurs peuvent en effet exécuter des routines d'attentes. Toutefois, minimiser le CPI moyen en prenant en compte tous les processeurs est une solution plus simple que nous adoptons. Nous considérons par ailleurs que le CPI optimal et le taux d'échec sont le même pour tous les processeurs.

$$\widetilde{CPI} \approx \frac{\sum_{i=0}^{i<N} (CPIopt + Te.Pe_i)}{N} \tag{8.3}$$

$$\Leftrightarrow \widetilde{CPI} \approx CPIopt + Te.\frac{\sum_{i=0}^{i<N} Pe_i}{N} \tag{8.4}$$

Dans l'équation 8.4 nous déterminons le CPI moyen en fonction de la pénalité d'échec moyenne. Nos travaux visent à minimiser ce dernier paramètre en plaçant et déplaçant au mieux les données sur l'ensemble de la puce. Notons que la pénalité d'échec est approximativement constante tant qu'il n'existe pas de congestion sur les liens de communication ou les bancs mémoire. Nous considérons un environnement multiprocesseur à mémoire partagée distribué sur M bancs mémoire sur lequel de la congestion peut exister. Il convient donc de préciser Pe :

$$Pe_i = \sum_{j=0}^{j<M} A_{ij} \widetilde{L}_{ij} \tag{8.5}$$

Où A_{ij} est le nombre d'accès entre le processeur i et le banc mémoire j, $\widetilde{L}_{ij}$ est la latence moyenne en nombre de cycles des accès sur le lien (i, j).

8.2.1 Coût d'accès à une donnée

Dans l'équation précédente 8.4, nous avons exprimé le CPI moyen en fonction de la pénalité d'échec subie par un processeur. Minimiser cette pénalité d'échec n'est pas une chose simple. Si l'on se contente de rapprocher les données accédées nous améliorerons très certainement la pénalité d'échec du processeur concerné, mais en contre-partie, nous pénaliserons tous les processeurs qui partagent la même donnée.

Ce genre de politique a déjà été appliquée avec un succès mitigé, elle s'appelle le placement « *first-touch* ». Le premier processeur (cache) qui réalise l'accès récupère la donnée.

Si l'on prend le problème dans l'autre sens, nous allons mesurer le coût d'accès à une donnée D, placée sur un banc mémoire j, par N processeurs au lieu du coût d'accès perçu par un processeur à un ensemble de données.

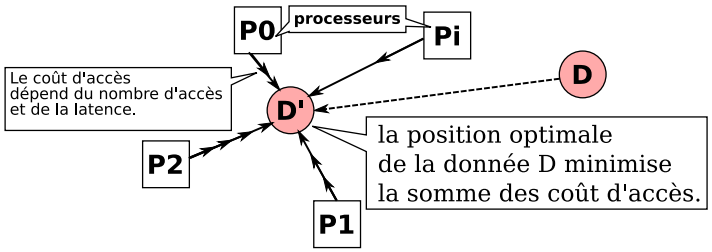


FIG. 8.2 – Placement optimal de la donnée D qui minimise le coût d'accès par les processeurs P_i.

La figure 10.1 montre cette vision du coût d'accès centré sur la donnée. Soit une donnée D placée sur un banc mémoire j et $(P_i)_{i=0..N-1}$ les processeurs du système. Nous pouvons définir ainsi le coût d'accès à cette donnée :

$$C_j = \sum_{i=0}^{i<N} A_{ij}\widetilde{L}_{ij} \tag{8.6}$$

Ce coût est aisé à calculer par un nœud mémoire car il possède (potentiellement) toutes les informations nécessaires (cf. section 9.2.3). Le point qui minimise ce coût n'est pas le barycentre des processeurs pondérés par leur quantité d'accès. Ce coût n'étant pas une distance, ce n'est pas non plus un point de Fermat.

8.2.2 Minimisation du $\widetilde{CPI}$ à l'aide d'un placement optimal des données

Si nous reprenons les équations 8.4 et 8.5 nous avons :

$$\widetilde{CPI} \approx CPIopt + Te.\frac{\sum_{i=0}^{i<N}\sum_{j=0}^{j<M} A_{ij}\widetilde{L}_{ij}}{N} \tag{8.7}$$

$$\Leftrightarrow\widetilde{CPI} \approx CPIopt + Te.\frac{\sum_{j=0}^{j<M}\sum_{i=0}^{i<N} A_{ij}\widetilde{L}_{ij}}{N} \tag{8.8}$$

$$\Leftrightarrow\widetilde{CPI} \approx CPIopt + Te.\frac{\sum_{j=0}^{j<M} C_j}{N} \tag{8.9}$$

La placement optimal de toutes les données d'un système permet donc bien de minimiser le CPI moyen de l'ensemble des processeurs.

8.3 Congestion d'accès : heuristique

Le placement optimal des données présenté précédemment fait l'hypothèse que la latence entre deux points d'un réseau est variable et dépend de la disponibilité du lien. Si, afin de simplifier l'implantation, on considère que la latence n'est fonction que de la topologie du réseau, il se peut alors que des points de congestion se créent. En effet, si un grand nombre de données fréquemment accédées sont placées au même endroit afin d'en minimiser le coût d'accès, la latence considérée va rester constante alors que le contrôleur mémoire sera saturé. La latence réelle va augmenter de façon exponentielle, mais comme elle n'est pas prise en compte, le choix de placement restera inchangé.

8.3.1 Causes de la pathologie

Le point de congestion peut apparaître à cause d'une multitude de facteurs. Nous avons en premier lieu une mauvaise conception d'une application parallèle où certaines structures sont globales et massivement partagées par toutes les tâches. Toutefois, d'autres causes ne sont pas l'œuvre d'un programmeur mais d'un concours de circonstances. Un ensemble de données non partagées ont pu par exemple être allouées dans un même banc mémoire, créant ainsi un point de congestion.

Déplacer dynamiquement les données peut également créer des points de congestion si statistiquement toutes les données ont tendance à se regrouper au même endroit. Un banc mémoire qui devient un point de congestion va voir sa latence d'accès croître de façon exponentielle.

Comme nous le verrons par la suite, la solution que nous avons adopté ne prend pas en compte la latence réelle des accès. Nous appliquons donc une heuristique afin d'éviter ce problème.

8.3.2 Solution mise en place

La solution est implantée à l'aide de deux mécanismes :

1. Si sur une période de temps ΔT, le nombre d'accès au banc mémoire j dépasse un seuil S_{Mc}, alors il convient de déplacer la donnée la plus accédée vers un autre nœud.
2. Le choix de placement d'une donnée D d'un banc mémoire k ayant observé un coût d'accès $C_{k|D} = \sum_{i=0}^{i<N} A_{ik}^{D}$ accès, doit exclure les bancs mémoires j tels que :

$$C_{k|D} + C_j > S_{Mc}$$

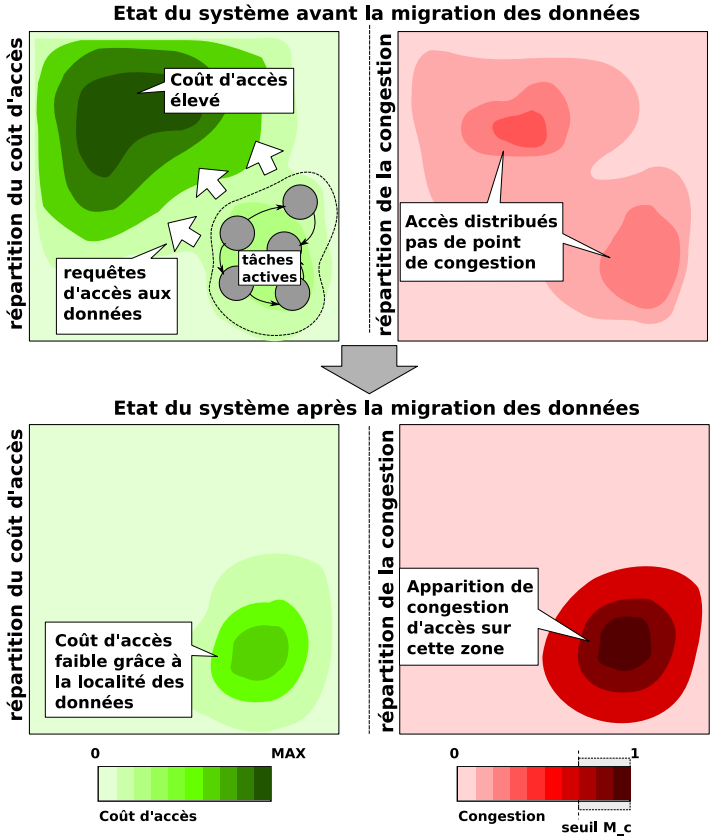


FIG. 8.3 – Exemple où la migration des données crée un point de congestion.

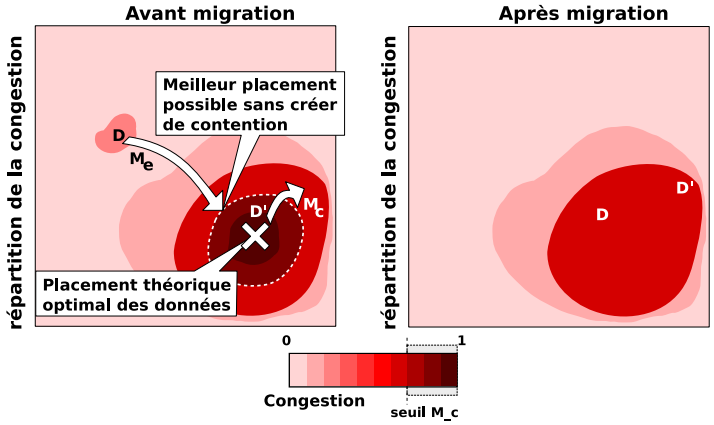


FIG. 8.4 – Application de la solution pour éviter la congestion.

Où $C_{k|D}$ est la partie du coût d'accès de k imputée à la seule donnée D et A_{ik}^{D} est le nombre d'accès du processeur i à la donnée D se trouvant sur le banc mémoire k. Autrement dit, la donnée ne doit pas être placée à un endroit qui risque de devenir un point de congestion.

L'application de ces deux mécanismes limite l'apparition et favorise la disparition des points de congestion. Dans la figure 8.4 nous observons l'application de ces deux méca-

nismes sur deux données D et D'. La donnée D est placée au mieux à la périphérie de la zone sous congestion. La donnée D' est déplacée afin de faire diminuer la pression sur cette zone.

8.3.3 Avantages

Cette solution a plusieurs avantages. Le premier avantage est d'éviter en amont la formation de points de congestion. Ce système, permet également de détecter très rapidement la formation d'un point de congestion et de le traiter dans les plus brefs délais.

Chapitre 9

Solution et implantation proposée pour la migration dynamique des données

Nous avons vu dans le chapitre précédent comment placer de façon optimale des données dans un système. Cette solution théorique n'est pas simple à implanter. Dans ce chapitre nous présentons une implantation de cette solution. Les problèmes à résoudre sont nombreux : garantir l'adressage de toutes les données, garantir la cohérence des adresses, réduire l'impact du transfert des données, etc...

9.1 Déplacer les données dans le système : choix réalisés

Notre but premier est d'offrir une solution dont l'usage ne soit pas une contrainte ni une charge pour le programmeur. En ce sens, nous devons proposer une solution totalement indépendante de l'application. Ceci implique qu'elle sera implantée soit dans le système d'exploitation (et invisible au programmeur), soit dans l'architecture matérielle, ou bien, dans les deux à la fois.

Nous avons précédemment montré que les contraintes de réactivité, d'agnosticismes, et d'efficacité nous conduisent à proposer une solution gérée par le matériel avec un support éventuel du logiciel. Ce dernier point n'a pas été exploré dans le cadre de nos travaux mais, en marge de ce chapitre nous proposerons quelques pistes de recherche complémentaires qui peuvent y avoir trait.

Pour réaliser un transfert continu et efficace des données dans le système, il est nécessaire de satisfaire les contraintes suivantes :

1. La charge sur le lien de communication induite par les transferts de données ne doit pas perturber de façon significative les accès aux données par les différents composants du système.
2. Le placement des données doit être réalisé en fonction de la topologie et de la fréquence des accès. Dans le cas idéal, le placement des données minimisera les coûts d'accès de l'ensemble du système.
3. Le coût de la migration des données devra être minimisé afin de maximiser l'efficacité de la migration.
4. Un meilleur placement des données se traduit par une baisse de la consommation du système liée à la distance moindre que parcourent les données sur la puce. Ce gain en

énergie devra être inférieur au coût lié à la migration des données.

9.1.1 Granularité des déplacements

Les mécanismes et techniques mis en œuvre pour garantir l'adressage de toutes les données, et calculer le coût d'accès aux données, ont un coût d'implantation en matériel important.

Par conséquent, afin de miniser ce coût nous avons été obligés de choisir une unité de migration à gros grain. Nous avons choisi de migrer les données par pages de 4 kilo-octets, taille qui implique un coût d'implantation raisonnable.

9.1.2 Choix d'une solution pour le transfert des données

Puisque l'unité de déplacement a la taille d'une page, la migration des données a une influence non négligeable sur la disponibilité du réseau. De plus, il est impératif de pouvoir déplacer rapidement des données d'un banc mémoire qui subit une congestion d'accès.

Pour cela, nous avons deux solutions : implanter des canaux virtuels prioritaires dans le médium de communication, ou bien, utiliser un deuxième réseau d'interconnexion.

Nous avons privilégié cette deuxième solution qui d'une part, était plus simple à réaliser, et d'autre part permet de mieux prendre en compte les spécificités des besoins. Le transfert des données lors d'une migration de pages n'a pas besoin d'être aussi efficace que l'accès à une donnée à cause d'un échec de cache. En effet, le nombre de migrations sera très inférieur au nombre d'accès aux données réalisés par les processeurs. De plus, nous n'autorisons qu'une seule migration à la fois pour simplifier les protocoles d'échange.

Finalement, afin de limiter le surcoût matériel lié à l'implantation d'un deuxième réseau d'interconnexion, nous avons choisi un lien en anneau qui relie tous les contrôleurs mémoire.

9.2 Implantation de la solution proposée

9.2.1 Architecture globale

Comme nous l'avons précisé dans l'introduction, nous utilisons comme base une architecture homogène à mémoire partagée (cf fig. 9.1).

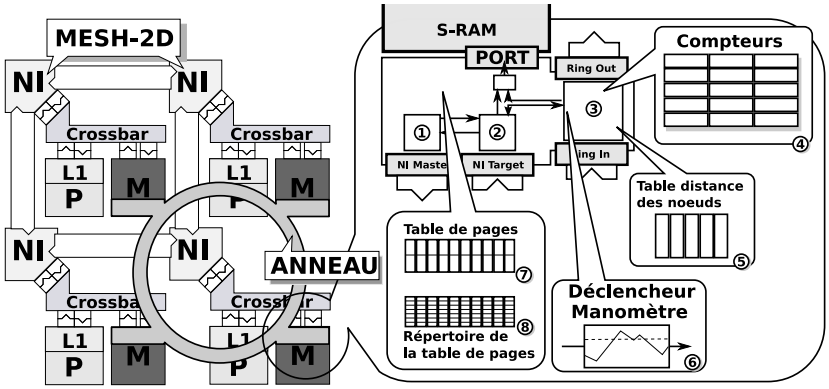


FIG. 9.1 – Architecture homogène à mémoire partagée et distribuée.

Afin de pouvoir transférer les pages entre les différents bancs mémoire, nous avons implanté un canal de communication spécifique et dédié à cette tâche. Ce canal est constitué d'un anneau de communication superposé à l'architecture. Réaliser l'intégration physique d'un tel anneau ne pose pas de problème avec les technologies actuelles [BZ07a, BZ07b].

Le canal de communication dédié est nécessaire pour deux raisons : éviter des interblocages et permettre une communication efficace même lorsque l'interface mémoire est saturée. Le réseau en anneau pourrait être substitué par des canaux virtuels prioritaires dans le réseau de base. La topologie en anneau a été choisie dans le but de minimiser le surcoût d'implantation du réseau dédié. Nonobstant, la solution présentée s'appliquerait à toute autre topologie ou implantation jugée plus pertinente en fonction du contexte d'application.

Les différents modules du contrôleur mémoire sont :

① : Module d'envoi des requêtes d'invalidation (cache - TLB).
② : Module de traitement des requêtes de lecture, écriture et échec de TLB.
③ : Module de migration des pages (déclenchement, contrôle, transfert).

Les objectifs d'une telle architecture sont multiples

1. Elle doit converger vers un placement optimal des données sur la puce.
2. Elle doit être réactive à toute évolution du système : variation de charge, migration des tâches.
3. Elle ne doit pas induire de surcoût significatif lorsque les données n'ont pas besoin d'être déplacées.
4. Elle doit avoir un coût d'implantation acceptable.

Les défis à relever sont les suivants :

– Être en mesure d'adresser à tout instant toutes les données du système.
– Déclencher la migration d'une page au moment approprié et sur des critères pertinents.
– Transférer efficacement les données.
– Garantir la cohérence de l'ensemble de l'espace d'adressage.

9.2.2 Adressage des données

Lors de la migration d'une page, il faut garantir que les données sont toujours correctement adressables. Pour cela, nous pouvons, soit avoir une reconfiguration dynamique de l'espace d'adressage via les routeurs du réseau, soit avoir une traduction d'adresse au niveau de la source. La première solution n'est pas envisageable car pour permettre une grande flexibilité au niveau du placement des données, les tables de routage auraient occupé une surface au coût excessif.

Nous avons donc implanté une traduction d'adresse physique en adresse matérielle épaulée par un tampon de traduction (*Translation Lookaside Buffer*,TLB). La solution proposée est présentée dans la figure 9.2. Nous appelons *adresses physiques* les adresses issues du processeur, et *adresses matérielles* les adresses émises sur le réseau[1].

Lors d'un échec de traduction dans la TLB ❶, une requête spécifique est envoyée au nœud mémoire adressé, afin de récupérer une traduction d'adresse pour la page à laquelle on accède. Pour un banc mémoire, l'ensemble des traductions de sa plage d'adressage est maintenu dans une table de pages matérielle ❷. Cette table de pages contient 2 entrées adressées par le numéro de page accédé. La première entrée `@dest` est l'adresse matérielle où se

[1]Cette terminologie permet d'ajouter la notion d'adresse logique si les processeurs la supportent

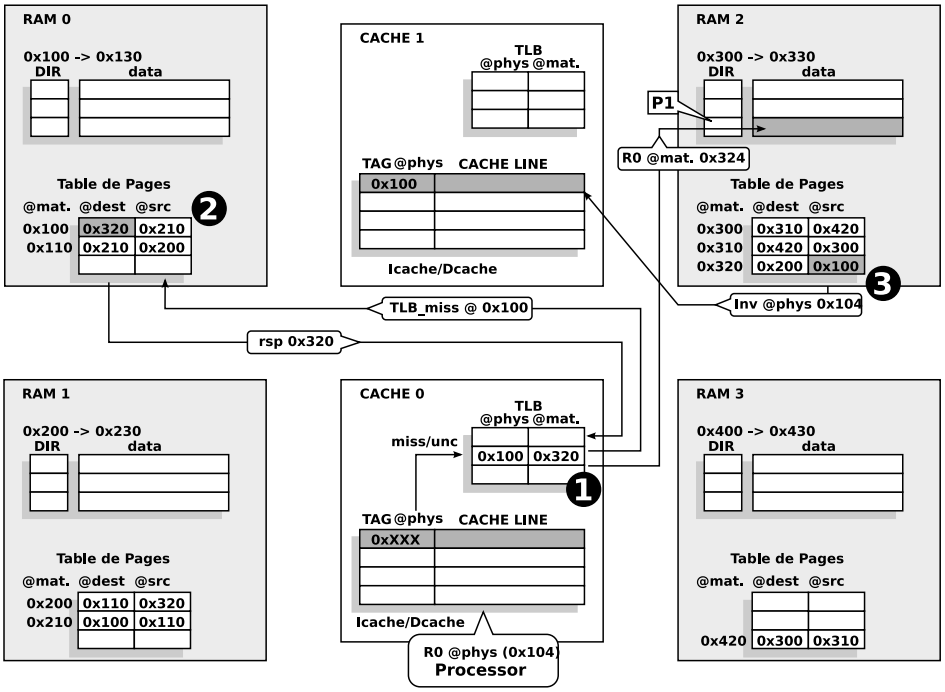


FIG. 9.2 – Mécanisme d'adressage des données.

trouve actuellement la page recherchée. La deuxième entrée `@src` indique l'adresse d'origine de la page actuelle présente dans cet emplacement. Cette deuxième entrée est, entre autre, utilisée pour l'invalidation des données des caches ❸.

Les deux espaces d'adressage (physique et matériel) ont la même taille. Lorsque l'on déplace une donnée, elle est échangée avec une autre donnée dans le système. Par conséquent, à tout instant, il existe une et une seule donnée par adresse. Si l'on modélise la translation d'adresses à l'aide d'une application au sens mathématique, celle-ci est une bijection entre l'espace d'adressage physique et l'espace d'adressage matériel.

L'implantation d'un adressage virtuel (qui est orthogonal et indépendant à notre solution) est plus coûteux et complexe pour deux raisons : l'espace d'adressage virtuel est plus grand que l'espace d'adressage de la mémoire physique [2] ; une même adresse physique peut correspondre à plusieurs adresses virtuelles.

Nous énumérons ci-dessous les avantages d'une translation d'adresse physique/matérielle.

1. La table de pages est de taille réduite, elle est donc réalisable en matériel.
2. Les caches ne sont pas invalidés lors de la migration d'une page car ce sont des caches d'adresses physiques et non matérielles.
3. Les synonymes de traduction dans le cache ne sont pas possibles, deux adresses physiques ne peuvent correspondre à la même adresse matérielle.
4. La TLB étant placée après le cache, le chemin critique d'accès aux données n'est pas allongé, l'accès à la TLB ne se fait que sur un échec de donnée en cache.
5. Une petite TLB (8 entrées) est suffisante car le nombre de traductions physiques/matérielles est relativement restreint.

[2] D'un point de vue du nombre de bits nécessaires à adresser la capacité de stockage fournie

6. La pénalité d'échec de TLB est équivalente à une pénalité d'échec en cache.

9.2.3 Déclenchement de la migration

Pour déclencher efficacement la migration d'une page, nous avons besoin de connaître à tout instant quelles sont les pages les plus accédées, et pour chaque page, quels sont les nœuds qui y font le plus souvent référence. Nous avons donc implanté comme dans les machines SGI Origin [LL97] des compteurs par page (fig. 9.3 et 9.1 ④), et pour chaque page un compteur par nœud. À l'aide d'un comparateur, nous conservons l'indice de la page la plus accédée. Ces compteurs sont mis à zéro dès qu'une migration de pages a été effectuée, ou bien, dès que la période de remise à zéro a été atteinte. Cette remise à zéro périodique permet d'éviter la prise en compte d'une vision périmée de l'état du système. La période a été fixée arbitrairement à 100000 cycles d'horloges.

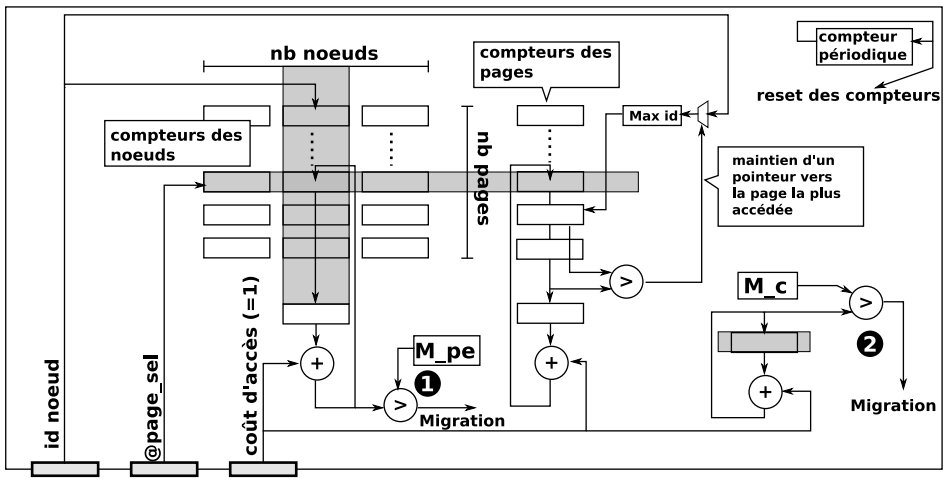


FIG. 9.3 – Module de comptage des accès par nœud et par page

Comme nous l'avons présenté dans le chapitre précédent, nous agissons sur deux éléments : la latence moyenne des accès et les points de congestion.

9.2.3.1 Éliminer la congestion

Si de nombreux nœuds accèdent à des données se trouvant dans le même banc mémoire (données partagées, privées, pile, code) ; les latences d'accès à ces données augmentent de façon exponentielle au-delà d'un certain débit. Pour éviter cela, nous avons un module (figures 9.3 ❷ et 9.1 ⑥) qui compte les requêtes arrivées sur le banc mémoire. Si le nombre de requêtes sur un intervalle de temps (pression) dépasse un certain seuil (S_{M_c}), une migration M_c est déclenchée (❷). La page déplacée est celle qui est le plus souvent accédée (❸).

9.2.3.2 Diminuer la latence globale

Le but principal du déplacement des données est de diminuer la latence moyenne des accès dans leur globalité. Pour cela, nous avons dans chaque banc mémoire un mécanisme (cf. figure 9.3 ❶) qui périodiquement va déclencher la migration de la page (probablement) la plus accédée. Dans notre architecture, cette période est fixée à 128 requêtes mémoire (par un processeur sur une page), seuil au-delà duquel la migration d'une page devient rentable. Nous notons ce type de migration M_{pe}.

Dans l'ensemble, les migrations M_c vont contribuer à distribuer les données afin d'éliminer la congestion d'accès sur certains bancs mémoire. En revanche, les migrations M_{pe} vont contribuer à mieux placer les données quel que soit l'état du système. Ces migrations permettent également de rapatrier des données initialement bien placées mais qui avaient été éloignées à cause d'une congestion sévère sur le banc mémoire.

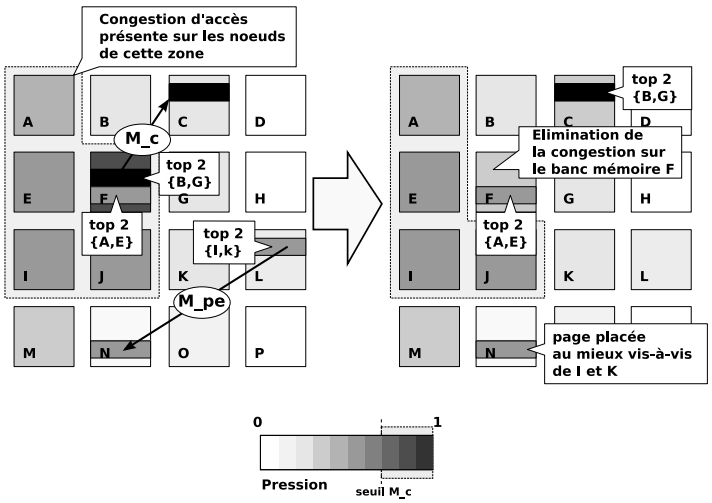


FIG. 9.4 – Exemple de migration de données pour diminuer la congestion (M_c), et rapprocher les données (M_{pe}).

9.2.4 Placement des données et stabilité du système

Pour placer les données, nous rappelons les deux critères décrits précédemment (cf. section 8.3.1) :

- Le nœud sélectionné ne doit pas subir une pression telle que, ajoutée a la pression subie sur la page le seuil de congestion soit atteint.
- Le nœud sélectionné devra minimiser la somme des coût d'accès de chacun des processeurs. Le coût d'accès est le produit du nombre d'accès à la page par la distance de Manhattan entre le banc mémoire et le processeur. Pour réduire le coût d'implantation de ce calcul, le nombre d'accès par un nœud à une page est arrondi à la puissance de deux inférieure.

Lorsqu'une page a été sélectionnée pour être déplacée, le contrôleur fait un appel d'offre sur l'anneau. Chaque nœud calcule le coût d'accès vis-à-vis de la pression exercée par chaque processeur. Chaque nœud connaît sa distance vis-à-vis des processeurs grâce à une table (figures 9.5 et 9.1 ⑤).

Le nœud initiateur de la requête choisi comme destinataire le nœud ayant le plus petit coût d'accès. Lors du calcul du coût, si la somme de la pression subie par le nœud et de la pression subie par la page à migrer dépasse le seuil de congestion, alors le nœud informe d'un coût maximal afin de ne pas être sélectionné.

S'il n'existe pas de banc mémoire offrant un meilleur placement pour la page, la migration est annulée. De plus, on remet à zéro les compteurs de cette page afin de favoriser l'évincement d'une autre page.

La migration de pages se stabilise lorsque toutes les pages sont placées au mieux (les migrations M_{pe} échouent), ou bien lorsque l'ensemble du système est sous pression constante

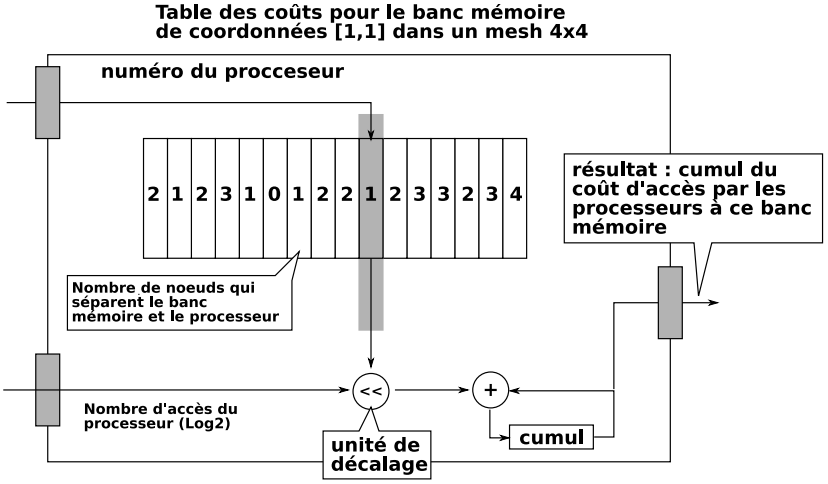


FIG. 9.5 – Module permettant de calculer le coût d'accès d'une page vis-à-vis de tous les processeurs du système

(échec des migrations M_c).

Dans la figure 9.6 sont présentées les deux types de migrations. Le banc mémoire du nœud `F` est accédé par quatre processeurs se trouvant dans les nœuds `A`, `B`, `E` et `G`. Néanmoins, ces accès ciblent spécifiquement deux pages. La page la plus accédée de ce banc mémoire l'est en priorité par les nœuds `B et G`. Ces derniers peuvent exécuter des tâches concurrentes travaillant sur un même ensemble de données, ou bien exécuter deux tâches indépendantes dont le hasard des allocations mémoire génère beaucoup d'accès à cette page. Dans ce cas, le système de migration va détecter un début de congestion (pression supérieure à un seuil) et déclencher la migration (M_c) de la page la plus accédée. Le nœud `C` est le mieux placé pour recevoir une page et ainsi permettre au nœud `F` de subir moins de pression et servir efficacement les requêtes sur les autres pages. Par ailleurs, le nœud `L` bien que ne subissant pas de pression particulière, va essayer de trouver un meilleur emplacement pour sa page la plus accédée. Celle-ci serait placée au mieux sur le nœud `J`. Néanmoins, puisque le nœud `J` subit une pression élevée, il n'est pas sélectionné et `N` sera choisi. Cette migration (M_{pe}) va contribuer à mieux distribuer les données tout en évitant de créer de la congestion comme par exemple, au centre de la puce dont les bancs mémoire sont statistiquement mieux placés vis-à-vis du plus grand nombre de processeurs.

Dans l'implantation actuelle, une seule migration peut avoir lieu à la fois dans le système.

9.2.5 Transfert des données

Dès qu'un nœud a été sélectionné pour recevoir une page, il va choisir aléatoirement une page à évincer. Le choix aléatoire est simple à implanter et il offre de très bons résultats car les références sont en général concentrées sur un très faible nombre de pages.

Les données des deux pages sont échangées par paquets de 32 bits, soit la largeur de l'anneau. Le répertoire associé à la page est également transféré afin de garantir la cohérence des données. La migration d'une page de 4 kilo-octets est réalisée en à peu près 2000 cycles d'horloge sur une architecture à 16 nœuds. Ceci inclut également le temps nécessaire au choix du nœud. Afin de garantir la cohérence des adresses accédées, les pages en cours de transfert sont marquées comme étant « contaminées » . Cette technique a été inspirée des

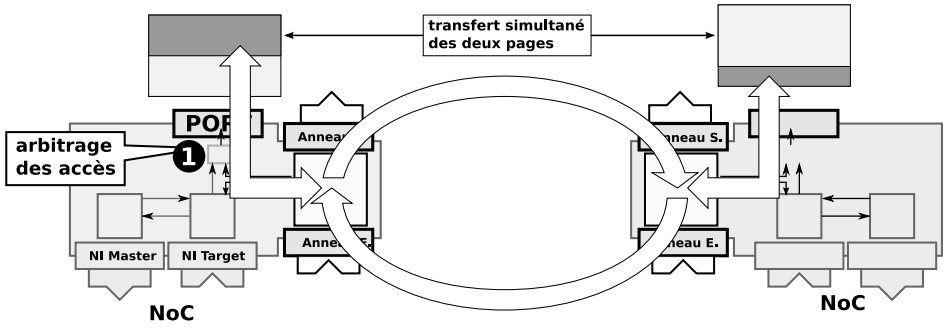


FIG. 9.6 – Transfert de deux pages entre deux nœuds

machines SGI Origin. Tout accès à une page contaminée est acquitté négativement (NACK), le cache devra ré-émettre la requête ultérieurement. Les accès aux pages qui ne migrent pas sont traitées normalement. Néanmoins, ces accès sont perturbés par la migration car l'accès à la mémoire est partagé (cf. ❶ figure 9.6).

Toutefois, le coût induit par la migration des données va être compensé par des accès en moyenne plus rapides.

9.2.6 Réactivité du système et surcoût lié à la migration

Le problème majeur des solutions logicielles est leur faible réactivité et l'utilisation des processeurs pour des tâches du système à la place de tâches de calcul. En effet, la migration des données implique une prise de décision au niveau logiciel à l'aide d'un traitant d'interruption ou d'une tâche périodique. De plus, si le mécanisme est basé sur l'utilisation de la mémoire virtuelle les caches seront très certainement invalidés. Finalement, l'invalidation de toutes les traductions d'adresses peut prendre un temps considérable.

Ce manque de réactivité a un défaut : la migration est potentiellement en retard sur le comportement de l'application. Dans notre architecture, le temps nécessaire au déclenchement d'une migration est très inférieur à la tranche de temps système allouée à une tâche. De plus, il n'y a aucun effet de bord (invalidation des caches, blocage des bancs mémoire). Toutes les requêtes du systèmes continuent d'être traitées normalement. Seules les requêtes concernant les pages en cours de migration subissent un surcoût de latence.

9.2.7 Évaluation approximative du surcoût en surface de l'implantation

Le coût d'implantation de notre solution correspond au coût d'implantation des différents éléments ajoutés :

1. Réseau d'interconnexion supplémentaire.
2. Compteurs par nœud et par page.
3. Table des pages.
4. Cache de traduction des adresses (TLB).
5. Répertoire de la table des pages.
6. Table de calcul des distances (une par banc mémoire).
7. Logique de contrôle et unités de calcul nécessaires (additionneurs et unités de décalage).

Nous ne sommes pas en mesure de fournir un coût en surface d'implantation de notre solution car nous n'avons pas réalisé d'implantation synthétisable.

Néanmoins, nous pouvons évaluer le coût d'implantation des différents éléments de mémorisation. Dans le tableau 9.2.7, nous donnons une évaluation de ce coût pour une plateforme à 128 processeurs et des pages de 4 kilo-octets.

TAB. 9.1 – Surcoût en quantité d'information des éléments mémorisants de la solution implantée.

Élément	coût/page	ratio (%)
Compteurs	$128 \times 7 + 17$ bits	$2,8$ %
Entrée dans la T.P*.	$2 \times 16 + 128$ bits	$0,49$ %
Total	-	$3,9$ %

* Table des Pages, entrée dans le répertoire incluse.

Le coût des compteurs est de 4% pour une architecture à 128 nœuds, et de seulement 0,5% pour une architecture 16 processeurs. Notons que ce coût est proportionnel au nombre de processeurs et inversement proportionnel à la taille des pages utilisées. Ainsi, avec des pages de très petite taille (512 octets), nous avons pour 16 processeurs un coût de 8,34% et pour 128 processeurs un coût de 53%.

Par conséquent, le coût d'implantation des compteurs peut être négligeable ou prohibitif en fonction de l'architecture.

Le coût des autres éléments dépend du nombre de processeurs et de bancs mémoire. Mis à part l'anneau d'interconnexion, le coût le plus important est celui des tables de cacul des distances. Celui-ci est proportionnel au nombre de processeurs et ne dépasse pas 1 kilo-octet pour 128 processeurs (128×6 bits).

9.2.8 Cohérence des traductions

9.2.8.1 Problème

La difficulté majeure de la migration des pages est de garantir à moindre coût la cohérence des TLB. Il faut que toute requête de lecture ou d'écriture atteignant un banc mémoire s'adresse effectivement à la page contenue.

Dans notre système, nous n'avons aucune garantie sur les temps d'accès aux différents bancs mémoire. Par conséquent, invalider naïvement toutes les TLB du système lors de la migration d'une page n'est pas suffisant, une requête pourrait avoir été émise avant la migration d'une page et atteindre le banc mémoire après qu'elle ait migré.

9.2.8.2 Solution

Nous supposons que notre réseau d'interconnexion possède les caractéristiques suivantes :

H1 : Tous les accès entre un émetteur et un récepteur du réseau sont ordonnés.

H2 : Le temps d'accès entre deux points du réseau est fini mais non borné.

H3 : Les accès entre un émetteur et plusieurs récepteurs ne supposent aucune relation d'ordre.

Ces hypothèses sont vraies pour la plupart des réseaux point à point que l'on peut trouver dans un système embarqué.

Le protocole de migration doit :

– S'assurer de pouvoir identifier les requêtes valides d'après la nouvelle translation d'adresses.
– S'assurer de la mise à jour des TLB's.
– Minimiser le temps d'attente d'un nœud vis-à-vis d'une donnée en cours de transfert.

Nous présentons ci-dessous un protocole de migration qui garantit la cohérence des TLB, chacune des étapes est représentée dans la figure 9.7.

Ci-dessous, les étapes du protocole et les abréviations associées.

E1 : Décider de migrer une page : Mig_dec

E2 : Sélectionner un nœud avec qui échanger la page : Mig_sel

E3 : Contaminer les deux pages qui vont être échangées : $cont\_rep$

E4 : Mettre à jour les tables de pages (jusqu'à quatre nœuds concernés) : maj_tpages

E5 : Envoyer les invalidations vers les TLB concernant ces deux pages : TLB_inv

E6 : Effectuer la migration des données (en parallèle du point précédent) : tr_page

E7 : Attendre l'acquittement d'invalidation de TLB émis par les contrôleurs de cache : TLB_ack

E8 : Décontaminer les pages concernées : $decont\_rep$

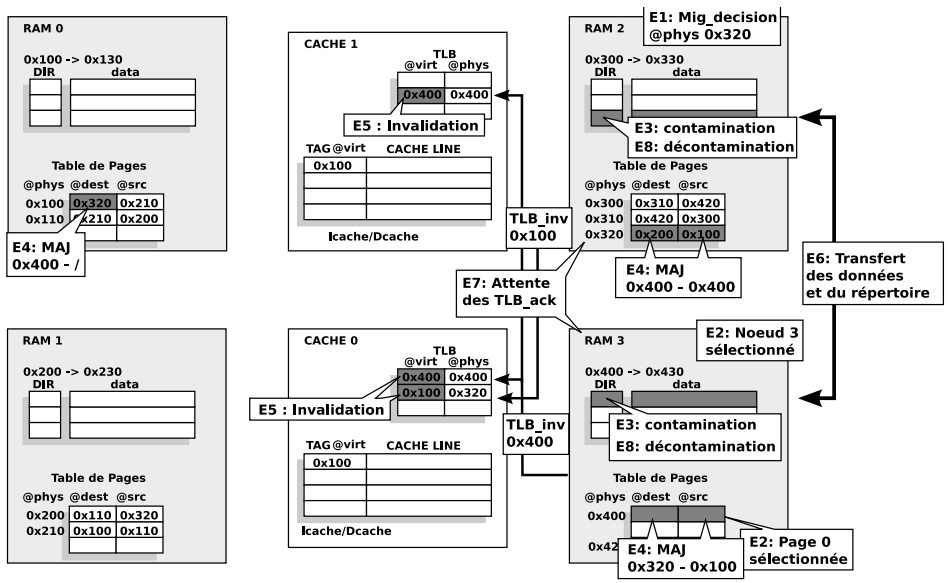


FIG. 9.7 – Étapes de la migration de la page d'adresse matérielle 0x320

Pour que ce protocole garantisse effectivement la cohérence des TLB, il faut qu'au niveau du contrôleur de cache les contraintes suivantes soient respectées :

C1 : Une invalidation de TLB doit entraîner la suppression de la traduction d'adresse de la page concernée. Puis, une requête d'acquittement sera émise sur l'interface maître du contrôleur de cache.

C2 : L'acquittement à l'invalidation de TLB doit être envoyé après la réception de la réponse à toute requête en cours.

Et au niveau du contrôleur mémoire :

C3 : Toute requête concernant une page qui est contaminée est acquittée négativement (NACK). La requête n'est pas prise en compte par le contrôleur mémoire dans le cas d'une écriture. Ceci concerne également les échecs de TLB.

Nous contaminons les pages échangées pendant leur migration afin d'être sûr d'y accéder dans un état cohérent. Ce système de contamination est inspiré du protocole de migration mis en œuvre dans les machines SGI origin [LL97].

9.2.8.3 Preuve de cohérence

Ci-dessous nous utiliserons les abréviations suivantes : caches « C » , mémoire « M » , envoi « env » , réception « rec » , échec de TLB « TLB_miss » , succès de TLB « TLB_hit » , requête (LD/ST) « req » , réponse « rsp » .

Sur le protocole susmentionné, nous pouvons établir les relations d'ordre suivantes dans lesquelles $x < y$ implique que l'évènement x a eut lieu avant y, et x, y que x et y ont lieu sans relation d'ordre établie (potentiellement en parallèle). Notons pouvons écrire $w < x, y < z$ ce qui implique entre autres que $w < z$, $w < y$ et $x < z$. La relation $x = y, z$ indique que x est la réalisation de y et z. Le premier des évènements est le signal $Reset$ du système.

R1 : $Reset < C_{env\_TLB\_miss} < C_{TLB\_hit} < C_{env\_req} < M_{recep\_req} < C_{recep\_rsp}$

R2 : $Reset < M_{Debut\_mig} < C_{recep\_TLB\_inv} < M_{Fin\_mig}$

R3 : $M_{Debut\_mig} < M_{tr\_page}, M_{recep\_TLB\_ack} < M_{Fin\_mig}$

R4 : $M_{Debut\_mig} = M_{Mig\_dec}$, M_{cont_rep}, M_{maj_tpages}

R5 : $M_{Fin\_mig} = M_{decont\_rep}$, $M_{env\_TLB\_inv}$

Dans la figure 9.7, sont présentées les différentes actions possibles. L'interaction entre les caches et les mémoires est représentée par tous les ordonnancements possibles des évènements constituant R1 et R2. Nous pouvons diviser en quatre l'ensemble des ordonnancements possibles des évènements de R1 et R2. L'idée générale de la preuve va être de montrer qu'en toute circonstance, quel que soit l'ordonnancement dans le temps des évènements, la donnée accédée par un processeur correspond toujours à une traduction d'adresse valide.

1. $C_{TLB\_hit} < C_{recep\_TLB\_inv}$ et $M_{Debut\_mig} < M_{recep\_req}$: Cet espace correspond aux cas où une traduction a été réalisée avec l'ancienne translation et que la prise en compte par la mémoire de la requête se fait potentiellement sur la nouvelle page. D'après R4, la requête arrive après la contamination du répertoire.

   De plus, $M_{recep\_TLB\_ack} < M_{Fin\_mig}$ et $C_{env\_TLB\_ack} < M_{recep\_TLB\_ack} < M_{decont\_rep}$ (cf C1, H1, R5). Or, d'après C2 et R1 $C_{recep\_rsp} < C_{env\_TLB\_ack}$.

   Donc, nécessairement, $M_{recep\_req} < M_{decont\_rep}$. D'après C3, La requête n'est pas prise en compte par le banc mémoire, un NACK est envoyé et la requête sera émise à nouveau avec une traduction d'adresse correcte.

2. $C_{TLB\_hit} < C_{recep\_TLB\_inv}$ et $M_{recep\_req} < M_{Debut\_mig}$: Dans ce cas, la TLB fournit l'ancienne traduction d'adresse et le contrôleur mémoire possède la page correspondante. Il prend en compte la requête et répond avec une donnée valide pour une lecture, ou bien, met à jour la donnée dans le cas d'une écriture.

3. $C_{recep\_TLB\_inv} < C_{TLB\_hit}$ et $M_{recep\_req} < M_{Debut\_mig}$: D'après R1 $C_{TLB\_hit} < M_{recep\_req}$, on aurait donc $C_{recep\_TLB\_inv} < C_{TLB\_hit} < M_{recep\_req} < M_{Debut\_mig}$, ce qui est impossible d'après R2. Cette situation ne se produira jamais.
4. $C_{recep\_TLB\_inv} < C_{TLB\_hit}$ et $M_{Debut\_mig} < M_{recep\_req}$: La traduction d'adresse correspond à la nouvelle correspondance de page. Deux cas de figures peuvent se produire :
   - $M_{recep\_req} < M_{Fin\_mig}$: La requête est perçue par le contrôleur mémoire alors que le répertoire est contaminé. Un NACK est envoyé et la requête émise à nouveau.
   - $M_{Fin\_mig} < M_{recep\_req}$: La requête est perçue par le contrôleur mémoire alors que le répertoire n'est plus contaminé, la requête et la réponse sont considérées valides.

L'ensemble des possibilités d'ordonnancement a été couvert, nous avons donc prouvé que toute requête prise en compte correspond à l'adresse effective désirée (TLB non-périmée).

9.3 Expérimentations

Afin d'évaluer l'efficacité de cette solution, nous allons comparer deux types de plateformes, l'une pourvue du système de migration de données (**P_MIG**) et l'autre non (**P_STD**), sur lesquelles nous allons exécuter une pile logicielle complète : système d'exploitation et applications multi-tâches.

Nous présentons ci-dessous l'environnement de simulation, le système d'exploitation et les applications utilisées.

9.3.1 Environnement de simulation et architectures modélisées

Nos plateformes de simulation sont construites à l'aide de composants SoCLib [soc] avec une précision CABA. Cet environnement de simulation nous permet d'exécuter une pile logicielle cross-compilée pour le jeu d'instructions des processeurs embarqués. Nous avons modélisé une architecture décomposée en 16 nœuds connectés à l'aide du réseau DSPIN (4x4) [MPCVG08]. Les nœuds sont constitués d'un crossbar (fig. 9.1) auquel sont reliés un processeur MIPS, une mémoire locale et d'autres mémoires ou périphériques. Les données et le code se trouvent dans des nœuds distincts afin d'éviter de pénaliser l'architecture P_STD. Dans la table 9.3.1 nous présentons l'ensemble des paramètres de cette architecture.

TAB. 9.2 – Paramètres des architectures modélisées, les paramètres (*) ne concernent que P_MIG

taille du cache données/instructions	$4kb/4kb$, (ligne de 32 bytes)
protocole de cohérence	*write-through write-invalidate*
temps de traversée d'un nœud DSPIN	5 cycles
nombre d'entrées dans la TLB	8, évincement aléatoire
taille d'une page	$4\ ko$

9.3.2 Validation de la solution

Dans cette section nous allons montrer à l'aide d'une application très simple, le comportement de l'architecture proposée.

Nous avons écrit l'application `placement_test`. Celle-ci, « active » les processeurs un à un. Ceux-ci vont réaliser en boucle des accès continus à une page mémoire spécifique (notée

D). Pour permettre l'espacement de l'activation des processeurs (dans un ordre prédéfini), ils réalisent une boucle d'attente pendant un grand nombre d'itérations avant d'activer le prochain processeur de la liste. Cette boucle d'attente réalise un grand nombre d'accès à une variable contenue dans la pile.

Dans la figure 9.8 nous observons le déroulement de l'application. Comme nous pouvions nous y attendre, les pages contenant les données en pile de chacun des processeur migrent vers les mémoires locales des processeurs.

La page D accédée par les processeurs dès leur activation, migre vers le nœud 12 dès le début de l'exécution car c'est le premier nœud activé. Lorsque le nombre des accès par les autres nœuds est suffisant, celle-ci migre vers le centre de la puce sur le nœud 9.

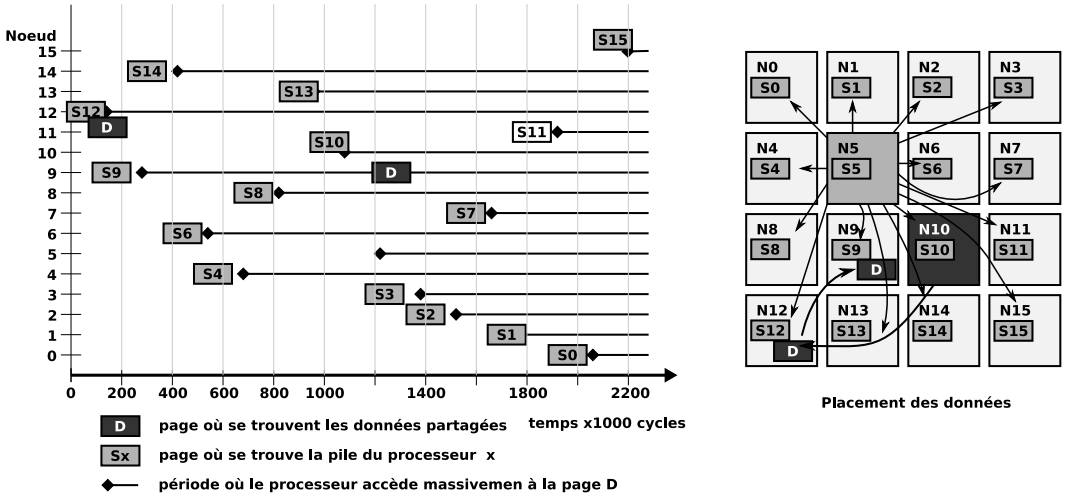


FIG. 9.8 – Évolution du placement des données dans l'application `placement_test`

Nous observons que la page D ne se déplace pas régulièrement sur la puce. Le problème est étudié plus en détail dans le chapitre suivant, mais il est dû en partie à un accès massif et continu par le processeur voisin (12). Ces accès bénéficient d'une latence très courte, et se font donc à une fréquence très élevée. Aucun des autres nœuds n'est en mesure d'accéder avec la même fréquence cette page car les latences d'accès sont beaucoup plus grandes. Ce comportement ne devrait pas se produire très souvent dans des applications réelles.

Le comportement observé est le comportement attendu, les processeurs bénéficient d'accès optimisés aux données privées (pile), et d'accès améliorés aux données fortement partagées (D).

9.3.3 Évaluation de la solution

Dans cette section nous allons évaluer notre solution à l'aide d'applications multi-tâches complexes. Nous utilisons quelques-unes des application que nous avons utilisé dans les chapitres précédents et dont nous rappelons les principales caractéristiques.

9.3.3.1 Applications

Nous avons sélectionné quatre applications de types différents. Les trois premières proviennent du jeu d'applications SPLASH-2 [CME+95].

ocean_c et water_ns : applications scientifiques où le calcul est découpé en zones de travail réparties sur n (16) tâches. Notre solution devrait réussir à exploiter correctement

le cloisonnement des données. Pour `ocean_c` le problème est une grille de 66x66 avec une précision de 10^{-3}, pour `water_ns` le problème est de 64 moles. La taille du problème a été réduite afin de diminuer les temps de simulation.

fft : application qui réalise une transformée de Fourrier rapide. L'application est essentiellement du calcul intensif sur un petit ensemble de données.

mjpeg : application de décodage vidéo. Les tâches (6) sont de natures différentes (calcul, manipulation de données, contrôle) et communiquent à travers des FIFO. L'unité de traitement sont des blocs de 8x8 pixels. Cette application est exécutée sur une plateforme plus petite à quatre processeurs (DSPIN 2x2). Nos résultats ont été obtenus pour un décodage de 5 images.

9.3.3.2 Système d'exploitation

Au lieu de Mutek, nous utilisons le système d'exploitation DNA [GP09]. Il permet d'écrire des programmes concurrents avec la librairie POSIX `pthreads`. Dans la suite, nous comparerons deux configurations logicielles :

AD : Architecture Dynamique, les tâches migrent entre les processeurs au gré des synchronisations. La pile, les données statiques et dynamiques se trouvent dans une mémoire globale située dans un des nœuds. Pour l'application `mjpeg`, la tâche critique s'exécute sur le processeur 3 devenant ainsi la tâche qui limite le système. Les autres tâches peuvent migrer normalement.

AS : Architecture Statique, les tâches sont placées sur un processeur et ne migrent pas. La pile est allouée dans la mémoire locale et les données statiques en mémoire globale. Les données allouées dynamiquement peuvent être explicitement placées en mémoire locale (`fft` uniquement), elles sont allouées par défaut en mémoire globale. Pour l'application `mjpeg`, la tâche critique s'exécute sur le processeur 1 et n'est plus la tâche qui limite le système. Les autres tâches sont placées sur les autres processeurs.

9.3.3.3 Résultats :

Nous avons exécuté les quatre applications sur deux architectures logicielles (AS, AD) et sur deux architectures matérielles (P_MIG et P_STD).

Temps d'exécution : les temps d'exécution de chacune des applications sont présentés dans la figure 9.9. Nous remarquons comme prévu que les architectures à placement statique et allocation en local des données (AS), offrent de meilleures performances que les architectures dynamiques (AD). Cet avantage est moins marqué pour l'application `mjpeg` car le placement statique des tâches de natures différentes ne permet pas d'exploiter pleinement la puissance de calcul de tous les processeurs.

Pour les quatre applications AD + P_MIG offre des meilleures performances que AD + P_STD. Pour `ocean_c, mjpeg et water_ns` le gain approche les 50%, sur la `fft` le gain est encore supérieur et atteint les 70%.

Pour l'application `mjpeg` le traitement d'un flux de données se prête moins à la répartition en pages de l'ensemble du travail. Malgré cela, les performances obtenues sont excellentes.

Bien qu'une architecture à placement statique optimale puisse être considéré comme une architecture de référence, nous observons que le système AD + P_MIG offre des meilleures performances (à l'exception de `mjpeg`). Ceci montre la difficulté d'optimiser une application au niveau utilisateur, la migration des données permet d'exploiter le placement de données que l'utilisateur ne peut pas contrôler (pile, données statiques, code).

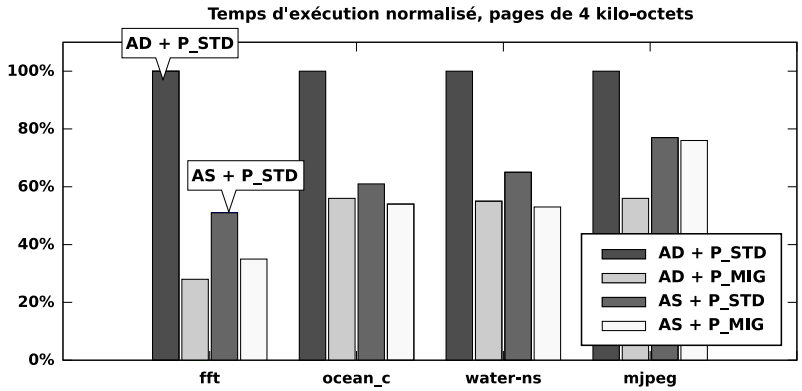


FIG. 9.9 – Temps d'exécution normalisé des applications.

Sur `ocean_c` le gain n'est que de 10% car le placement statique des tâches est très efficace, ceci se confirme par les résultats de l'architecture AS + P_MIG qui sont sensiblement égaux.

Nous observons que les performances des architectures AS + P_MIG et AD + P_MIG sont sensiblement équivalentes (à l'exception de `mjpeg`). Ceci montre que le coût de migration des tâches est faible et que notre solution de placement automatique des données s'affranchit du comportement de l'application.

L'application `mjpeg` avec un placement statique n'est pas idéale. Les tâches n'ont pas la même charge de travail et il est difficile de les répartir statiquement de façon optimale sur un ensemble de processeurs. Avec un placement statique, la migration des données offre un gain marginal dans ce cas précis.

Notons que pour d'autres applications il est certainement possible d'obtenir un placement statique des tâches, données et code optimal permettant d'obtenir des meilleures performances que notre système de migration de pages. Néanmoins, notre solution offrira de bons résultats quelles que soient l'application et le système d'exploitation.

Consommation d'énergie des accès à la mémoire : les accès à la mémoire consomment de l'énergie à plusieurs niveaux, l'un d'entre eux est le transfert sur le lien de communication.

Nous avons mesuré pour chaque nœud, la somme des distances de tous les accès. Cette distance est mesurée en nombre de nœuds traversés. Ainsi, pour chaque nœud nous avons mesuré la « distance totale parcourue » par les requêtes et nous en avons calculé la somme (cf. figure 9.10). Bien qu'une étude approfondie serait nécessaire, nous pouvons dans une première approximation, considéré que la consommation des accès est proportionnelle à leur distance. Ainsi, cette figure présente un première approximation de la consommation des différentes plateformes.

Nous pouvons observer que notre solution permet de réduire considérablement la « consommation d'énergie » de la plateforme AD + P_STD. L'économie réalisée est de l'ordre de 70%. Nous remarquons également que l'architecture AS + P_STD accède le plus souvent des données locales (en pile), ainsi, elle offre des performances égales ou supérieures à celles de l'architecture AD + P_MIG. La plateforme AS + P_MIG offre une amélioration faible des performances.

Afin de connaître le gain énergétique il conviendrait de modéliser précisément la consommation d'énergie. De plus, il faudrait évaluer la consommation engendrée par les

compteurs ajoutés, les migrations de pages et l'interrogation des TLB.

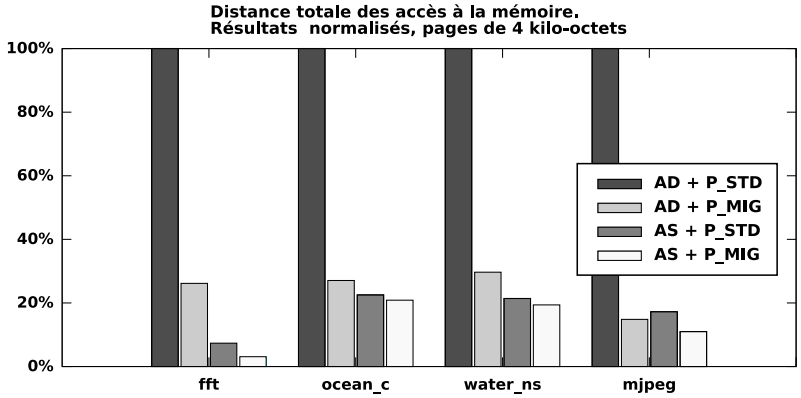


FIG. 9.10 – Distance totale parcourue par les accès à la mémoire.

Résultats divers : Nous avons également observé que le taux d'échec des TLB est de l'ordre de 1% (10% pour le `mjpeg`) avec seulement 8 entrées. L'utilisation de notre solution où la migration des données serait désactivée offrirait des performances sensiblement équivalentes à un système dépourvu du mécanisme de migration. Nous avons également observé une stabilisation rapide du système car seul 10% (approximativement) des migrations sont effectivement réalisées.

9.4 Conclusion

Nous avons présenté dans cet article une solution innovante permettant de placer au mieux les données dans un système multiprocesseur embarqué à hiérarchie mémoire partagée et distribuée. Cette solution est particulièrement adaptée aux systèmes composés d'applications, couches multiples d'abstraction et virtualisation où il devient impossible d'optimiser ou contrôler le placement des données. Les premiers résultats sont très encourageants et montrent que cette solution permet toujours d'obtenir une réduction du temps d'exécution pouvant atteindre 70%. De plus, les performances obtenues sont meilleures que celles des systèmes où le placement des données et la migration des tâches ont été optimisés. Les premiers résultats laissent présager une réduction de la consommation d'énergie très importante (70%). Les travaux futurs permettrons de confirmer ces résultats sur de plus grandes architectures et d'évaluer précisément la consommation d'énergie.

Nous détaillerons dans le prochain chapitre les limitations de cette étude, mais aussi les perspectives de recherche et des pistes pour à explorer.

Chapitre 10

Limitations et pistes de recherche de l'étude de migration des données

Dans ce chapitre nous allons exposer, dans un premier temps, les différentes limitations de notre étude. Puis, nous allons présenter différentes solutions alternatives ainsi que des axes de recherche encore inexplorés.

10.1 Limitations de l'étude et problèmes potentiels

10.1.1 Plus d'applications

Il conviendrait de tester un plus grand nombre d'applications pour confirmer les résultats présentés. Il serait également intéressant d'observer le comportement de l'architecture en fonction du taux de partage réel des données de l'application.

Par ailleurs, nous n'avons pas exploité la possibilité du système d'organiser les tâches par équipes. Chaque équipe est exécutée sur un groupe de processeurs spécifiques afin d'optimiser le placement des tâches en fonction des données. Si deux équipes correspondent à deux applications différentes nous devrions observer deux groupes de données se déplacer de part et d'autre de l'architecture à l'endroit où s'exécutent les équipes. Une telle configuration permet de conserver une certaine souplesse lors de l'exécution d'une application (migration dynamique des tâches) tout en facilitant la tâche de migration des données.

10.1.2 Passage à l'échelle

Nos architectures ne possèdent que 16 processeurs alors que se profilent déjà à court terme des architectures à plus de 100 processeurs. Nous n'avons pas réalisé pour l'instant d'expérimentations avec des architectures plus grandes. Notre solution ne devrait poser de problème lors du passage à l'échelle (100 - 1000 processeurs) à un détail près.

Dans l'état actuel, l'anneau de communication est une ressource partagée. Bien que ce ne soit pas une ressource critique, il n'est pour l'instant pas possible de réaliser plus d'une migration à la fois. Dans un cas extrême, il pourrait y avoir un effet de famine sur le traitement des requêtes de migration. Celles-ci seraient satisfaites avec un retard conséquent ce qui les rendrait potentiellement obsolètes et au choix de placement sous-optimal. Dans la section suivante, nous exposerons quelques idées pour palier à ce problème.

10.1.3 Variation des paramètres

Dans ces travaux nous n'avons pas exploré l'influence des différents paramètres. Que ce soient les paramètres de l'architecture (taille des caches, nombre de processeurs par nœud) ou ceux du système de migration (seuil de congestion, fréquence de migration).

La cause principale est un manque de temps pour réaliser les simulations et exploiter les résultats. En effet, le nombre de paramètres variables crée un nombre énorme de configurations différentes à tester. Les paramètres utilisés sont ceux que nous avons jugé pertinents au regards des résultats des quelques explorations que nous avons fait.

Il pourrait être intéressant de comparer notre solution à une architecture possédant un cache d'une taille plus grande, équivalente à l'ajout du surcoût en surface que présente notre solution. Nous sommes convaincus que notre solution sera tout de même meilleure au vu de nos résultats préliminaires et du fait que l'augmentation de performances due à l'augmentation de la taille des caches n'est pas linéaire.

10.2 Améliorations et variantes de la solution actuelle

Nous présentons ici un ensemble de réflexions sur des améliorations à apporter à notre solution, ou bien, des études qu'il conviendrait de réaliser.

10.2.1 Technique de déclenchement

Le critère de déclenchement des migrations que nous avons choisi est un seuil de fréquence d'accès atteint par un nœud sur une page. Une variante tout à fait envisageable serait un seuil atteint par un compteur de page.

La deuxième solution a l'avantage de représenter réellement une fréquence d'accès à la page dans son ensemble (par tous les nœuds). L'inconvénient est que les pages fortement partagées risquent d'avoir un nombre d'accès dans les compteurs de nœud qui ne soit pas représentatif de la réalité. Par conséquent la page risque d'être placée au mauvais endroit, voir même de migrer de façon continue entre certains nœuds.

La première solution (celle que nous avons utilisé), garantit qu'au moins un des nœuds a réalisé un nombre conséquent d'accès, qui nous le croyons, contribue à un placement plus qualitatif de la page. Son inconvénient est que la page candidate à la migration n'est peut être pas celle qui est la plus accédée. Néanmoins, si tel est le cas nous n'y voyons pas ici un inconvénient majeur car tôt ou tard les autres pages vont être également déplacées.

10.2.2 Compteurs d'accès à base de filtres de Bloom

Un des points qui mériterait d'être améliorer serait le coût en surface lié à l'usage des compteurs. Ceux-ci constituent sans contexte la partie la plus importante du coût d'implantation.

Une solution simple pour réduire la taille des compteurs est d'utiliser des filtres de Bloom [Blo70]. Un filtre de Bloom permet, grâce à l'usage de fonctions de hachage, de coder avec perte un ensemble de clés. Lorsque l'on teste l'existence d'une clé dans l'ensemble on peut obtenir un faux positif mais jamais un faux négatif.

Il est possible de réaliser des compteurs d'estimation à l'aide de filtres de Bloom. Dans [BKK+07] les auteurs présentent un module d'estimation du nombre de requêtes traitées pour une adresse 26bits particulière. Un tel compteur peut fournir une surestimation mais jamais une sous-estimation de la valeur.

L'usage de ces filtres permettrait de compresser les compteurs par nœud de chaque page. Ils pourraient être également utilisés pour compresser les compteurs des pages.

Une contrainte importante liée à l'utilisation d'un filtre de Bloom est qu'il est impossible de retirer une référence d'un ensemble. Par conséquent, leur usage en tant que compteur impose que tous les compteurs soient remis à zéro au même moment. Ceci est toujours le cas en ce qui concerne les compteurs par nœud, mais peut se révéler problématique pour les compteurs par pages.

Dans l'implantation actuelle lorsqu'une tentative de migration échoue, les compteurs de la page concernée sont remis à zéro. Par conséquent, une autre page susceptible d'être candidate à la migration peut être élue. De cette façon, lorsque la page la plus accédée ne peut pas migrer, il est tout de même possible d'en déplacer une autre.

Si l'on décide d'utiliser des compteurs à base de filtres de Bloom pour les compteurs de page, il faudrait alors être en mesure de désactiver les compteurs de la page la plus accédée (et qui ne peut être déplacée) sur une durée déterminée.

10.2.3 Prise en compte du coût en latence

Actuellement les compteurs permettant d'évaluer la contribution des processeurs au coût d'accès d'une page, augmentent d'une unité à chaque accès. Ainsi, comme nous l'avons vu précédemment les nœuds mémoire vont évaluer le coût d'accès à la page et celui qui proposera la valeur minimale sera sélectionné.

Dans cette solution on place la donnée à la meilleure position théorique en fonction du nombre d'accès réalisés. Cette méthode a un inconvénient. Elle ne prend pas en compte le coût d'accès réel de la donnée. Par exemple, il se pourrait que la page la plus accédée ne soit pas celle qui subisse le plus grand coût d'accès.

Ce problème est exacerbé par le fait que la fréquence des accès d'un processeur à une donnée dépend d'une part de leur fréquence dans le code exécuté, et d'autre part de la latence des accès (vitesse d'exécution du code). Ainsi, si l'on considère deux processeurs qui exécutent le même code, celui qui bénéficiera d'un meilleur placement des données aura un CPI supérieur et donc, aura une fréquence d'accès aux données supérieure. Nous avons pu observer ce phénomène lors de l'exécution de l'application `placement_simple`, la mobilité de la page D y était restreinte.

Une solution envisageable est de ne plus comptabiliser dans les compteurs la fréquence d'accès, mais le coût d'accès. Pour cela il suffit que les paquets contiennent une information supplémentaire : le temps de traversée.

Cette donnée peut être obtenue avec par exemple un additionneur dans chaque nœud de routage. Celui-ci incrémenterait la valeur contenue dans le paquet avec l'indice d'arrivée dans la file d'attente (beaucoup moins coûteux qu'un incrément à chaque cycle d'horloge).

La prise en compte de la latence au moment du déclenchement de la migration permet de classer les pages par coût d'accès et non plus par nombre d'accès. De plus, elle permettrait de compenser la diminution de la fréquence des accès due à latence du lien. Il reste cependant à évaluer le coût d'implantation dans les routeurs et voir si le gain obtenu est significatif.

Toutefois, la prise en compte de la latence dans les compteurs de nœud fausserait le calcul du placement optimal des données. Le placement ne minimiserait plus la fonction de coût établie précédemment (cf. 8.9, page 75).

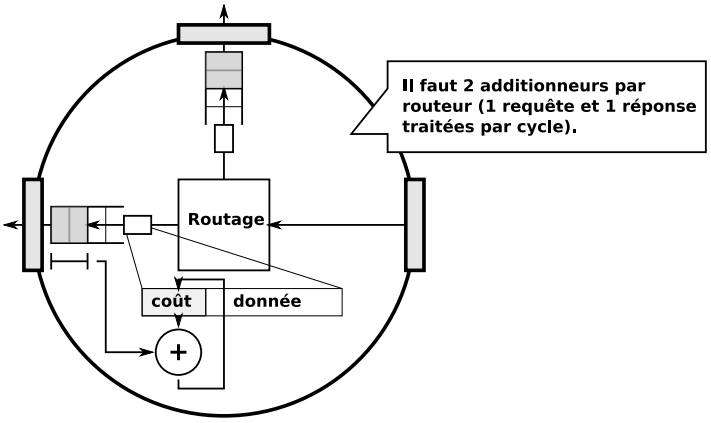


FIG. 10.1 – Exemple d'implantation dans un routeur de la comptabilisation du temps de traversée

10.3 Pistes de recherches et travaux futurs

Quelles que soient les améliorations apportées à notre solution, il existe deux points qui méritent une attention particulière : le contrôle par le système de la migration et le passage à grande échelle du système de migration.

10.3.1 Support par le système d'exploitation

Bien que nous militons pour l'usage d'une solution entièrement gérée par le matériel, certains algorithmes complexes requièrent un support logiciel. De plus, il est souhaitable de pouvoir avoir un contrôle sur le matériel pour certains types de situations ou d'applications

10.3.1.1 Soigner des pathologies

Nous n'avons pas encore découvert de pathologie particulière lors de nos expérimentations. Néanmoins, il est possible que dans certaines situations apparaisse un comportement pathologique faisant diminuer les performances du système.

Supposons par exemple qu'un ensemble de tâches synchronisées entre-elles, exécutent tour à tour un algorithme complexe sur un ensemble de données. Si la fréquence du transfert de contrôle entre les tâches est la même que celle de la prise de décision de migration des données, nous allons observer des mouvements migratoires continuellement en retard sur le comportement de l'application. Puisque lors de la migration des pages les données ne sont pas disponible, chaque tâche subira à son tour la pénalité de migration des données ainsi qu'un coût d'accès élevé à cause d'un placement sous-optimal.

Ce genre de problème peut probablement s'observer dans une application où un ensemble de tâches appliquent un traitement de façon successive sur un grand ensemble de données. Certains encodeurs vidéo ou audio doivent probablement présenter ce genre de comportements.

Dans ce genre de situation, il serait utile que le système ou l'application puissent contrôler la migration des données et détecter les comportements pathologiques.

10.3.1.2 Agglomérer les données pour économiser de l'énergie

Bien que notre solution permette d'économiser de l'énergie en réduisant la distance moyenne des accès, une autre solution plus efficace dans certains cas n'a pas été expérimentée.

En matériel il est assez simple de connaître la valeur maximum d'un ensemble de valeurs (i.e. la page la plus accédée) et de prendre une décision en conséquence. Toutefois, le système d'exploitation a la charge de diminuer la consommation du système lorsque celui-ci est sous-utilisé. Dans un système multicœur il est souhaitable de faire passer en mode basse consommation les nœuds qui ne sont pas utilisés.

Il serait également souhaitable que le système puisse faire de même avec les bancs mémoire. Pour cela, il doit être capable de relever les compteurs d'activité et appliquer une politique d'agglomération des pages sur certains nœuds afin de mettre en mode basse consommation les bancs mémoire qui ne sont plus utilisés.

10.3.1.3 Placement coordonné des tâches et des données

Une autre piste à explorer serait le couplage d'une tâche à un ensemble de données. La migration des données pourrait donc être contrôlée et anticipée par le système d'exploitation afin que celles-ci suivent le déplacement des tâches.

Des travaux dans ce sens ont déjà été réalisés par le passé mais ne bénéficiaient pas d'un système de placement de données transparent pour le logiciel. Il était essentiellement basé sur l'usage de la MMU et de l'allocation des pages.

10.3.1.4 Contrôle et déterminisme

Un autre aspect que nous n'avons pas abordé est le déterminisme. Notre solution induit dans le système des performances et des comportements hautement indéterministes. Ainsi, bien que notre solution permette d'obtenir un gain conséquent en terme de vitesse d'exécution, elle n'est absolument pas applicable dans le domaine des systèmes contraints de type temps-réel.

Ce type de système requiert des garanties en terme de pire temps d'exécution afin de pouvoir satisfaire les contraintes imposées par les concepteurs.

Afin de rendre utilisable notre plateforme pour ce genre d'application, il faut que le système puisse contrôler très précisément la migration des données.

10.3.1.5 *Data mining*

Une piste en cours d'exploration est l'application d'algorithmes du monde du data-mining au placement des données sur la puce. L'objectif est d'être en mesure d'extraire des patrons d'accès récurrents aux données. Ainsi, le système pourra prédire avec une grande justesse les déplacements de données nécessaires à l'optimisation du système.

Nos travaux seront réutilisés dans le cadre de ces recherches en tant que support matériel à la migration des données.

10.3.2 Passage à l'échelle et migrations en parallèle

Un problème potentiel de notre solution est le passage à l'échelle. Dans l'implantation actuelle nous avons pour une migration trois phases :

1. Détection du besoin de migration

2. Interrogation de tous les nœuds du système et sélection d'un destinataire.
3. Échange des données

La première phase est indépendante de toute ressource centralisée et n'a aucune influence sur le système (coût en temps). La deuxième phase n'a également pas de d'influence sur le système. Néanmoins, le temps nécessaire à la sélection du destinataire croît de façon linéaire avec le nombre de bancs mémoire du système.

Si ce temps venait à devenir trop important, la migration effective des données pourrait être effectuée avec un retard conséquent sur son déclenchement. Entre temps, l'application aura pu modifier son comportement et le nouveau placement devenu sous-optimal.

La troisième phase, l'échange des données a une influence directe sur le coût de la solution. Dans l'implantation actuelle, une page en cours de transfert n'est pas accessible. Néanmoins, une fois l'ensemble des tampons des routeurs remplis, le débit doit en théorie avoisiner les 1 donnée par cycle d'horloge. De plus, afin d'augmenter le débit il est possible d'augmenter la largeur des liens de communication.

Le principal problème du passage à l'échelle est que dans l'implantation actuelle il n'est pas possible de réaliser plusieurs transferts en même temps. Ainsi, la fréquence des migrations est limitée par la performance du réseau (un anneau dans notre cas) alors que le nombre de migrations requises par le système croît avec le nombre de nœuds intégrés.

Afin d'être efficace sur des architectures à 1000 processeurs, il faudra revoir l'implantation de la solution afin de rendre possible la migration en parallèle des données.

Chapitre 11

Conclusion

Les travaux présentés dans cette thèse ont, dans leur ensemble, exploré des techniques et trouvé des solutions pour améliorer la performances des accès à la mémoire dans les systèmes multiprocesseurs intégrés, tout en étant totalement transparents au programmeur, à quelque niveau qu'il se place.

Au début de ce manuscrit nous avons soulevé un certain nombre de questions, suite à notre étude et à nos travaux, nous sommes en mesure d'y apporter des réponses, au moins partielles.

L'évolution des contraintes dans les systèmes embarqués remet-elle en cause les choix réalisés quand au protocoles de cohérence des données à utiliser ?

Notre étude a considéré l'avènement de systèmes d'un nouveau type, des systèmes multiprocesseurs interconnectés par un NoC et possédant de la mémoire partagée et distribuée, le tout intégré dans une seule puce. L'intégration de l'ensemble des composants dans une même puce a redéfini une des contraintes fondamentales des systèmes multiprocesseurs : la bande passante disponible pour accéder aux données.

En effet, l'utilisation d'un réseau d'interconnexion point-à-point permet d'avoir une bande passante disponible très élevée. L'intégration et la distribution des mémoires dans la puce permet de franchir une étape supplémentaire, l'accès aux données n'est plus limité par la bande passante.

Dans le chapitre 4 nous avons réalisé une étude des avantages et inconvénients des protocoles de type *write-through invalidate* et *write-back invalidate*. Le protocole *write-through invalidate* bénéficie d'une implantation très simple ce qui permet de réduire la surface dédiée et donc son coût d'intégration. Il permet également d'avoir des données maintenues à jour dans les bancs mémoire, ce qui permet de réduire la latence d'accès en cas de données partagées en écriture par plusieurs processeurs.

Nous avons implanté les deux protocoles au niveau CABA à l'aide de l'environnement SoCLib. Nous avons réalisé un ensemble d'expérimentations afin de comparer les performances des deux protocoles.

Les résultats obtenus, en désaccord avec la littérature existante, prouvent que la redéfinition des contraintes remet en cause les choix réalisés. Nous en déduisons donc qu'il est nécessaire de revisiter les conclusions des travaux portants sur les machines parallèles discrètes en ayant en tête les contraintes de l'intégration.

L'utilisation d'un protocole de cohérence simple, *write-through invalidate* est-elle envisageable dans les systèmes multiprocesseurs intégrés ?

Les résultats présentés, ne permettent pas de conclure à un gain en performances avec l'utilisation d'un protocole *write-through invalidate*. Néanmoins, les critères qui déterminent l'adéquation d'une solution sont multiples. Tout d'abord, les performances obtenues sont tout à fait acceptables, l'écart mesuré avec un protocole de type *write-back invalidate* est inférieur à 20% sur l'ensemble des applications testées. De plus, la simplicité d'implantation du protocole induit un coût en surface très réduit.

Par ailleurs, nous sommes convaincus que le surcoût en bande passante est acceptable dans la plupart des cas. Il conviendrait tout de même d'évaluer la consommation du protocole avec une implantation réelle. En effet, le gain en trafic généré se traduit inévitablement par un gain en consommation d'énergie. Toutefois, l'économie en surface due à sa simplicité d'intégration devrait en réduire l'impact.

Ce protocole est donc un bon candidat lorsque l'objectif est de fournir, sur un système articulé autour d'un NoC, de la cohérence mémoire dans un système contraint en surface.

Comment comparer précisément des protocoles de cohérence de façon à abstraire les détails de l'implantation mais tout en continuant de modéliser les effets de bord et les interactions ?

Nous avons proposé une solution qui permet d'obtenir en simulation, une borne inférieure de la mesure des latences et du trafic généré par l'implantation d'un protocole de cohérence mémoire. Cette solution est basée sur une technique d'actions *omniscientes* qui permet de modifier l'état d'un système en un temps nul.

L'application de cette technique aux protocoles comparés précédemment, a permis de mesurer l'impact du maintien de la cohérence sur ces deux protocoles.

Nous avons montré que cet impact était plus faible pour le protocole *writ-through invalidate* et nous avons surtout montré qu'il était limité pour les applications réelles, rendant crédible la comparaison faite au travers des implantations.

Comment réduire la pénalité d'échec d'accès aux caches ?

En début de la seconde partie de nos travaux nous avons présenté une étude sur le placement optimal des données dans le système afin de réduire le CPI moyen des processeurs. Ce placement minimise la somme des coûts d'accès aux données par les différents processeurs.

Nous avons montré comment placer au mieux les données pour diminuer la latence des accès, mais également pour supprimer les points de congestion.

Comment placer ou déplacer les données sur une puce afin d'améliorer l'efficacité des accès ?

La principale contribution de ces travaux a été d'implanter une architecture (en simulation CABA) qui permette de placer et déplacer dynamiquement les données dans le système et ce, à l'insu des processeurs.

La migration des données se fait par pages à l'aide d'un réseau d'interconnexion supplémentaire. L'ensemble du protocole est géré entièrement en matériel et permet d'avoir une réactivité bien supérieure aux solutions logicielles existantes.

L'implantation réalisée introduit un nouveau niveau d'adressage invisible aux processeurs. Dans cette implantation nous avons trouvé un ensemble de solutions pour adresser toutes les données à tout moment, garantir la cohérence des adresses et choisir un emplacement idéal pour celles-ci.

Les premiers résultats obtenus sur une architecture 16 processeurs montrent un gain en vitesse d'exécution de 40% à 70% pour des applications de nature différente.

Notons par ailleurs que la simplicité d'une politique de cohérence de type *write-through invalidate* permet de simplifier l'implantation du protocole de migration des données. La solution présentée ouvre la voie à de nouveaux axes de recherche dans le placement dynamique des données à l'intérieur d'une puce.

La suite de ces travaux portera sur deux éléments essentiels. Premièrement un support du système d'exploitation pour permettre au logiciel d'avoir un contrôle sur le placement des données. Un des objectifs est entre autre de réaliser l'agrégation des données en vue de placer certains bancs mémoire en mode basse consommation. Deuxièmement, étudier des solutions permettant d'appliquer la technique proposée à des architecture à très grande échelle.

Chapitre 12

Publications

Les travaux réalisés au cours de la thèse et du master ont donné lieu à plusieurs publications internationales.

1. **P. Guironnet de Massas**, F. Pétrot, Migration de données dans les MPSoC : une solution matérielle, in *SYMPA'09, Toulouse*
2. **P. Guironnet de Massas**, F. Pétrot, Comparison of memory write policies for NoC based multicore cache coherent systems, in *Proceedings of the conference on Design, automation and test in Europe (DATE'08), 2008*
3. **P. Guironnet de Massas**, P. Amblard, F. Pétrot, On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration , in *VLSI Design Journal, 2007*
4. **P. Guironnet de Massas**, P. Amblard, Experiments around SPARC LEON-2 for MPEG encoding , in *International Conference Mixed Design of Integrated Circuits and Systems (MIXDES'06), 2006,* ***Outstanding paper award***

Bibliographie

[AAHV91] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of hardware and software cache coherence schemes. In *ISCA '91 : Proceedings of the 18th annual international symposium on Computer architecture*, pages 298–308. ACM Press, 1991. 5.1

[AB84] James Archibald and Jean Loup Baer. An economical solution to the cache coherence problem. In *ISCA '84 : Proceedings of the 11th annual international symposium on Computer architecture*, pages 355–362, New York, NY, USA, 1984. ACM. 3.3.2

[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols : evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4) :273–298, 1986. 3.3.1

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA '88 : Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 280–298. IEEE Computer Society Press, 1988. 3.3.2

[BCP62] L. Bloom, M. Cohen, and S. Poter. Consideration in the design of a computer with a high logic-to-memory speed ratio. In *Proc. Gigacycles Computing Systems, AIEE, Winter Meeting*, Jan. 1962. 2.5.1, 3.1

[Bel04] Frank Bellosa. When physical is not real enough. In *EW11 : Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 25, New York, NY, USA, 2004. ACM. 7.1.1.2

[BKK+07] Arkaprava Basu, Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose Martinez. Scavenger : A new last level cache architecture with global block priority. In *MICRO '07 : Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, Washington, DC, USA, 2007. IEEE Computer Society. 10.2.2

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7) :422–426, 1970. 10.2.2

[BMW06] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. Asr : Adaptive selective replication for cmp caches. In *MICRO 39 : Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society. 7.2.1

[BSS+06] G. Beltrame, D. Sciuto, C. Silvano, D. Lyonnard, and C. Pilkington. Exploiting tlm and object introspection for system-level simulation. In *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, pages 100–105, 3001 Leuven, Belgium, 2006. European Design and Automation Association. 5.4

[BT98] S. Basu and J. Torrellas. Enhancing memory use in simple coma : Multiplexed simple coma. *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 152–161, Feb 1998. 7.1.1.2

[BW04] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37 : Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society. 7.2.1

[BZ07a] S. Bourduas and Z. Zilic. A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing. In *First International Symposium on Networks-on-Chip*, pages 195–204, May 2007. 9.2.1

[BZ07b] S. Bourduas and Z. Zilic. Latency reduction of global traffic in wormhole-routed meshes using hierarchical rings for global routing. In *IEEE International Conf. on Application -specific Systems, Architectures and Processors*, pages 302–307, July 2007. 9.2.1

[CDV+94] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS-VI : Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 12–24, New York, NY, USA, 1994. ACM. 7.1.1.2

[CF78] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *ieee Transactions on Computers*, C-27 :1112–1118, 1978. 3.3.2, 5.1

[CHPS99] A.E. Condon, M.D. Hill, M. Plakal, and D.J. Sorin. Using lamport clocks to reason about relaxed memory models. *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 270–278, Jan 1999. 6.3, 6.3

[CME+95] Steven Cameron, Wooand Moriyoshi, Oharaand Evan, Torrieand Jaswinder, Pal Sing, and Anoop Gupta. The splash-2 programs : Characterization and methodological considerations. In *ISCA '95 : Proceedings of the 22nd annual international symposium on Computer Architecture*. IEEE Computer Society, 1995. 4.2.2.2, 9.3.3.1

[CML03] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the memory page migration influence in the system performance : the case of the sgi o2000. In *ICS '03 : Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129, New York, NY, USA, 2003. ACM. 7.1.1.2

[CPV05] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. *SIGARCH Comput. Archit. News*, 33(2) :357–368, 2005. 7.2.1

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA '06 : Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society. 7.2.1

[Fra84] S. J. Franck. Tightly coupled multiprocessor systems speed memory access time. In *Electron*, volume 57, pages 164 – 169, Janv. 1984. 3.3.1

[FW97] Babak Falsafi and David A. Wood. Reactive numa : a design for unifying s-coma and cc-numa. In *ISCA '97 : Proceedings of the 24th annual international*

symposium on Computer architecture, pages 229–240, New York, NY, USA, 1997. ACM. 7.1.1.2

[GG00] Pierre Guerrier and Alain Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proc. of Design Automation and Test in Europe*, pages 250–256, March 2000. 2.4

[GH93] Stephen R. Glodsmchidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *SIGMETRICS ' 93 : Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 146–157. ACM, 1993. 5.2.1

[GKO+00] Jeff Gibson, Robert Kunz, David Ofelt, Mark Horowitz, John Hennessy, and Mark Heinrich. Flash vs. (simulated) flash : closing the simulation loop. *SIGPLAN Not.*, 35(11) :49–58, 2000. 5.3.1

[Goo83] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA*, pages 124–131, 1983. 3.1

[Goo98] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '98 : 25 years of the international symposia on Computer architecture (selected papers)*, pages 255–262. ACM Press, 1998. 3.3.1

[GP09] Xavier Guérin and Frédéric Pétrot. A System Framework for the Design of Embedded Software Targeting Heterogeneous Multi-Core SoCs. In *IEEE International Conf. on Application -specific Systems, Architectures and Processors*, Boston, MA, 2009. To be published. 9.3.3.2

[HCM09] H. Hammoud, S. Cho, and R. Melhem. Acm : An efficient approach for managing shared caches in chip multiprocessors. *Proceedings of the 4th Int'l Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 355–372, 2009. 7.2.1

[HPS03] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. Design and implementation of power-aware virtual memory. In *ATEC '03 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association. 7.1.2

[il99] Vijayaraghavan Soundararajan incomplete list. Flexible use of memory for replication/migration incache-coherent dsm multiprocessors. Technical report, Stanford, CA, USA, 1999. 7.1.1.2

[ITR07] ITRS. International technology roadmap for semiconductors. In *System Drivers*, 2007. (document), 1, 1.1

[JBP06] Ahmed A. Jerraya, Aimen Bouchhima, and Frédéric Pétrot. Programming models and hw-sw interfaces abstraction for multi-processor soc. In *DAC '06 : Proceedings of the 43rd annual conference on Design automation*, pages 280–285, New York, NY, USA, 2006. ACM. 5.3.1

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X : Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, New York, NY, USA, 2002. ACM. 7.2.1

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA '85 : Proceedings of the 12th annual international symposium on Computer architecture*, pages 276–283. IEEE Computer Society Press, 1985. 3.3.1

[KTJR05] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11) :32–38, 2005. 1

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, July 1978. 6.3

[LC96] Tom Lovett and Russell Clapp. Sting : a cc-numa computer system for the commercial marketplace. In *ISCA '96 : Proceedings of the 23rd annual international symposium on Computer architecture*, pages 308–317, New York, NY, USA, 1996. ACM. 7.1.1.1

[LF00] An-Chow Lai and Babak Falsafi. Comparing the effectiveness of fine-grain memory caching against page migration/replication in reducing traffic in dsm clusters. In *SPAA '00 : Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 79–88, New York, NY, USA, 2000. ACM. 7.1.1.2

[LFZE00] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *ASPLOS-IX : Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 105–116, New York, NY, USA, 2000. ACM. 7.1.2

[LH91] Anders Landin and Seif Haridi. Ddm - a cache-only memory architecture. Technical report, 1991. 7.1.1.1

[LL97] James Laudon and Daniel Lenoski. The sgi origin : a ccnuma highly scalable server. In *ISCA '97 : Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251, New York, NY, USA, 1997. ACM. 7.1.1.2, 9.2.3, 9.2.8.2

[LPB06] Mirko Loghi, Massimo Poncino, and Luca Benini. Cache coherence tradeoffs in shared-memory mpsocs. *Trans. on Embedded Computing Sys.*, 5(2) :383–407, 2006. 3.3.1

[McC84] E. M. McCreight. The dragon computer system : An early overview. In *Technical Report, Xerox Corporation*, September 1984. 3.3.1

[MGP+03] P. Marchal, J. I. Gomez, L. Pinuel, D. Bruni, L. Benini, F. Catthoor, and H. Corporaal. Sdram-energy-aware memory allocation for dynamic multi-media applications on multi-processor platforms. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10516, Washington, DC, USA, 2003. IEEE Computer Society. 7.1.2

[MH07] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. In *ISCA '07 : Proceedings of the 34th annual international symposium on Computer architecture*, pages 46–56, New York, NY, USA, 2007. ACM. 7.2.1

[MHW02] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02 : Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM. 5.4

[MPCVG08] I. Miro-Panades, F. Clermidy, P. Vivet, and A. Greiner. Physical implementation of the dspin network-on-chip in the faust architecture. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 139–148, April 2008. 9.3.1

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4) :92–99, 2005. 5.4

[NAB+95] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The s3.mp scalable shared memory multiprocessor. In *Proc. Int'l Conf. Parallel Proceeding*, 1995. 7.1.1.1

[Naf06] S. Naffziger. High-performance processors in a power-limited world. In *VLSI Circuits, 2006. Digest of Technical Papers. 2006 Symposium on*, pages 93–97, 0-0 2006. 1

[NPP+00] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A case for user-level dynamic page migration. In *ICS '00 : Proceedings of the 14th international conference on Supercomputing*, pages 119–130, New York, NY, USA, 2000. ACM. 7.1.1.2

[NvdP99] Lisa Noordergraaf and Ruud van der Pas. Performance experiences on sun's wildfire prototype. In *Supercomputing '99 : Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38, New York, NY, USA, 1999. ACM. 7.1, 7.1.1.2

[OCC08] Frank E.B. Ophelders, Samarjit Chakraborty, and Henk Corporaal. Intra- and inter-processor hybrid performance modeling for mpsoc architectures. In *CODES/ISSS '08 : Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 91–96, New York, NY, USA, 2008. ACM. 5.4

[PG03] F. Petrot and P. Gomez. Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 51–56 suppl., 2003. 4.2.3

[PGJ+05] Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *Computers, IEEE Transactions on*, 54(8) :1025–1040, Aug. 2005. 2.2.2

[PMT04] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. Microlib : A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37 : Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society. 5.3.1

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84 : Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354. ACM Press, 1984. 3.3.1, 5.1

[QY89] Bao-Chin Liu Qing Yang, Laxmi N. Bhuyan. Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. *ieee Transactions on Computers*, 38, 1989. 3.3.1, 5.1

[SJG92] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *ISCA '92 : Proceedings of the 19th annual international symposium on Computer architecture*, pages 80–91, New York, NY, USA, 1992. ACM. 7.1.1.1

[soc] Soclib project. `http://www.soclib.lip6.fr/Home.html`. 4.2.1, 9.3.1

[Ste90] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6) :12–24, June 1990. 3.3.2

[SWCL95] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 276–285, 1995. 7.1.1.2

[Tan76] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *AFIPS National Computer Conference*, volume 45, pages 749–753, 1976. 3.3.2

[TM94] M. Tomasevic and V. Milutinovic. Hardware approaches to cache coherence in shared-memory multiprocessors, part 1. In *Micro, IEEE*, volume 14, page 52, 1994. 3.3.1, 3.3.2

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly : a multiprocessor workstation. In *ASPLOS-II : Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 164–172, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. 3.3.1

[vci00] Virtual component interface standard (ocb 2 2.0). *VSI Alliance*, 2000. 4.2.1

[VDGR96] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *SIGOPS Oper. Syst. Rev.*, 30(5) :279–289, 1996. 7.1.1.2

[YRC+00] S. Yoo, K. Rha, Y. Cho, J. Jung, and K. Choi. Performance estimation of multiple-cache ip-based systems : case study of an interdependency problem and application of an extended shared memory model. In *CODES 2000*, pages 77 – 81. ACM Press, 2000. 5.2.1

[ZT97] Zheng Zhang and Josep Torrellas. Reducing remote conflict misses : Numa with remote cache versus coma. In *HPCA '97 : Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, page 272, Washington, DC, USA, 1997. IEEE Computer Society. 7.1.1.1

RÉSUMÉ

Les capacités d'intégration sur une puce continuent de croître contrairement à l'augmentation des fréquences de fonctionnement. Afin de fournir toujours plus de puissance de calcul les architectes intègrent désormais plusieurs dizaines de processeurs dans une même puce.

L'accès aux données est un point clé de la performance des systèmes. Le but de nos travaux est d'en améliorer l'efficacité à l'aide de solutions entièrement transparentes au logiciel. Notre contexte vise les machines multiprocesseurs interconnectées à l'aide d'un NoC qui possèdent des caches L1 et de la mémoire partagée et distribuée. L'intégration de la mémoire dans la puce rend possible son utilisation dans l'embarqué.

Dans une première partie nous montrons que la redéfinition des contraintes dans les systèmes embarqués rend l'utilisation du protocole de cohérence *write-through invalidate* envisageable dans ce type d'architectures. Nous présentons également une solution innovante pour évaluer et comparer les protocoles de cohérence mémoire.

Dans une deuxième partie nous présentons une solution innovante à la migration des données dans la puce. Celle-ci, gérée par le matériel, vise à placer dynamiquement et intelligemment les données afin de diminuer le coût d'accès moyen à la mémoire.

MOTS-CLÉS : systèmes sur puce, accès aux données, caches, NoC, migration de données, placement de données.

TITLE

Study of methods and mechanisms for software-seamless data accesses in a multiprocessor system-on-chip.

ABSTRACT

The integration capabilities are continuously increasing in opposition to execution frequencies. In order to provide evermore computational power, architects integrates dozen of processors in the same chip.

Data access is a corner stone in system performances. The main goal of our work is to enhance them using software-seamless solutions. Our context targets NoC based muliprocessor systems which contains L1 caches and distributed shared memory. Memory integration allows to use the system in an embedded device.

In a first part, we show that the constraints evolution in embedded systems makes possible the usage of a write-through invalidate coherence protocol in such systems. We present also a novel method to evaluate and compare memory coherence protocols.

In the second part we present a novel solution for on-chip data migration. It is hardware driven, and it dynamically and wisely places the data in order to decrease the mean cost access to memory.

ISBN : 978-2-84813-145-0