Gestion de la cohérence des données dans les systèmes multiprocesseurs sur puce

Hajer Chtioui



N°d'ordre: 11/34

L'École Nationale d'Ingénieurs de Sfax &
L'Université de Valenciennes et du Hainaut Cambrésis

THESE

En vue de l'obtention du

DOCTORAT

Dans la discipline Ingénierie des Systèmes Informatiques

Par

Hajer CHTIOUI

Gestion de la cohérence des données dans les systèmes multiprocesseurs sur puce

Thèse soutenue le 21 Décembre 2011, devant le jury composé de :

M. Rached TOURKI	**Professeur FSM**	Président & Rapporteur
M. Bertrand GRANADO	**Professeur des Universités à l'ENSEA**	Rapporteur
M. Jean-Luc DEKEYSER	**Professeur Université Lille 1**	Examinateur
M. Ahmed Chiheb AMMARI	**Maître de conférence INSAT**	Examinateur
M. Mohamed ABID	**Professeur ENIS**	Directeur de Thèse
M. Smail NIAR	**Professeur Université de Valenciennes**	Directeur de Thèse

THÈSE EN CO-TUTELLE ENTRE L'ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'ENIS DE L'UNIVERSITÉ DE SFAX ET L'ÉCOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR UNIVERSITÉ LILLE NORD-DE-FRANCE

ENIS CES Lab. Computer & Embedded Systems
UVHC Université de Valenciennes et du Hainaut-Cambrésis
LAMIH Laboratoire d'Automatique de Mécanique et d'Informatique Industrielles et Humaines

N°d'ordre: 11/34

L'École Nationale d'Ingénieurs de Sfax &
L'Université de Valenciennes et du Hainaut Cambrésis

THESE

En vue de l'obtention du

DOCTORAT

Dans la discipline Ingénierie des Systèmes Informatiques

Par

Hajer CHTIOUI

Gestion de la cohérence des données dans les systèmes multiprocesseurs sur puce

Thèse soutenue le 21 Décembre 2011, devant le jury composé de :

M.	Rached TOURKI	Professeur FSM	Président & Rapporteur
M.	Bertrand GRANADO	Professeur des Universités à l'ENSEA	Rapporteur
M.	Jean-Luc DEKEYSER	Professeur Université Lille 1	Examinateur
M.	Ahmed Chiheb AMMARI	Maître de conférence INSAT	Examinateur
M.	Mohamed ABID	Professeur ENIS	Directeur de Thèse
M.	Smail NIAR	Professeur Université de Valenciennes	Directeur de Thèse

THÈSE EN CO-TUTELLE ENTRE L'ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'ENIS DE L'UNIVERSITÉ DE SFAX ET L'ÉCOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR UNIVERSITÉ LILLE NORD-DE-FRANCE

Sommaire

Liste des figures

Liste des tableaux

Chapitre 1

Introduction générale

1. Contexte

Les applications mobiles et embarquées sont de plus en plus complexes et variées. Elles exigent par conséquent des architectures performantes, flexibles et un court temps de conception. Pour répondre à ces exigences, deux approches sont utilisées : La première consiste à offrir aux architectures monoprocesseurs un haut niveau de performance par une augmentation importante de la fréquence d'horloge. Cette solution a été largement utilisée dans le passé en particulier dans les GPP (*General Purpose processor*). L'inconvénient majeur de cette solution est qu'elle fait augmenter très sensiblement la consommation d'énergie. Plusieurs propositions ont été faites, comme le DVFS pour (*Dynamic Voltage and Frequency Scaling*) (Choi *et al.*, 2004) afin de mieux contrôler la fréquence et ainsi réduire la consommation d'énergie lorsque le processeur n'a pas besoin d'une fréquence élevée. L'utilisation des architectures MPSoCs (*Multi-Processor System-on-Chip*) est la deuxième approche. L'idée consiste à remplacer une architecture monoprocesseur et complexe par des processeurs de structures plus simples mais en nombre plus grand. Comme la consommation de puissance est proportionnelle à la fréquence d'horloge, sous certaines conditions, cette approche permet d'augmenter les performances tout en limitant la consommation de puissance.

Néanmoins, l'utilisation des MPSoCs exige un modèle de programmation simple. Le choix que nous avons fait dans cette thèse est d'utiliser des architectures MPSoCs et un modèle de programmation à mémoire partagée. Ce dernier simplifie la transformation des codes qui ont été conçus pour les systèmes monoprocesseurs et les adapter relativement facilement aux systèmes multiprocesseurs. Pour éviter le problème du goulot d'étranglement au niveau de la mémoire partagée et permettre une extensibilité facile de l'architecture, nous avons proposé deux solutions. D'un côté, chaque processeur est équipé d'une mémoire cache pour réduire les accès à la mémoire partagée, et de l'autre côté le traditionnel bus partagé est remplacé par un NoC (*Network-on-Chip*).

L'intégration des mémoires caches dans une architecture pose cependant le problème de cohérence des données dans les caches (John *et al.*, 2006). En effet, afin de gérer les copies multiples d'une même donnée dans les différents caches, il est nécessaire de faire appel à un protocole de gestion de la cohérence. Ce protocole garantit que les données partagées ont des valeurs identiques dans tous les caches de l'architecture. Les travaux de cette thèse se placent dans ce même contexte. Il s'agit d'étudier le problème de cohérence de données en caches dans les systèmes multiprocesseurs et de proposer une solution efficace

dédiée aux MPSoCs à mémoire partagée et utilisant des NoCs complexes comme le crossbar et les réseaux multi-étages.

2. Problématique

Dans la littérature, un nombre important de travaux a été réalisé pour concevoir des protocoles de cohérence des données dans les caches qui réduisent la surcharge en temps et en ressources matérielles additionnelles. Généralement, ces travaux se regroupent en deux familles: le protocole par invalidation et le protocole par mise à jour. Le protocole par invalidation consiste à envoyer des messages d'invalidation aux caches qui contiennent une copie du bloc modifié. À l'opposé, avec le protocole par mise à jour la nouvelle valeur du mot modifié est envoyée aux caches qui se partagent le bloc contenant ce mot. Néanmoins, ces travaux souffrent de deux points :

1. Les protocoles proposés sont pour la plupart dédiés aux architectures haute-performances et ne prennent pas donc en compte les contraintes en énergie et en ressources que nous trouvons dans les architectures embarquées.

2. Ces protocoles supposent aussi que les schémas d'accès aux données mémoires et la façon avec laquelle les processeurs communiquent entre eux sont identiques tout le long de l'application. Comme, les systèmes embarqués deviennent de plus en plus ouverts, les différentes applications à exécuter sont variées et peuvent contenir par conséquent des schémas de partage de données différents et variés. A titre d'exemple, le nombre de lecteurs et de rédacteurs pour une donnée mémoire peut changer d'une application à une autre, d'une donnée à l'autre dans le programme et d'un instant à l'autre en cours d'exécution. Il est donc intéressant d'utiliser un protocole de gestion de la cohérence des données dans les caches qui soit dynamique et qui prenne en compte les besoins des différentes applications et de leur évolution dans le temps. Ainsi l'architecture est capable de s'adapter dynamiquement aux besoins des applications.

3. Contributions

Afin de résoudre les problèmes évoqués précédemment, les contributions de notre travail sont multiples:

La proposition d'un nouveau protocole pour la cohérence des données dans les caches

1. Un protocole hybride invalidation/mise à jour qui tire profit des avantages de ces deux protocoles et dynamique qui est capable d'analyser dynamiquement les accès mémoires des processeurs et de choisir le protocole de gestion de la cohérence le plus adapté pour chaque donnée mémoire.

2. Un protocole de gestion de la cohérence des données dans les caches spécialement conçu pour les MPSoCs à mémoire partagée. Ce dernier consomme une quantité relativement faible de temps et d'énergie et exige une quantité relativement faible de ressources additionnelles.

La proposition d'une architecture matérielle qui facilite l'implémentation du protocole proposé

3. Une architecture matérielle qui facilite et optimise les performances du protocole de gestion de la cohérence des données dans les caches est proposée dans cette thèse. Cette architecture permet de réduire le nombre des opérations de maintien de la cohérence réalisées à travers le réseau d'interconnexion. Par conséquent, grâce à cette architecture la surcharge en temps et en consommation de puissance du protocole proposé devient négligeable.

L'étude des profils d'un ensemble d'applications de traitement de signal intensif

4. Une étude des profils d'un ensemble d'applications de traitement de signal intensif est réalisée pour évaluer l'intérêt de notre solution pour la cohérence des données dans les caches dans les MPSoCs. En effet, la présence de différents modèles d'accès aux données partagées dans une application est l'un des paramètres qui affectent les performances des protocoles de cohérence des données dans les caches. En conséquence, il est nécessaire d'étudier l'influence de ce paramètre sur le protocole proposé.

L'évaluation des performances et de la consommation d'énergie du protocole proposé

5. Une mesure de performances et de la consommation d'énergie est réalisée à l'aide de plusieurs benchmarks parallèles. Les résultats expérimentaux ont montré que l'utilisation du protocole hybride proposé en conjonction avec l'architecture proposée permet de réduire le temps d'exécution et la consommation d'énergie.

4. Plan

Ce manuscrit est organisé selon le plan suivant :

Chapitre 2 : État de l'art sur les unités de mémorisation dans les systèmes multiprocesseurs. Dans ce chapitre nous présentons le contexte général de notre étude, centrée sur les architectures mémoires dans les systèmes multiprocesseurs. Ensuite, nous focalisons sur l'étude du problème de cohérence des données dans les caches pour les systèmes multiprocesseurs embarqués. Enfin, nous regroupons les différents protocoles de cohérence des données dans les caches qui sont apparus dans la littérature.

Chapitre 3 : Un protocole hybride et adaptatif pour la cohérence des données dans les caches dans les MPSoCs. Dans ce chapitre nous proposons un nouveau protocole hybride et adaptatif pour la cohérence des données dans les caches dédié aux MPSoCs à mémoire partagée utilisant des NoCs complexes. Tout d'abord, nous décrivons les critères de choix du protocole proposé. Ensuite, nous détaillons le principe de fonctionnement de ce protocole à l'aide des machines à états finis (FSMs) pour *Finite State Machines*.

Chapitre 4 : Architecture matérielle pour le protocole proposé. Nous présentons dans ce chapitre l'architecture matérielle que nous proposons afin de faciliter l'implémentation du protocole proposé dans le chapitre précédent et optimiser ainsi ces performances en termes de temps d'exécution et de consommation d'énergie.

Chapitre 5 : Parallélisation des benchmarks pour architecture MPSoC. Dans ce chapitre nous étudions les différentes formes permettant l'exploitation du parallélisme dans une application donnée. Ensuite, nous proposons une classification des données partagées issue de ces formes de parallélisme. En se basant sur cette classification, nous parallélisons un ensemble d'applications de traitements de signal intensif. La parallélisation de ces applications est réalisée de façon à avoir différents modèles d'accès aux données partagées. Ce qui permet alors d'évaluer le protocole hybride proposé avec une variété de modèles de programmation parallèle.

Chapitre 6 : Environnement de simulation et résultats expérimentaux. Dans ce chapitre, nous présentons l'environnement de simulation que nous avons adopté afin d'évaluer le protocole proposé. Une présentation des résultats expérimentaux réalisés avec différents benchmarks parallèles sera aussi effectuée. Nous montrons à l'aide de ces résultats que le protocole hybride proposé et l'architecture proposée pour l'implémenter offrent un niveau de

performance et de consommation de puissance intéressant.

Chapitre 7 : Conclusion et perspectives. Dans ce dernier chapitre, nous finalisons ce manuscrit tout d'abord par le bilan des travaux effectués. Ensuite, nous présentons quelques perspectives à nos travaux.

Chapitre 2

État de l'art sur les unités de mémorisation dans les systèmes multiprocesseurs

1. Introduction

Afin de satisfaire les nouvelles applications embarquées qui sont gourmandes en terme de puissance de calcul, les architectures parallèles basées sur un grand nombre de processeurs sont de plus en plus utilisées. Parallèlement à cela, et grâce aux avancées technologiques, il est désormais possible d'intégrer plusieurs processeurs sur une même puce. De ce fait, la tendance actuelle s'oriente vers l'utilisation d'architecture embarquée multiprocesseur ou *Multi-Processor System-on-Chip* (MPSoC). Or, dans ces systèmes, les unités de mémorisation jouent un rôle de première importance. En effet, des études ont montré pour les applications du multimédia, des jeux, ou de la télécommunication, une partie importante du traitement de l'application concernant le transfert de données vers ou depuis la mémoire. Par suite, ces applications exigent l'intégration de plusieurs unités de mémorisation de différents types et différentes tailles. Ceci est confirmé aussi par les prévisions de l'ITRS (*International Technology Roadmap for Semiconductors*) (ITRS, 2009) (figure 1). Cette figure montre que dans les prochaines années et parallèlement à la croissance du nombre de cœurs de processeurs intégrés sur une seule puce, la taille de la mémoire va continuer à augmenter fortement. Concernant le nombre de cœurs de processeurs, cette tendance a d'ailleurs déjà commencé. En effet, récemment Intel a développé une puce intégrant 80 cœurs de processeurs (Intel, 2010).

Nous pouvons alors affirmer que les unités de mémorisation impactent significativement les performances et la consommation de l'architecture MPSoC. Dans cette thèse, nous nous intéressons en particulier aux MPSoCs à mémoire partagée et qui utilisent des réseaux d'interconnexions complexes, comme le crossbar ou les réseaux multi-étages. En effet, ce type d'architecture MPSoC est très intéressant du fait qu'il facilite à la fois le développement des applications parallèles, grâce à leur modèle de programmation, et aussi l'intégration d'un nombre important de processeurs. Néanmoins, pour ces systèmes, la gestion de la cohérence des données dans les caches représente un point crucial de l'architecture. Dans ce contexte, ce chapitre met l'accent sur ce problème et éventuellement sur les différentes solutions qui sont proposées pour le résoudre.

Ce chapitre présente tout d'abord les architectures mémoires dans les systèmes multiprocesseurs à travers la section 2. Ensuite, nous nous intéresserons plus particulièrement aux architectures mémoires multi-bancs dans la section 3 et aux architectures mémoires dans les systèmes embarqués dans la section 4. Par la suite, dans les sections 5 et 6 nous expliquerons respectivement le problème de cohérence des données

dans les caches et le problème de consistance de données dans les mémoires dans les systèmes multiprocesseurs à mémoire partagée. Dans le reste de ce chapitre, nous focalisons sur l'étude du problème de cohérence des données dans les caches pour les systèmes multiprocesseurs embarqués. En effet, la section 7 décrit les protocoles de maintien de la cohérence des données dans les mémoires caches dans les MPSoC. Il s'agit en particulier des protocoles par invalidation des données et par mise à jour des données. La section 8 réalise un état de l'art sur les mécanismes de cohérence des données dans les caches, l'accent est mis sur les solutions matérielles. L'objectif de la section 9 consiste à comparer le protocole par invalidation avec le protocole par mise à jour et montrer l'utilité d'un nouveau protocole hybride de cohérence des données dans les caches qui combine les deux protocoles. Ceci, sera éventuellement, l'objectif de la dernière section 10 qui consiste à regrouper les différents protocoles hybrides (Invalidation/ Mise à jour) qui sont apparus dans la littérature.

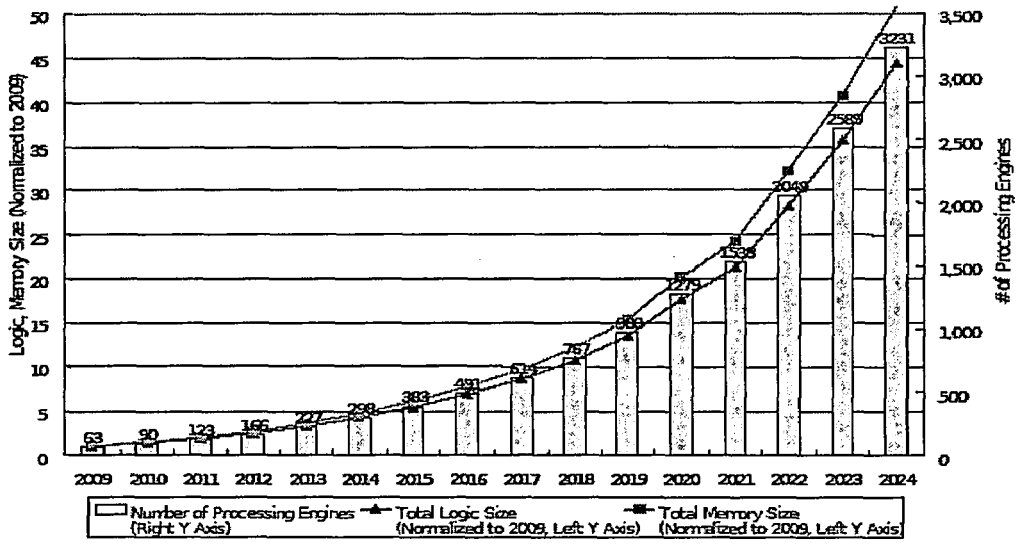


Figure 1. *Prévisions ITRS sur le nombre de cœurs de processeurs et la mémoire dans les SoCs (ITRS, 2009)*

2. Les architectures mémoires dans les systèmes multiprocesseurs

Flynn (Flynn, 1966) a proposé une classification des systèmes multiprocesseurs selon le parallélisme dans les flots d'instructions et dans les flots de données. Il a dégagé quatre catégories d'architectures multiprocesseurs, nous décrivons ici:

- Un seul flot d'instructions, un seul flot de données (*Single Instruction stream and Single Data* stream ou SISD): C'est le cas du système avec un seul processeur.
- Un seul flot d'instructions et plusieurs flots de données (*Single Instruction stream and Multiple Data stream* ou SIMD) : Dans cette catégorie, une même instruction est exécutée par plusieurs processeurs pour traiter des données différentes.
- Plusieurs flots d'instructions, un seul flot de données (*Multiple Instructions stream and Single Data stream* ou MISD): Il y a plusieurs flots d'instructions agissant sur un seul flot de données. Les processeurs partageant une mémoire unique. Une donnée est traitée parallèlement par tous les processeurs qui exécutent chacun une instruction particulière. Ce modèle n'a pas encore eu d'implémentation pratique.
- Plusieurs flots d'instructions, Plusieurs flots de données (*Multiple Instructions stream and Multiple Data stream* ou MIMD): Chaque processeur a ses propres instructions qu'il doit appliquer pour traiter ses propres données. Cette dernière catégorie a l'avantage qu'elle exploite un degré élevé de parallélisme. De ce fait, les machines MIMD sont largement utilisées avec les architectures de haute performance modernes, ces dernières années.

Les systèmes multiprocesseurs de type MIMD peuvent être classifiée en deux familles selon la topologie de la mémoire : La première famille comporte les systèmes multiprocesseurs à mémoire partagée unique. Tandis que la deuxième famille consiste aux systèmes multiprocesseurs à mémoire distribuée.

2.1. *Systèmes multiprocesseurs à mémoire partagée centralisée*

Dans cette première catégorie d'architectures multiprocesseurs, l'ensemble des processeurs est homogène et chacun a sa propre hiérarchie de caches. Comme le décrit la figure 2, ces processeurs se partagent la même mémoire à travers un bus partagé, qui contient les données communes à eux. La distance entre la mémoire centrale et les processeurs est identique. De ce fait, cette famille de systèmes multiprocesseurs est caractérisée par un temps d'accès mémoire uniforme. D'où l'appellation de ces systèmes par UMA (*Uniform Memory Access*). Ces architectures sont aussi dites architectures Multiprocesseurs Symétriques ou SMP (*Symmetric multiprocessors*), il s'agit d'un système dont les processeurs sont tous identiques.

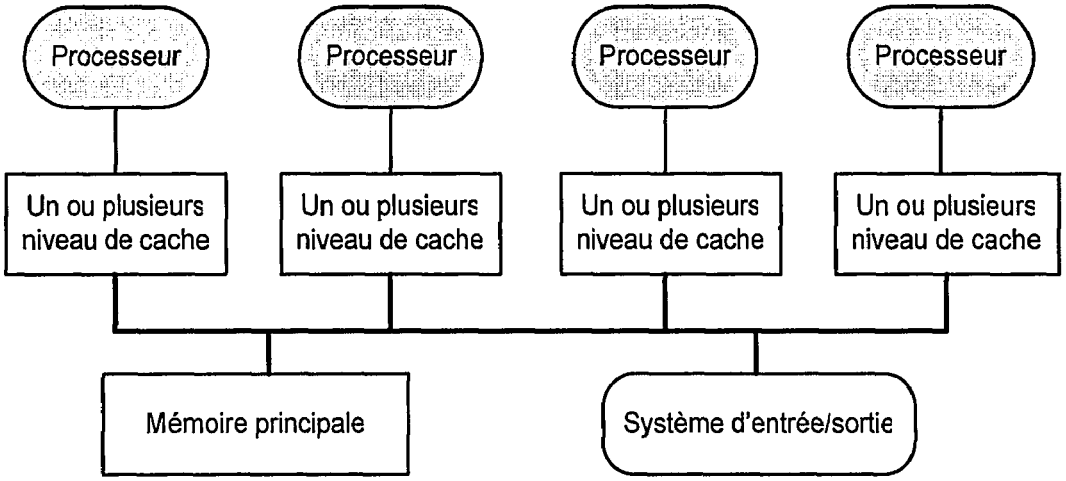


Figure 2. *Systèmes multiprocesseurs à mémoire partagée centralisée*

L'avantage principal des architectures SMP est la simplicité de programmation. En effet, un programme séquentiel peut être porté relativement facilement sur une architecture de ce type. Cette facilité de portabilité permet de réduire le coût de développement des applications parallèles. Par contre, l'inconvénient majeur de ce type de système réside dans la limitation du bus partagé en tant que moyen de communication entres les processeurs et la mémoire. Cette limitation vient du fait que la bande passante du bus est partagée par tous les processeurs. Par suite, elle devient assez limitée dès que le nombre de processeurs commence à augmenter. Pour cette raison, ces systèmes ne sont pas extensibles. Les limites du système SMP ont mit en cause l'apparition du système à mémoire distribuée. Ce sont des systèmes ayant de nombreux processeurs et qui permettent une grande capacité de calcul.

2.2. *Systèmes multiprocesseurs à mémoires privées*

Ces systèmes sont appelés aussi « Systèmes à mémoire distribuée » ou *Distributed memory system*. Il s'agit d'un ensemble de nœuds indépendants. Chaque nœud est formé par un processeur qui possède une mémoire locale (figure 3). La mémoire alors n'est plus unique et globale, mais elle est répartie sur les nœuds. Chaque processeur travaille en parallèle avec les autres processeurs mais sur ses propres données. Cette architecture permet l'utilisation d'un nombre important de processeurs. Elle offre ainsi de meilleures performances vis à vis d'une application donnée. Néanmoins, l'inconvénient majeur de cette architecture est que la communication entre les processeurs est réalisée via un réseau d'interconnexion complexe et qui nécessite une gestion sophistiquée.

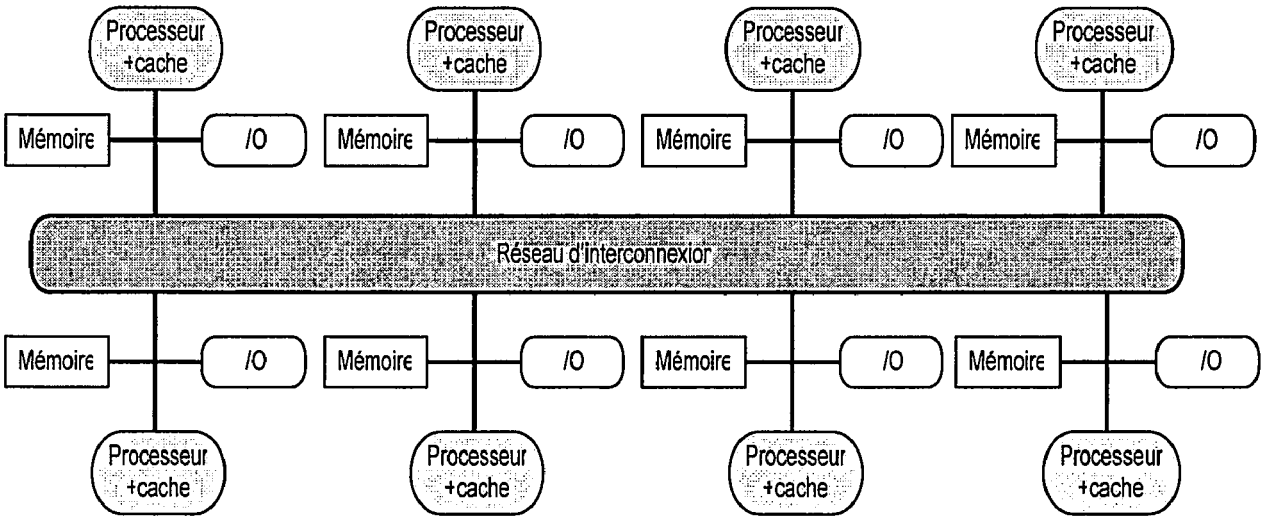


Figure 3. *Systèmes multiprocesseurs à mémoire distribuée*

2.3. *Systèmes multiprocesseurs à mémoires partagées distribuées*

Les architectures à mémoires partagées distribuées nommées encore « *Distributed Shared Memory* » représentent une architecture hybride entre l'architecture à mémoire partagée centralisée et l'architecture à mémoire distribuée. Il s'agit d'un ensemble de nœuds où chaque nœud est composé d'un processeur avec sa mémoire locale. Chaque processeur a un accès à sa propre mémoire, et il peut aussi accéder à celle des autres processeurs. Notons que les accès à des adresses en mémoire locale sont plus rapides que ceux aux mémoires des autres processeurs. Ainsi, le temps d'accès à la mémoire dépend de la localisation de l'information en mémoire. De ce fait, cette architecture est dite NUMA pour (*Non Uniform Memory Access*). Nous parlons alors des mémoires physiquement distribuées et logiquement partagées. En conséquence, cette architecture offre un modèle de programmation proche de celui de l'architecture avec mémoire partagée sans perdre en termes d'extensibilité des processeurs. Malgré l'amélioration, la gestion de la communication des données entre les processeurs avec cette architecture représente son inconvénient principal. À titre d'exemple des systèmes qui incluent les architectures NUMA, nous citons le Silicon Graphics (SGI) 2000 (Laudon *et al.*, 1997) et IBM NUMA-Q (Tom *et al.*, 1996).

Le Tableau 1 récapitule les différences essentielles entre les différents types d'architectures mémoires dans les systèmes multiprocesseurs.

Tableau 1. *Comparaison des architectures à mémoire partagée et à mémoire distribuée*

	Architecture à mémoire partagée	**Architecture à mémoire privée (distribuée)**	**Architecture à mémoire partagée distribuée**
Temps d'accès à la mémoire	Uniforme pour tous les processeurs (UMA)	Non uniforme	Non uniforme
Extensibilité en terme de nombre de processeurs	Petit nombre de processeurs	Grand nombre de processeurs	Grand nombre de processeurs
Modèle de programmation	Facile	Très complexe	Facile
Réseau d'interconnexion utilisé	Bus	Réseau d'interconnexion	Réseau d'interconnexion
Architecture de la mémoire	Grande mémoire physiquement centralisée	Petites mémoires physiquement distribuées	Petites mémoires physiquement distribuées

En conclusion, la faible capacité du bus partagé de supporter le trafic de données de tous les processeurs dans les systèmes multiprocesseurs à mémoire partagée centralisée (SMP) d'une part, et la complexité de la gestion de la communication dans les systèmes multiprocesseurs à mémoires partagées distribuées (NUMA) d'autre part, ont donnée lieu à une nouvelle approche. Cette approche bénéficie de ces deux architectures en réduisant les inconvénients de chacune. Nous parlons ainsi, d'un troisième type des architectures mémoires à savoir. Ce sont les architectures SMP à mémoires partagées multi-bancs, objectif de la prochaine section.

3. Système multiprocesseur à mémoires partagées multi-bancs

Cette nouvelle famille d'architectures de processeurs à mémoires partagées tente à résoudre les limitations des architectures mémoires présentées précédemment. D'une part, les architectures multiprocesseur symétrique (SMP) à mémoire partagée centralisée qui ont l'avantage qu'elles sont plus généralistes, donc portables et plus faciles à exploiter, mais elles sont limitées à un nombre réduit de processeurs. Ainsi, elles sont incapables de

répondre aux besoins des applications en termes de puissance de calcul. D'autre part, les architectures multiprocesseurs à mémoires partagées distribuées qui peuvent fournir une large extensibilité des processeurs grâce à l'utilisation du réseau d'interconnexion. Par contre, elles souffrent des difficultés liées à la gestion complexe de la communication entre processeurs. En effet, cette nouvelle architecture de mémoire combine ces deux architectures en gardant l'aspect SMP mais en remplaçant le bus partagé par un réseau d'interconnexion complexe, qui facilite le passage à l'échelle avec un nombre plus grand de processeurs et qui permet également d'organiser la mémoire selon plusieurs bancs. Cela signifie que la mémoire n'est plus centralisée mais que plusieurs bancs de mémoire sont présents et se partagent l'espace adressable (figure 4). En conséquence, les architectures à mémoires partagées multi-bancs représentent une solution à mi-chemin entre les architectures à mémoire partagée centralisée et celle à mémoire partagée distribuée. En effet, cette architecture permet de diminuer le temps d'accès mémoire et d'augmenter la puissance de calcul avec un coût réduit de conception.

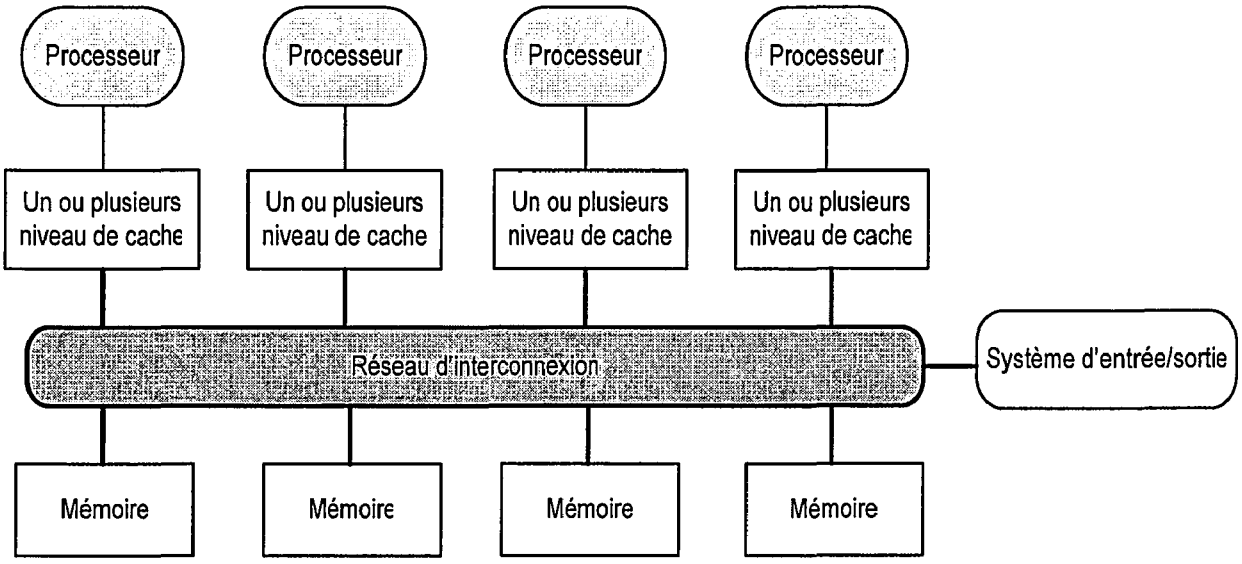


Figure 4. *Architecture SMP à Mémoire Multi-bancs*

4. Les architectures mémoires dans le domaine embarqué

Depuis quelques années, la tendance dans la conception des systèmes embarqués est allée vers la conception des systèmes-sur-puce multiprocesseurs ou *Multi-Processor System-on-Chip* (MPSoC). Ceci s'explique essentiellement pour deux raisons : La première raison est liée à l'émergence des applications modernes gourmandes en puissance de calcul tels que les applications de télécommunication, du multimédia et les outils de transports (automobile

et transport aérien). Pour ces applications, les MPSoCs offrent un temps de conception relativement plus court que les ASIC (*Application Specific Integrated Circuit*) et un rapport performance/consommation plus intéressant que les architectures monoprocesseurs utilisant une fréquence d'horloge très élevée. Enfin, ces derniers permettent une meilleure exploitation des ressources du silicium et une structure modulaire réduisant le temps de conception (*time-to-market*). La deuxième raison est liée aux avancés technologiques qui permettent de profiter du budget en nombre important de transistors. Par suite, il est possible de concevoir des systèmes embarqués de plus en plus complexes.

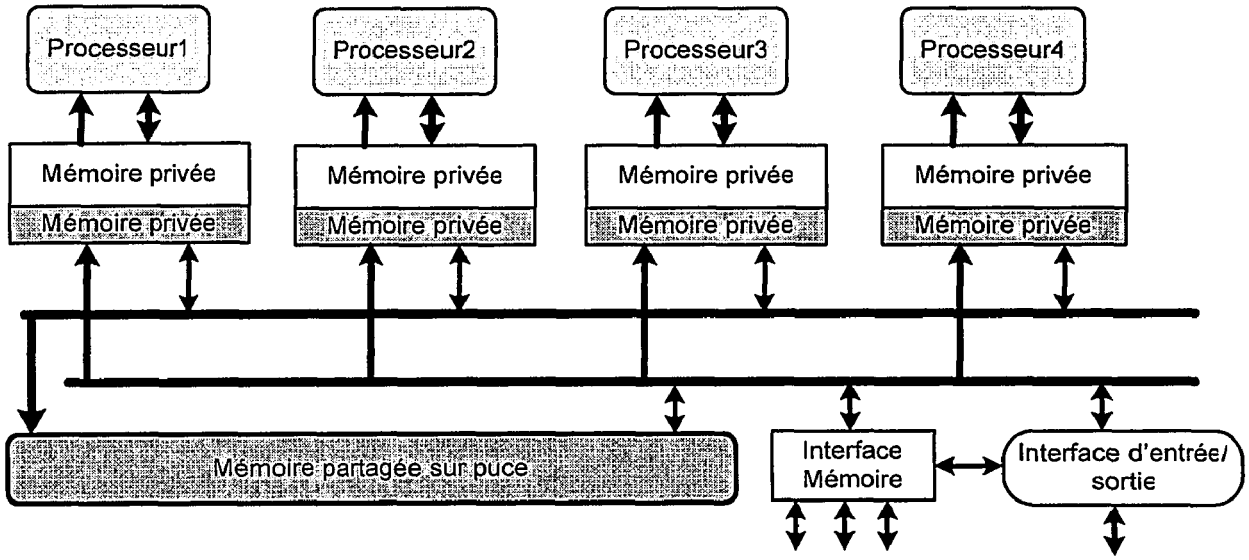


Figure 5. *La structure interne d'un MPSoC*

La figure 5 représente la structure d'un MPSoC, il s'agit d'une puce qui intègre à la fois des processeurs, des unités de calcul spécialisées, des périphériques et une hiérarchie de mémoire. La puce contient également des réseaux-sur-puce (NoCs). Nous distinguons deux types d'architecture MPSoC :

1. Les architectures MPSoCs hétérogènes, qui contiennent différents modèles de processeurs, par suite différents modèles de programmation. De ce fait, l'inconvénient de ces architectures est qu'elles sont difficiles à programmer. Par contre, elles fournissent des performances élevées aux applications pour lesquelles elles sont implémentées.
2. Les architectures MPSoCs homogènes, qui contiennent des processeurs identiques (SMP) et une mémoire partagée. Par suite, ils ont un modèle de programmation

standard. Ainsi, comme ce type d'architecture MPSoC est facile à implémenter, la tendance actuelle s'oriente vers les SMP embarqués.

Dans les travaux de cette thèse, nous nous sommes focalisés sur les architectures à mémoire partagée et utilisant un nombre de processeurs inférieur à quelques dizaines de processeurs. En effet, l'utilisation d'un modèle de programmation à mémoire partagée simplifie le développement des applications sur systèmes multi-cœurs par l'intégration d'un espace d'adressage unique. De cette façon, l'utilisateur n'a pas à orchestrer l'envoie et la réception de message entre les threads. Ce problème est bien souvent la source des erreurs dans les programmes utilisant le principe d'envoi/réception des messages. Cependant, les architectures à mémoire distribuée permettent une extensibilité plus aidée et plus efficace du système. Pour cette raison, plusieurs travaux tentent d'hybrider les deux modèles de programmation afin de profiter des deux avantages (Laudon *et al.*, 1997).

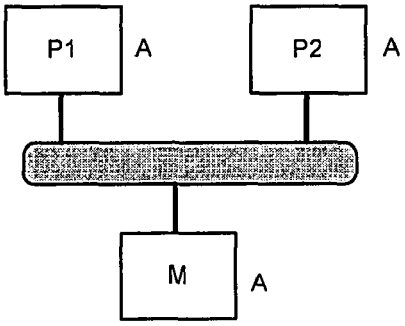
Bien que les architectures de mémoires partagées dans les architectures multiprocesseurs soient diverses, la mise en œuvre d'algorithmes parallèles pour une application donnée, nécessite une gestion sophistiquée des données et de leurs utilisations par les différents processeurs de l'architecture MPSoC. Nous distinguons deux points importants liés à la gestion des données partagées pour ces architectures, qui sont la consistance des données en mémoire et la cohérence des données dans les mémoires caches.

Dans le reste de ce chapitre, nous présentons en premier lieu les problèmes de cohérence des données dans les caches et de consistance des données en mémoire. Ensuite, nous focalisons l'étude sur les solutions proposées dans la littérature pour résoudre le problème de cohérence des données dans les caches.

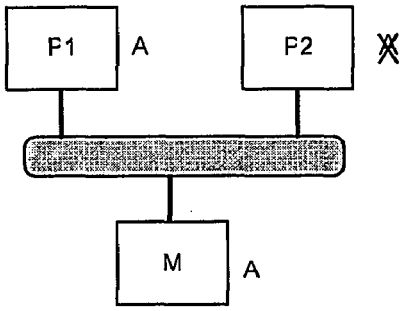
5. Cohérence des données dans les caches

Aujourd'hui, avec l'émergence des applications modernes dans les domaines du multimédia, télécommunication etc., la présence des mémoires caches dans les systèmes multiprocesseurs est indispensable pour l'amélioration des performances. En effet, ces mémoires caches permettent de réduire le temps d'exécution des programmes. Elles réduisent également, les besoins en termes de bande passante sur le réseau d'interconnexion entre les processeurs et les bancs des mémoires partagées. Cependant, dans un système multiprocesseur, le fait de mettre simultanément en mémoire cache des données partagées

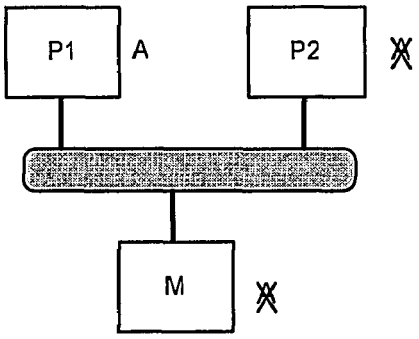
entre plusieurs processeurs, la réalisation d'une opération d'écriture par l'un des processeurs rend la copie de la donnée modifiée non cohérente chez les autres caches. Cette situation est appelée problème de cohérence des données dans les caches (John *et al.*, 2006). À travers la figure 6, nous expliquons ce problème en étudiant le cas d'une architecture multiprocesseur contenant deux processeurs (P1 et P2) et qui se partagent une même mémoire partagée « M ».



a. Les processeurs et la mémoire ont une copie de A



b. P1 écrit dans A avec Ecriture simultanée



c. P1 écrit dans A avec Ecriture différée

Figure 6. *Problème de la cohérence des données dans les caches*

Initialement, la variable désignée par « A » dans cette figure est partagée entre P1 et P2, la mémoire M contient également une copie de A (figure 6.a). En suite, P1 réalise une opération d'écriture dans A. Deux cas peuvent se produire :

- Utiliser la technique d'écriture simultanée (*write-through*). Avec ce mécanisme, toutes les opérations d'écriture en cache se font simultanément dans le cache et dans la mémoire partagée. Par conséquent, cette technique maintient une cohérence de la valeur de A entre le processeur P1 et la mémoire M (figure 6.b). Néanmoins, la valeur de la variable A dans le processeur P2 n'est plus cohérente avec celle dans P1et dans M.

- Utiliser la technique d'écriture différée (ou *write-back*) qui ne met à jour la mémoire partagée que lorsque le bloc est éjecté du cache, ou bien quand un processeur lit le bloc modifié à partir du cache d'un autre processeur. Ainsi, avec cette technique, les copies de A dans P2 et dans la mémoire M ne sont plus cohérentes avec celle de P1 (figure 6.c).

En conséquence, quelque soit la technique d'écriture en cache employée, les copies de A dans le système sont non cohérentes. Pour que ce système de cache devienne cohérent, il faut que le processeur P1 informe le processeur P2 et la mémoire M de l'opération d'écriture qu'il a réalisée dans A. Ceci est l'objectif principal du protocole de cohérence des données dans les caches. Deux protocoles de cohérence des données dans les caches sont habituellement proposés pour cet objectif, le premier applique l'invalidation des données modifiées et le deuxième consiste à les mettre à jour. Nous expliquerons ces deux protocoles dans la suite de ce chapitre.

6. Consistance de mémoire

Bien que le mécanisme de cohérence des données dans les caches permette de garantir que chaque opération d'écriture réalisée par un processeur dans une donnée partagée, soit visible pour un autre processeur utilisant la donnée. Il ne fournit pas aucune information concernant l'instant à partir du quel cette écriture va devenir visible. En effet, il faut bien définir un ordre entre l'opération d'écriture et l'opération de lecture.

Nous considérons l'exemple présenté par la figure 7 (David *et al.*, 1999). Dans cet exemple, l'ordre dans le quel les opérations mémoire seront exécutés influence le fonctionnement du programme. En effet, le processeur P1 modifie une variable « A » qui va être utilisée ensuite par le processeur P2. Afin de respecter cet ordre, une variable partagée appelée « flag » ou drapeau, est utilisée pour la synchronisation. Pour obtenir le résultat attendu de ce programme, il faut que l'écriture dans A devienne visible par P2 avant que l'écriture dans flag soit visible. Dans le cas contraire, le résultat attendu est erroné. Ce problème qui n'est plus impliqué par la cohérence des données dans les caches est appelé problème de consistance des données en mémoire. Par suite, il faut un modèle de programmation spécifique qui garantit que les opérations d'écritures dans les données partagées soient visibles à tous les autres processus dans le même ordre. Autrement, c'est l'ordre dans lequel ces opérations doivent être exécutées. Ainsi, ce modèle permet d'imposer qu'un processeur lit toujours la valeur la plus récente d'une donnée. Ceci, est l'objectif des

modèles de consistance de mémoire.

Figure 7. *Exemple d'un problème de consistance de mémoire*

Dans la littérature, une variété de modèles de consistance de mémoire a été proposée (Adve *et al.*, 1996) (David *et al.*, 1999). Parmi ces modèles, nous citons le modèle de consistance séquentielle ou « *Sequential Consistency* » (Lamport, 1979) de Lamport. Selon Lamport, un système multiprocesseur assure la consistance séquentielle si le résultat de l'exécution de tâches parallèles est identique à une exécution séquentielle de l'ensemble des opérations sur la mémoire et les opérations émises par chaque processeur s'exécutent dans l'ordre prévu. Ceci, est réalisé en se basant sur le principe de l'écriture atomique ou « *Write atomicity* ». L'écriture atomique est assurée en imposant qu'une opération mémoire doit se terminer par rapport aux autres processus avant que la suivante commence. Une opération d'écriture n'est réalisée qu'après avoir reçu les messages d'acquittement venant de tous les processeurs. Par conséquence, aucun processeur ne peut voir la nouvelle valeur jusqu'à qu'elle soit visible pour les autres processeurs. Ce modèle est adopté par le système SGI Origin (Laudon *et al.*, 1997). Bien que le modèle de consistance séquentielle soit facile à implémenter, dans le cas des NoC complexes, ce modèle devient coûteux en termes de matériel et avec extensibilité limitée du système.

Un deuxième modèle de consistance de mémoire est largement exploité, qui est le modèle consistance relâchée « *Release consistency* » (Kourosh *et al.*, 1990). Avec ce modèle, les données partagées sont protégées par des verrous « *locks* ». Un *lock* est géré par deux opérations de synchronisations « *acquire* » et « *release* ». Une donnée partagée ne peut être mise à jour qu'à l'intérieur de ces deux points de synchronisation. Ce modèle utilise le principe de la section critique manipulée par un verrou et délimitée par les primitives « *acquire* » et « *release* ». L'objectif de la primitive « *acquire* » consiste à modifier la donnée partagée localement. De l'autre part, l'objectif de la primitive « *release* » consiste à propager les modifications effectuées sur la donnée au moment de l'opération « *acquire* », vers les autres processeurs. En conséquence, un « *acquire* » ne propage pas immédiatement les mises à jour locales vers les autres processeurs. Par contre, il permet de retarder

l'exécution de leur accès mémoires à une donnée partagée. Une discussion du modèle « *Release consistency* » est effectuée par les auteurs de (Petrot *et al.*, 2006). Comme exemple de système qui emploie le modèle « *Release consistency* » pour la consistance des données en mémoire, nous citons le système DASH (Lenoski *et al.*, 1992).

7. Protocoles de cohérence des données dans les caches

La solution appelée « Sans Cohérence » (Zandvelt, 1998), qui consiste à ne pas mettre dans le cache les données partagées, simplifie l'architecture, puisqu'elle ne s'occupe plus de la gestion de la cohérence des données dans les caches. Par contre, elle charge l'utilisateur d'indiquer au système les données pouvant être mises dans la mémoire cache (ou cachables) qui sont aussi les données privées au processeur et les données partagées ou (non-cachables). En plus de la surcharge qu'elle provoque sur le programmeur, nous avons montré dans un travail ultérieur (Chtioui, 2008) que cette solution ne permet pas une utilisation efficace des mémoires caches et augmente le temps d'exécution et l'énergie consommée. En effet, nous avons comparé cette solution avec deux protocoles de cohérence des données dans les caches qui sont : le protocole par invalidation (noté Inval dans la figure 8) et le protocole par mise à jour (noté Maj dans la figure 8). Nous présenterons ces deux protocoles dans les prochaines sous-sections. Les expérimentations sont réalisées à l'aide de l'application FFT (*Fast Fourier Transform*) parallélisée sur un MPSoC de 4 processeurs. Comme le montre les figures 8 et 9, des gains intéressants en performance et en consommation d'énergie pourraient être obtenus grâce à l'utilisation d'un protocole de cohérence des données dans les caches.

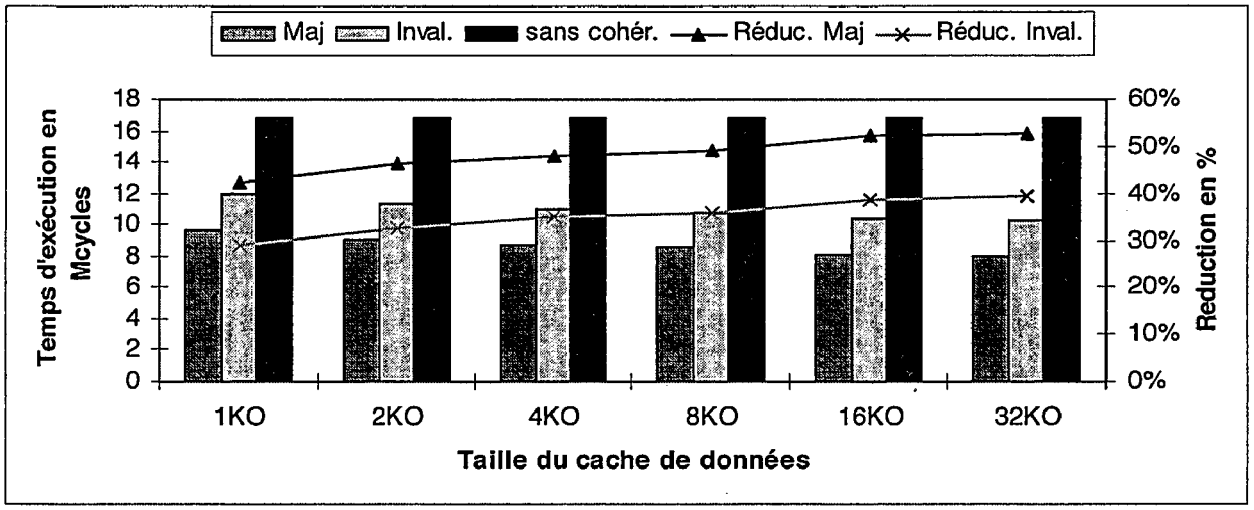


Figure 8. *Temps d'exécution et rapport de réduction en fonction de la taille du cache de données avec l'application FFT pour un MPSoC 4-pocesseurs*

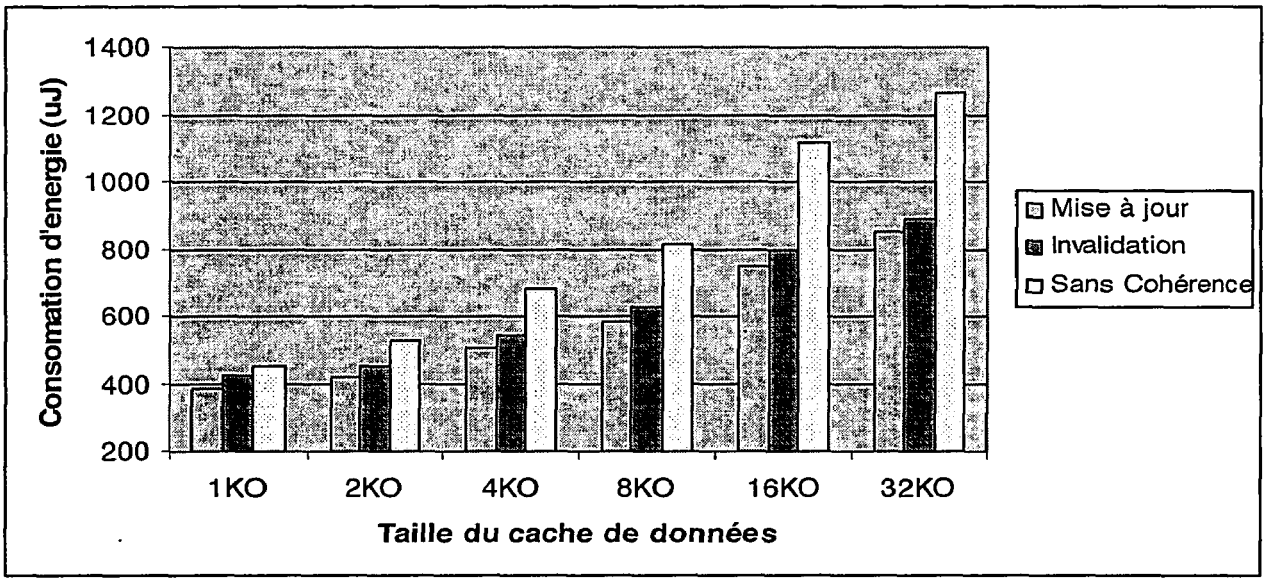


Figure 9. *Consommation de l'énergie en fonction de la taille de cache de données avec l'application FFT pour un MPSoC 4-pocesseurs*

Pour éviter ces inconvénients et profiter des avantages de l'utilisation des mémoires caches, la solution adéquate pour avoir un système de caches cohérent, consiste à rendre chaque opération réalisée localement par un processeur visible aux autres caches. Ceci se réalise en introduisant un protocole de gestion de la cohérence des données dans les caches. Ainsi, le but de ce protocole consiste à garantir qu'une donnée partagée possède la même valeur dans toutes les mémoires caches du système et dans la mémoire partagée. Dans un système multiprocesseur, quand un processeur modifie un bloc de données partagées, il doit informer les autres processeurs qui possèdent ce bloc selon deux protocoles : Protocole par mise à jour des données et protocole par invalidation des données.

Appliquer le protocole par invalidation ou le protocole par mise à jour suite à la modification locale d'un bloc de cache partagé exige la connaissance de quelques informations sur l'état actuel de ce bloc dans les autres caches. Par suite, l'idée de base de ces deux protocoles consiste à garder un ensemble d'informations sur chaque bloc de cache, suite à chaque opération d'écriture ou de lecture réalisée de façon locale ou par un autre processeur sur ce bloc. Dans les deux sous-sections suivantes, nous décrivons le protocole par invalidation et celui par mise à jour, en fournissant des exemples de base (Baslett *et al.*, 1988) (Sweazey *et al.*, 1986) de ces familles de protocoles. Ainsi, nous discuterons pour ces exemples, les états possibles pour un bloc partagé.

7.1. *Protocole par invalidation*

Avec ce protocole, la cohérence des données en caches est maintenue en émettant des messages d'invalidation aux processeurs ayant une copie du bloc modifié. De ce fait, les copies de ce bloc dans les différents caches ne sont plus valides. Par la suite, si de nouveau ce bloc est demandé par l'un des processeurs, il sera chargé depuis la mémoire partagée. Les systèmes multiprocesseurs PowerPC755 de IBM (Motorola) et IA32 Pentium de Intel (Papamarcos *et al.*, 1984) utilisent ce type de protocoles de cohérence des données dans les caches.

7.1.1. *Protocole MSI*

Le protocole MSI est un protocole de cohérence des données dans les caches largement adopté avec les systèmes multiprocesseurs (Baslett *et al.*, 1988). Il s'applique à des caches fonctionnant en *write-back* dans lesquels chaque ligne possède trois états possibles :

- *Modified* (M) : signifie que le bloc a été modifié par le processeur correspondant. Par suite, seul ce processeur possède la copie valide de ce bloc. Par contre, sa copie dans la mémoire partagée est non valide. Le cache doit alors envoyer la nouvelle valeur du bloc vers la mémoire partagée avant de l'éjecter.
- *Shared* (S) : indique que le bloc n'est pas modifié et il existe dans le cache correspondant, et il peut être stocké dans les autres caches. La mémoire partagée possède également une copie valide de ce bloc.
- *Invalid* (I) : signifie que le bloc est invalide dans le cache correspondant, et il doit être cherché à partir de la mémoire partagée ou à partir d'un autre cache.

L'apparition du protocole MSI a donné lieu à un grand nombre d'extensions telles que MESI, MOSI et MOESI (Sweazey *et al.*, 1986).

7.1.2. *Protocoles MESI, MOSI et MOESI*

Les protocoles MESI (Papamarcos *et al.*, 1984), MOSI et MOESI (Sweazey *et al.*, 1986) représentent des extensions du protocole standard MSI. Pour ces objectifs deux nouveaux états sont ajoutés aux états du protocole MSI. Le premier est l'état *Exclusive* (E). Une ligne de cache est à l'état *Exclusive*, si elle contient la donnée la plus récente qui se trouve aussi en mémoire partagée mais dans aucune des autres mémoires caches. Contrairement à l'état *Modified*, un bloc à l'état *Exclusive* peut être éjecté du cache sans mettre à jour la mémoire partagée. Après une opération d'écriture le bloc passe de l'état

Exclusive à l'état *Modified*.

Le deuxième état est noté *Owned* (O). Il indique que le processeur courant avec les autres processeurs possède la valeur la plus récente du bloc. En revanche, la mémoire principale ne la possède pas. Cet état sert à prolonger la durée pendant laquelle la mémoire partagée n'est pas actualisée. Cet état est indispensable puisque l'état *Shared* ne permet pas cette distinction.

7.2. *Protocole par mise à jour*

Avec ce protocole de cohérence des données dans les caches, lorsqu'un processeur écrit dans un bloc partagé, le mot modifié est envoyé vers les caches qui ont une copie de ce bloc. Les copies du bloc modifié sont cohérentes dans toutes les caches. Un large groupe de protocoles de cohérence des données dans les caches qui est apparu et qui est basé sur l'alternative de mise à jour des données partagées.

7.2.1. *Protocole Dragon*

Avec le protocole Dragon (Dragon, 1984), la technique d'écriture utilisée est le *Write-back*. La mémoire partagée n'est mise à jour qu'après l'éjection du bloc de cache. Toute opération d'écriture en cache d'une donnée partagée résulte en une mise à jour de la nouvelle valeur dans les caches qui possèdent une copie de cette donnée. Une ligne de cache peut être dans l'un des 4-états suivants :

- *Exclusive-clean* (E) : le processeur local possède la valeur la plus récente de la ligne de cache. La mémoire principale la possède également.
- *Shared clean* (Sc) : signifie que le processeur local possède avec d'autres processeurs la copie valide de la ligne de cache, la mémoire principale la possède aussi.
- *Shared modified* (Sm) : implique que le processeur local avec d'autres processeurs possède la ligne de cache, par contre la mémoire principale n'est plus actualisée.
- *Modified* (M) : signifie que seulement le processeur local possède une copie valide de la ligne de cache.

En conclusion, lorsqu'une opération d'écriture est déclenchée par un processeur, le protocole de cohérence des données dans les caches se charge pour la réalisation de l'action de cohérence adéquate. Selon cette action, l'état du bloc modifié dans les caches évolue vers

un nouvel état. Ce fonctionnement est réalisé à l'aide d'un ensemble de machines à états finis (FSMs), qui peuvent être implémentées dans les mémoires caches ou dans la mémoire partagée. La manipulation de ces FSMs, ainsi que le contrôle des différents états sont déterminés à l'aide d'un mécanisme de gestion du protocole de cohérence des données dans les caches. Nous présenterons dans la prochaine section une étude bibliographique sur ces mécanismes.

8. Mécanismes de cohérence des données dans les caches

Cette section regroupe les différents mécanismes de cohérence des données dans les caches qui sont adoptés dans les systèmes multiprocesseurs. L'implémentation de ces mécanismes peut être réalisée en matériel, en logiciel ou en combinant les deux approches. En effet, avec une implémentation matérielle, un support matériel dédié à la gestion de la cohérence des données dans les caches est ajouté au système. Contrairement à l'approche logicielle, le maintien de la cohérence des données partagées est à la charge du compilateur, en insérant un ensemble d'instructions dans le code pour cet objectif.

En premier lieu, nous commençons dans cette section par présenter quelques solutions logicielles pour la cohérence des données dans les caches. En suite, nous focalisons sur l'étude des mécanismes matériels. En effet, dans cette thèse nous nous sommes intéressés aux approches matérielles, du fait qu'elles sont plus pertinentes en termes de performance dans le contexte embarqué. D'ailleurs, la solution que nous proposons pour la cohérence des données dans les caches est une solution matérielle. Nous expliquerons ce choix dans le prochain chapitre, dont nous présenterons les points clés liés à sa conception.

8.1. *Solutions logicielles*

Plusieurs approches de cette classe de protocoles de cohérence des données dans les caches ont été développées. L'idée de base de ces protocoles consiste à invalider avec le logiciel toute ligne de cache modifiée par différents processeurs. La solution appelée « *Software-Flush* » (Susan *et al.*, 1989) est un exemple de cette famille de protocoles. Avec cette solution, si un processeur modifie une donnée partagée, elle sera enlevée des autres caches par des instructions placées dans le programme et qui sont appelées « flush instructions ». Toutefois, invalider ou enlever de manière logicielle des lignes de caches peut entraîner un nombre important d'invalidations inutiles. Ce qui est coûteux en termes de temps d'exécution et de consommation d'énergie.

Récemment, les auteurs de (Frank, 2009) ont proposé un protocole de cohérence des données dans les caches dédié aux MPSoCs hétérogènes utilisant des NoCs complexes. Avec ce protocole, les opérations de cohérence sont effectuées statiquement lors de la compilation du programme parallèle. Le programmeur est responsable d'insérer les opérations de cohérence des données dans les caches avec les opérations de synchronisation. Autrement, la cohérence des données dans les caches va être maintenue avec chaque opération de synchronisation. Toujours dans le contexte de l'architecture MPSoC utilisant des NoCs complexes, les auteurs de (Petrot *et al.*, 2006) ont proposé une solution pour la cohérence des données dans les caches orientée logicielle. Leur approche, est basée sur les travaux de (Zandvelt, 1998). Elle consiste à exploiter le fait que le programmeur connaît d'avance quelles sont les données qui vont être partagées entre les processeurs et quelles sont les données qui vont être privées aux processeurs. En se basant sur cette distinction, les données partagées entre les processeurs vont être des régions de mémoire non-cachables, tandis que les régions de mémoire privées sont cachables. L'apport de (Petrot *et al.*, 2006) consiste à améliorer la solution (Zandvelt, 1998), en simplifiant la manière employée pour séparer les données en des régions privées et autres partagées. Ceci, est effectué en introduisant dans le programme de nouvelles primitives dédiées à cet objectif. Malgré l'amélioration, ce protocole ne permet pas une bonne exploitation des mémoires caches. Ce qui résulte en une baisse des performances.

Les faiblesses des approches logicielles pour la cohérence des données dans les caches ont entraîné l'émergence d'un nombre important de solutions matérielles, objectif de la prochaine sous-section.

8.2. *Mécanismes matériels*

L'implémentation matérielle des mécanismes de cohérence des données dans les caches est largement adoptée par les systèmes multiprocesseurs (Anant *et al.*, 1995) (Tom *et al.*, 1996). Deux mécanismes fondamentaux sont généralement utilisés avec les protocoles de cohérence des données dans les caches, qui sont le mécanisme d'espionnage « *Snoop protocols* » (John *et al.*, 2006) et le mécanisme du répertoire « *Directory protocols* » (John *et al.*, 2006).

8.2.1. *Mécanisme d'espionnage « Snoop Protocols »*

Avec ce mécanisme, les processeurs avec la mémoire partagée sont connectés avec un

bus commun. Toute opération d'écriture ou de lecture d'un bloc mémoire réalisée par un processeur doit être diffusée à travers le bus. Les autres processeurs sont autorisés à écouter les transactions réalisées sur le bus pour détecter cette opération. Autrement, ils espionnent le bus et ceux qui ont une copie de ce bloc selon le protocole adopté, ils réalisent l'action de cohérence adéquate et changent l'état du bloc dans leurs caches.

En plus de son modèle simple et facile à implémenter, le mécanisme d'espionnage de bus lors d'un échec de cache, offre une faible latence. En effet, lorsqu'un processeur diffuse une requête de lecture d'un bloc, le processeur qui a le bloc demandé se charge de lui répondre directement (*cache-to-cache miss*). Ceci, introduit un temps de latence moins important que celui introduit par la recherche du bloc directement à partir de la mémoire partagée. Cet aspect est plus efficace avec les applications qui utilisent une quantité de données partagées importante. En revanche, diffuser la requête à tous les processeurs, exige l'augmentation du trafic de données dans le bus et nécessite une large bande passante. Ainsi, la bande passante doit être proportionnelle au carré du nombre de processeurs. Par contre, pour des contraintes physiques, l'architecture du bus avec la bande passante limitée ne permet pas d'avoir des architectures multiprocesseurs extensibles. Afin de dépasser ces limitations, plusieurs solutions sont proposées (Bilir *et al.*, 1999), (Martin *et al.*, 2003) et (Sorin *et al.*, 2002). Ces solutions ont le même objectif qui consiste à diminuer le besoin du mécanisme d'espionnage en bande passante et améliorer le rapport latence/bande passante. Ceci, en utilisant le principe « *Destination-set prediction* » (Bilir *et al.*, 1999) appelé aussi « *Predictive multicast* ». Cette approche consiste à prévoir les processeurs qui peuvent répondre au processeur demandeur du bloc, et par suite envoyer la requête de lecture seulement à ces processeurs. D'autres approches sont aussi apparues et qui consistent à augmenter la bande passante du bus en utilisant plusieurs bus fonctionnant simultanément. Toutefois, ces solutions sont limitées. En effet, ces bus partagent les mêmes ressources et la même mémoire, ce qui entraîne un délai important pour référencer la mémoire.

Malgré l'amélioration, avec les architectures multiprocesseurs largement extensibles telles que les architectures à mémoires partagées distribuées (DSM), le mécanisme d'espionnage reste inefficace. En effet, ces architectures permettent d'avoir un nombre important de processeurs en utilisant un réseau d'interconnexion complexe. Or, la diffusion des demandes vers toutes les mémoires caches du système génère un trafic de données important dans le réseau. De ce fait, les mécanismes d'espionnage de bus sont limités aux architectures multiprocesseurs utilisant un nombre réduit de processeurs. La solution

adéquate pour ces architectures est le mécanisme du répertoire. C'est un mécanisme qui élimine le principe de diffusion des requêtes, en envoyant seulement la requête au répertoire qui a des informations sur chaque bloc mémoire.

8.2.2. *Mécanisme du répertoire « Directory protocols »*

Ce mécanisme résout le problème de cohérence des données dans les caches en utilisant un répertoire partagé entre tous les processeurs (Tang, 1976) (Censier *et al.*, 1978). Ce répertoire garde pour chaque bloc de mémoire un historique sur son dernier accès par les processeurs. Ceci est réalisé en stockant différentes informations concernant son état dans les différentes mémoires caches. Ces états sont tels que les états des protocoles MSI, MOSI et MESI décrits précédemment. L'ensemble des informations stockées dans le répertoire permet de déterminer l'action de cohérence à réaliser suite à un accès mémoire particulier. Contrairement au mécanisme d'espionnage, avec ce mécanisme la requête d'un processeur est envoyée uniquement au répertoire et seulement les processeurs qui ont une copie du bloc demandé, reçoivent des messages de cohérence. De ce fait, le trafic de données dans le réseau d'interconnexion diminue et par suite le besoin en termes de bande passante diminue également.

Dès sa première apparition avec (Tang, 1976), plusieurs types de protocoles de cohérence des données dans les caches basés sur le mécanisme du répertoire sont apparus. Le besoin de ces protocoles en termes de performances temporelles et énergétiques a permis l'évolution de la conception du répertoire. Nous distinguons deux grandes catégories de répertoires : la catégorie « *Bit-Map schemes* » et la catégorie « *Linked lists scheme* ».

8.2.2.1. Catégorie « *Bit-Map Schemes* »

Parmi les mécanismes de répertoire les plus connus de cette famille, nous citons le mécanisme « *Full bit vector scheme* » (Censier *et al.*, 1978). Ce mécanisme consiste à associer *n bits* pour chaque bloc mémoire, où *n* représente *1 bit* pour la mémoire partagée et *n-1 bits* pour les *n-1* mémoires caches dans le système. Chaque bit fait rappel à la situation du bloc dans la mémoire partagée et dans les caches. Si par exemple un cache contient la valeur actualisée du bloc mémoire, alors le bit correspondant à ce cache est mis à 1, sinon le bit est mis à 0. L'avantage de cette technique est le fait que seulement les caches qui ont une copie du bloc modifié reçoivent des messages de cohérence. Par contre, l'inconvénient de cette technique est le coût matériel nécessaire pour le stockage des états des blocs mémoires. Ainsi, plus le nombre de caches et des blocs mémoire utilisés par le système augmente, plus

la taille du répertoire augmente et par suite le coût matériel augmente aussi et les performances se dégradent. De ce fait, la technique « *Full bit vector* » est limitée de point de vue extensibilité. Comme solution pour ce problème, la technique « *Limited directory schemes* » (Agarwal *et al.*, 1988) est proposée. Elle considère que généralement, un nombre limité de processeurs partagent à la fois un bloc mémoire (moins de quatre processeurs). Par la suite, cette approche fixe la taille du répertoire pour un nombre limité de processeurs. Lorsque le nombre de processeurs qui se partagent le bloc en même temps dépasse la taille fixée du répertoire, plusieurs mécanismes sont proposés pour augmenter dynamiquement cette taille. Les auteurs de (Chaiken et al., 1991) et (Chaiken et al., 1993) ont proposé le schéma « *LimitLESS directory scheme* ». Avec ce schéma, la taille du répertoire est gérée par un nombre limité de pointeurs matériels. Cependant, avec des applications dans les quelles le nombre de processeurs partageant un bloc dépasse le nombre de pointeurs, alors ce nombre de pointeurs est étendu en logiciel. Bien que cette approche réduise les coûts matériels en faisant appel à la conception logicielle, elle dégrade les performances temporelles.

8.2.2.2. Catégorie « *Linked lists scheme* »

Cette deuxième catégorie de répertoire vise à avoir des systèmes largement extensibles. Par suite plus performants mais aussi en réduisant les coûts liés au stockage des états des blocs. Cette catégorie est appelée « *Linked lists scheme* » (IEEE, 1993). Elle utilise un répertoire distribué entre la mémoire principale et les différents caches. Avec cette technique, chaque bloc de cache garde un pointeur sur le dernier cache qui lui a demandé une copie de ce bloc. Ceci est réalisé sous forme d'une liste chaînée. Le standard IEEE *Scalable Coherent Interface* (SCI) (IEEE, 1993), le protocole *Stanford's Distributed-Directory* (Thapar *et al.*, 1993), (Thapar *et al.*, 1990) et le *Scalable Tree Protocol* (STP) (Nilsson *et al.*, 1992) utilisent cette approche. En effet, cette approche permet une très bonne extensibilité du système avec un coût matériel minimum. Par contre, son inconvénient majeur réside dans sa complexité d'implémentation liée au maintien de la liste chaînée. Ajoutant aussi, que le temps mis pour consulter cette liste et invalider les caches contenant une copie du bloc modifié, peut entraîner la dégradation des performances.

Finalement, dans le but de profiter des avantages des deux catégories de répertoire que nous avons présentés, une alternative hybride qui combine ces deux catégories est proposée dans (Yeimkuan *et al.*, 1999).

À la lumière de cette étude, le répertoire est couramment employé dans des projets académiques et industriels, le tableau 2 (Alexander, 2003) résume ces travaux. Dans ce tableau, le cluster signifie un nœud contenant plusieurs processeurs.

Tableau 2. *Cohérence des données dans les caches dans des systèmes multiprocesseurs expérimentaux et commerciaux*

Nom	Intra-cluster	Inter-cluster	Organisation du répertoire
DASH (Lenoski *et al.*, 1992)-(Daniel *et al.*, 1992)	Espionnage	Répertoire	Full bit vector
Alewife (Anant *et al.*, 1995)	Non-clustered	Répertoire	Software extended
FLASH (Jeffrey *et al.*, 1994)	Non-clustered	Répertoire	Dynamic pointer allocation in memory
NUMAchine (Grindley *et al.*, 2000)	Répertoire	Répertoire	Limited pointer
SG Origin (Laudon *et al.*, 1997)	Répertoire	Répertoire	Full bit vector, coarse vector for large systems
Compaq AlphaServer GS320 (Kourosh *et al.*, 2000)	Répertoire	Répertoire	Full bit vector
Sun Fire 15k (Alan, 2000)	Espionnage	Répertoire	Full bit vector
HP SPP2000 (X-class) (Radhika *et al.*, 1997)	Répertoire	SCI	Linked list
HP Superdome (HP, 2002)	Répertoire	Répertoire	Full bit vector
IBM NUMA-Q (Tom *et al.*, 1996)	Espionnage	SCI	Linked list

8.2.3. *Mécanisme hybride Espionnage/Répertoire*

Afin de profiter du fait que le mécanisme d'espionnage de bus n'exige pas le passage par le répertoire après chaque opération d'écriture ou de lecture d'un bloc mémoire d'une

part, et du fait que le mécanisme du répertoire est plus extensible et n'exige pas une large bande passante d'autre part, une approche hybride qui tire profit de ces deux mécanismes est proposée. En effet, les auteurs de (Milo *et al.*, 2002) ont proposé un protocole hybride qui change le fonctionnement d'un mécanisme adaptatif entre le principe de l'espionnage et celui du répertoire. Ce changement d'un mécanisme à un autre est réalisé selon la variation de la bande passante du système au cours de l'exécution. Ainsi, l'idée de base consiste à utiliser le principe de l'espionnage quand la bande passante est importante et le répertoire quand la bande passante est limitée. Par suite, ce protocole hybride s'adapte dynamiquement à la bande passante disponible au cours du temps d'exécution et choisit le mécanisme adéquat avec elle. Dans ce contexte, plusieurs travaux sont apparus et qui visent fournir un protocole hybride Espionnage/Répertoire intelligent. Parmi ces travaux, nous citons (Diana *et al.*, 2003) et (Chun *et al.*, 2007).

9. Comparaison protocole par invalidation et protocole par mise à jour

Comme évoqué précédemment, deux stratégies sont généralement adoptées pour assurer la cohérence d'une ligne de cache dans un MPSoC à mémoire partagée. La première stratégie consiste à mettre à jour le bloc modifié chez les autres processeurs et ceci en émettant la nouvelle valeur du bloc modifié aux caches contenant une copie de ce bloc. Dans ce cas, le bloc modifié maintient toujours son état valide. Le bloc peut donc être réutilisé lors d'un futur accès. Ce protocole permet généralement une diminution du nombre d'échecs de cache et par la suite une réduction du temps d'exécution. Néanmoins, lorsque les données modifiées par un processeur sont faiblement utilisées par les autres processeurs, les mises à jour deviennent inutiles. Ce qui entraîne l'augmentation du temps d'exécution et la consommation d'énergie.

La deuxième stratégie consiste à invalider les copies du bloc modifié dans les autres caches. Avec ce deuxième protocole, dès qu'une opération d'écriture est réalisée sur un bloc, le bloc devient invalide dans les autres caches. Par la suite, si de nouveau ce bloc est demandé par un processeur, il sera chargé depuis la mémoire partagée. Par conséquent, ce protocole peut donner lieu à un nombre d'échecs en cache relativement plus grand, en particulier dans le cas où un nombre important de processeurs se partagent une grande quantité de données en lectures/écritures. Contrairement au protocole avec mise à jour, le protocole avec invalidation est relativement performant lorsque les données écrites sont faiblement utilisées par les processeurs. Ce deuxième protocole peut réduire la

consommation d'énergie, puisqu'il n'y a pas de transfert de données pour réaliser les mises à jour.

D'après la discussion précédente, nous pouvons conclure que les deux protocoles se complètent puisque les points forts de l'un correspondent aux points faibles de l'autre. Ceci à mis en cause l'apparition d'un troisième protocole de cohérence des données dans les caches hybride (Invalidation/ Mise à jour) qui exploite les avantages de ces deux protocoles. Dans la prochaine section, nous présenterons les principaux protocoles de cohérence des données dans les caches hybrides qui sont apparus.

10. Protocoles de cohérence des données dans les caches hybrides (Invalidation/Mise à jour)

Les limites en terme de performances qui découlent de l'utilisation du même protocole de cohérence des données dans les caches (invalidation ou mise à jour) pour les différents modèles d'accès aux données partagées d'une application, ont causé l'apparition de plusieurs protocoles de cohérence des données dans les caches hybrides. Tous ces protocoles ont un objectif commun qui consiste à réduire le nombre d'échecs de cache et des transactions nécessaire pour maintenir la cohérence des données dans le réseau (Jhalani *et al.*, 2007) (Farnaz *et al.*, 1995). Nous classifions ces protocoles en deux familles : La première famille, appelée « Protocoles hybrides dynamiques » ou encore « *On-line hybrid protocols* », dans laquelle la sélection du protocole est faite dynamiquement au cours du temps d'exécution. À l'opposé, la deuxième famille appelée « Protocoles hybrides statiques » ou encore « *Off-line hybrid protocols* », dans laquelle la sélection du protocole est faite statiquement avant de commencer l'exécution de l'application.

10.1. *Protocoles hybrides dynamiques « On-line »*

Le principe de cette famille de protocoles hybrides est basé sur l'analyse des accès précédents aux blocs de mémoire. Selon cette analyse, le mécanisme de cohérence des données dans les caches décide au cours du temps d'exécution quel protocole va être utilisé. Les protocoles de cette famille sont aussi appelés « Protocoles de cohérence des données dans les caches adaptatifs » ou encore « *Adaptive hybrid protocols* » (Jhalani *et al.*, 2007). Plusieurs protocoles adaptatifs sont apparus, le protocole « *Write-once* » (Goodman, 1983) est un exemple de cette famille. Avec ce protocole, la première écriture dans un bloc de cache entraîne une mise à jour de la mémoire principale et une invalidation de la copie de ce

bloc dans les autres caches. L'opération d'écriture suivante dans ce bloc par le même processeur entraîne la modification locale du bloc. Par contre, la mémoire principale n'est plus mise à jour. L'extension de ce protocole a permis l'appariation du protocole « *Archibald scheme* » (Archibald *et al.*, 1986) (Archibald, 1988), qui consiste à élever le nombre des opérations de mise à jour de la mémoire principale.

Le « *Competitive scheme* » est un autre exemple de protocole hybride de cette famille. C'est un protocole largement utilisé avec les mécanismes d'espionnage de bus (Karlin *et al.*, 1986) et du répertoire (Grahn *et al.*, 1995). Avec cette approche, un compteur est associé à chaque bloc de cache. Ce compteur est soumis initialement à une valeur appelée « Seuil des mises à jour » ou encore « *Competitive threshold* ». Le protocole de cohérence des données dans les caches commence par réaliser les opérations de mise à jour des données. Chaque opération de mise à jour entraîne la décrémentation de la valeur du compteur chez le cache du processeur qui a déclenché l'opération d'écriture. Une fois, cette valeur atteint zéro, le protocole de cohérence des données dans les caches change au mode d'invalidation. Cette approche reste moins efficace avec les applications contenant des données partagées caractérisées par des accès exclusifs en écriture et en lecture par différents processeurs. Ce type de données partagées est appelé « *Migratory shared data* ». Les auteurs de (Cox *et al.*, 1993) ont montré que pour ce type de données partagées, le protocole par invalidation est plus adéquat que celui par mise à jour des données. Ils ont proposé aussi un mécanisme qui permet de les détecter dynamiquement.

Avec l'approche « *Competitive scheme* », il y a un risque que les « *Migratory shared data* » soient mises à jour à cause du « *Competitive threshold* ». Ce qui va entraîner un trafic de données inutile. Pour résoudre ce problème, les auteurs de (Grahn *et al.*, 1996) ont proposé l'intégration du mécanisme de détection des « *Migratory shared data* » proposé par les auteurs de (Cox *et al.*, 1993) dans le mécanisme du « *Competitive scheme* ». Récemment, dans ce même contexte, les auteurs de (Jhalani *et al.*, 2007) ont proposé un protocole adaptatif basé sur le mécanisme du répertoire qui détecte les modèles d'accès aux données « *Migratory shared data* ». Cependant, des résultats expérimentaux ont montré que l'utilisation d'une même valeur de seuil pour tous les blocs de cache n'est pas une solution pertinente puisque chaque bloc de cache a un modèle d'accès propre à lui. Pour cela, Anderson et Karlin (Anderson *et al.*, 1996) ont proposé de varier la valeur du seuil d'un bloc à un autre. Ceci, en mesurant pour chaque bloc de cache, le nombre des opérations d'écritures consécutives réalisées par le même processeur sans avoir des accès par les autres

processeurs. En effet, plus ce nombre est important, plus le protocole par invalidation est efficace, donc la valeur du seuil des mises à jour doit être faible. Si autrement ce nombre est faible, le protocole adéquat est celui par mise à jour et par suite la valeur du seuil doit être importante pour réaliser le maximum des mises à jour.

Dans ce même contexte, les auteurs de (Liqun *et al.*, 2007) ont développé un protocole de cohérence des données dans les caches adaptatifs qui détecte dynamiquement les données partagées de type « Producteur-consommateur ». Cette classe de données partagées existe généralement dans les traitements de type producteur/consommateur. Il s'agit d'un traitement dont lequel les données écrites par un processeur sont utilisées par les autres processeurs. Pour cette classe de données partagées, le protocole par mise à jour fournit les meilleures performances. Par contre, dans le cas où les données écrites par un processeur ne sont plus utilisées par les autres processeurs, le protocole de cohérence des données dans les caches change du mode de mise à jour au mode d'invalidation. Récemment, les auteurs de (Abdullah *et al.*, 2010) ont proposé aussi un protocole hybride qui détecte dynamiquement les données de type « Producteur-consommateur ».

Bien qu'ils soient variés, les protocoles hybrides adaptatifs ont quelques limitations. En effet, généralement ces protocoles ont besoin de rassembler plusieurs informations concernant chaque bloc de cache. Ces informations sont regroupées à partir des différents processeurs. En conséquence, un trafic de données important est généré dans le réseau d'interconnexion (Grahn *et al.*, 1996). En outre, ces protocoles exigent une quantité de ressources matérielles importante et coûteuse en termes de consommation d'énergie. En termes de temps d'exécution, par rapport à l'utilisation du protocole simple (invalidation ou mise à jour), l'amélioration n'est pas assez importante. Ceci est dû principalement à l'incompatibilité entre les besoins des applications qui changent le comportement d'une application à une autre et dans l'application même au cours du temps d'exécution, ceci d'une part et les besoins de l'architecture de l'autre part.

10.2. *Protocoles hybrides statiques « Off-line »*

Avec cette deuxième famille de protocoles hybrides, le choix du protocole approprié est réalisé statiquement. Autrement, avant de commencer l'exécution de l'application, les opérations de cohérence sont insérées par le programmeur pendant la phase de conception, ou par le compilateur pendant le processus de compilation. Par suite, la décision d'utiliser le protocole par invalidation ou celui par mise à jour est réalisée en tenant compte non

seulement des accès précédents au bloc de cache, mais en se basant aussi sur des informations obtenues à partir du code source au cours de la compilation de l'application. Cette deuxième option offre la possibilité de prédire un profilage des futurs modèles d'accès aux données partagées.

(Farnaz *et al.*, 1995) ont présenté une stratégie matérielle logicielle, qui exploite l'aspect prédictive du compilateur afin de sélectionner le protocole adéquat avec une opération d'écriture. Les auteurs dans (Dahlgren, 1998) ont proposé une nouvelle approche appelée « *Hardware prefetching scheme* ». Cette approche vise la réduction de la pénalité associée aux accès mémoire en faisant recours à une combinaison de trois approches : La première approche appelée « *Adaptive sequential prefetching scheme* » est proposée par (Dahlgren *et al.*, 1995), consiste à prévoir en se basant sur la localité spatiale les futurs échecs de cache et à amener d'avance les blocs mémoire. Ce qui permet de réduire le rapport d'échec de cache. Cette dernière est combinée avec le protocole hybride « *Competitive scheme* » et avec l'approche de détection des « *Migratory sharing data* ».

En conséquence, cette famille de protocoles hybrides prévoit les futurs modèles d'accès afin de déterminer le meilleur protocole. Pour cet objectif, son principe consiste à analyser statiquement un nombre important d'informations sur le comportement dynamique des données, ce qui est difficile à obtenir. De ce fait, la technique de protocole hybride « *off-line* » peut être complémentaire et combinée avec la technique de protocole hybride « *on-line* ».

À travers le tableau 3, nous présentons une comparaison entre quelques protocoles de cohérence des données dans les caches hybrides existants et le protocole que nous proposons au cours de cette thèse. Dans ce tableau, le terme « Coh. Support » désigne le support matériel de cohérence des données dans les caches utilisé par la technique. Ce support peut être l'un des deux mécanismes (Espionnage ou Répertoire). De même, le terme « Adaptatif » réfère à la capacité du protocole à s'adapter aux patterns d'accès des données partagées, au cours du temps d'exécution.

Tableau 3. *Comparaison des protocoles de cohérence des données dans les caches hybrides*

Protocole	Adaptatif	Coût matériel	On-line/ Off-line
Write-once (Goodman, 1983)	Non	Coh.support	On-line
Archibald (Archibald *et al.*, 1986)	Non	Coh.support	On-line
Competitive-scheme (Karlin *et al.*, 1986)	Non	Coh.support+ 1 compteur par ligne de cache	On-line
Two adaptive protocols (Anderson *et al.*, 1996)	Oui	Mécanisme d'espionnage + (2 compteurs + 1 bit) par ligne de cache + 3 à 5 bits par bloc de mémoire	On-line
Migratory Detection mechanism (Grahn *et al.*, 1996)	Non	Support du schéma compétitive+ (1 bit + 1 pointeur (log2 N bits) pour N cache) par ligne de cache	On-line
Combined hardware-software strategy (Farnaz *et al.*, 1995)	Non	Coh. support	Off-line
Adaptive sequential prefeteching (Dahlgren *et al.*, 1995)	Non	Coh. Support+2 bits (prefetch, zero-bit) par ligne de cache+ 3 compteurs (4 bits) par cache	Off-line
Protocole proposé	Oui	Mécanisme du Répertoire + (2 compteurs+1 bit) par bloc de mémoire	On-line

11. Conclusion

Dans ce chapitre, nous avons mis l'accent sur un problème très important dans les architectures MPSoC à mémoire partagée, et à qui ces architectures doivent faire face. C'est le problème de cohérence des données dans les caches. Ainsi pour avoir un système de mémoires caches cohérent il faut que l'écriture réalisée par un processeur dans un bloc partagé soit visible aux autres processeurs. Pour résoudre ce problème, plusieurs protocoles de cohérence des données dans les caches sont apparus dont nous avons regroupé les plus essentiels.

Malgré la variété, les protocoles de cohérence des données dans les caches existants

possèdent quelques limites qui peuvent influencer sur les performances des systèmes multiprocesseurs et plus précisément les MPSoCs. Parmi ces limites, ces protocoles ne prennent pas en compte le comportement de l'application parallèle et par suite les modèles d'accès aux blocs mémoire. Ajoutant aussi une autre faiblesse, qui consiste au fait que ces protocoles sont conçus pour des architectures multiprocesseurs hautes performances, qui ne sont pas exigeantes en termes de contraintes temporelles et énergétiques. En conséquence, dans le contexte des MPSoCs, ces solutions sont moins efficaces. De ce fait, dans cette thèse notre contribution consiste à proposer un nouveau protocole pour la cohérence des données dans les caches. Il s'agit d'un protocole hybride invalidation/mise à jour qui tire profit des avantages de ces deux protocoles et qui permet de choisir le protocole de gestion de la cohérence le plus adapté pour chaque donnée mémoire, au cours du temps d'exécution. Ce dernier est conçu spécialement pour les MPSoCs à mémoire partagée. En effet, il consomme une quantité relativement faible de temps et d'énergie et exige une quantité relativement faible de ressources additionnelles.

Nous présenterons dans le prochain chapitre les points clés liés à la conception de ce protocole de cohérence de cache hybride (invalidation et mise à jour).

Chapitre 3

Un protocole hybride et adaptatif pour la cohérence des données dans les caches dans les MPSoCs

1. Introduction

Dans le chapitre précédent, l'accent a été mis sur le problème de la cohérence de données dans les caches pour les systèmes multiprocesseurs à mémoire partagée. En effet, une variété de solutions est proposée dans la littérature pour résoudre ce problème. Cependant, ces solutions souffrent de deux faiblesses : d'une part, la majorité de ces solutions sont implémentées dans des systèmes multiprocesseurs haute-performances. Or, ce type de systèmes, jusqu'à une date récente, ne prenait pas en compte les contraintes liées aux ressources hardwares et à la consommation d'énergie comme c'est le cas avec les systèmes embarqués. D'autre part, ces solutions ne prennent pas en compte les schémas dynamiques d'accès aux données mémoires. En effet, notre étude a démontré que les motifs d'accès aux données sont différents d'une application à une autre et peuvent varier pendant l'exécution d'une même application. De ce fait, les architectures embarquées existantes, sont impertinentes. Ainsi, le développement de nouvelles solutions pour le maintien de la cohérence des données dans les caches dans les MPSoCs s'avère nécessaire. Dans ce cadre, l'objectif principal de cette thèse consiste à développer une nouvelle approche pour la cohérence des données dans les caches dédiée aux MPSoCs utilisant un réseau d'interconnexion complexe. Ainsi, l'objectif de notre travail est double. Il faut proposer un mécanisme et une architecture capable de :

1. S'adapter aux schémas d'accès aux données réalisés par l'application en choisissant le protocole de cohérence le plus performant en termes de vitesse d'exécution et de consommation d'énergie. Cette adaptabilité du protocole doit être à la fois dynamique et automatique. Autrement dit, l'adaptabilité du protocole est obtenue sans intervention ni de l'utilisateur ni du programmeur ou du compilateur.
2. Pouvoir s'appliquer facilement et efficacement sur les nouvelles architectures de réseaux d'interconnexions embarquées (nommées ici NoC pour network-on-chip). En effet, ces architectures sont très attractives du fait qu'elles sont extensibles et elles permettent l'intégration d'un nombre important de processeurs. L'architecture à bus partagé, largement utilisée dans la littérature, arrive rapidement à la saturation dès que le nombre de processeurs dépasse la dizaine de processeurs. La cohérence des données dans les caches dans les MPSoCs utilisant un autre moyen de communication que le bus partagé, reste encore un problème ouvert.

De ce fait, nous présentons dans ce chapitre un nouveau protocole de cohérence des données dans les caches dédié à ces systèmes et dont les caractéristiques sont les suivantes :

- Basé sur le mécanisme du répertoire distribué sur plusieurs banc-mémoires.
- Hybride puisqu'il combine deux protocoles de cohérence des données dans les caches : *invalidation* et *mise à jour*.
- Dynamique, la sélection du protocole de gestion le plus approprié aux motifs d'accès courants, se fait automatiquement et en cours de l'exécution.
- Implémentation matérielle simple et peu coûteuse en ressources.
- Utilisation du mécanisme avec écriture en cache et mémoire simultanée. Ce procédé est nommé *write-through* en anglais.

Ce chapitre est organisé comme suit : Nous expliquons dans la section 2 les raisons des choix que nous venons de donner. La section 3 détaille l'implémentation du protocole proposé. Les sections 4 et 5 décrivent respectivement le fonctionnement du protocole avec une opération d'écriture, ensuite avec une opération de lecture réalisée par un processeur. Cette description est faite à l'aide des machines à états finis (FSM). Enfin, dans la section 6 nous détaillons à l'aide d'un algorithme, le principe de fonctionnement du protocole proposé. Dans cette section l'accent sera mis sur la méthodologie de sélection du protocole adéquat (invalidation ou mise à jour) selon le motif d'accès aux données.

2. Caractéristiques du protocole proposé

Lors de la conception d'un protocole de cohérence des données dans les caches, il y a plusieurs considérations à prendre en compte. Dans cette thèse et pour les raisons présentées précédemment, ce sont les systèmes embarqués multiprocesseurs (MPSoCs) à mémoire partagée utilisant des NoCs complexes qui sont visés. Ces systèmes sont dédiés aux applications embarquées qui exigent des contraintes strictes de performance et de consommation d'énergie. En conséquence, un choix judicieux du protocole de cohérence des données dans les caches doit permettre d'améliorer les performances et répondre aux besoins de ces applications.

Les différents schémas des protocoles de cohérence des données dans les caches peuvent être classifiés selon les points suivants:

- Le support utilisé pour la cohérence, généralement, il peut être soit le mécanisme d'espionnage soit celui du répertoire.

- La technique d'écriture peut être simultanée *Write through* ou différée *write-back.*
- Le protocole appliqué, il peut être par mise à jour (*update*), par invalidation ou une combinaison des deux.
- La décision de l'opération de cohérence peut être dynamique (au cours du temps d'exécution) ou statique (au cours du temps de compilation).
- L'implémentation du protocole de cohérence peut être matérielle ou logicielle.

Par la suite, toutes ces classifications interviennent dans les critères de choix du protocole de cohérence des données dans les caches. Dans cette section, nous présentons les critères de choix que nous avons fixés afin d'obtenir un protocole de cohérence des données dans les caches efficace.

Tableau 4. *Critères de choix des protocoles de cohérence des données dans les caches*

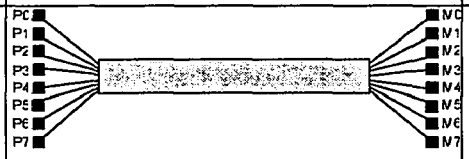
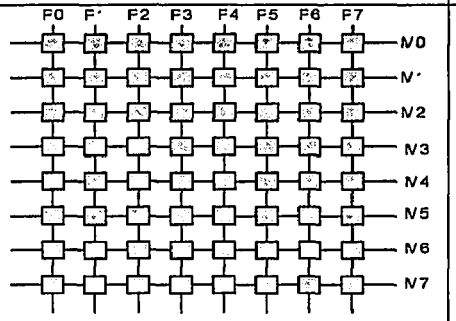
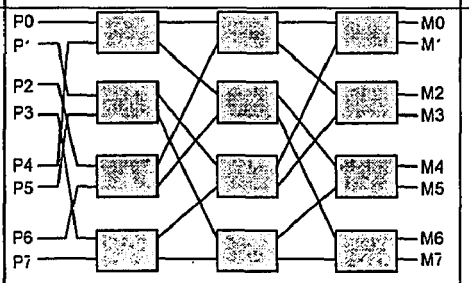
Critères	Solutions Possibles	
Réseau d'interconnexion	Bus partagé	Autre que le bus* (NoC complexe)
Mécanisme de gestion	Espionnage	Répertoire Centralisé ou Distribué*
Technique d'écriture en cache	Simultanée*	Différée
Le protocole de cohérence	Simple (Invalidation ou Mise à jour)	Hybride (Invalidation et Mise à jour) *
Moment de sélection du protocole	Au cours du temps d'exécution*	Au cours du temps de compilation
Implémentation	Logicielle	Matérielle*

Dans le tableau 4, la solution marquée d'une étoile (*) est celle qui a été choisie dans notre travail. Dans la suite, nous allons discuter l'ensemble de ces options et expliquer les raisons qui nous ont poussé à choisir telle ou telle solution.

Comme déjà dit dans l'introduction, nous avons choisi la conception d'un nouveau protocole de cohérence des données dans les caches pour les MPSoCs utilisant des NoCs

complexes. Ce choix s'explique du fait que ce type de réseau offre une bonne extensibilité du système en termes de processeurs. Ce qui se traduit en un niveau de performances très élevé. Le tableau 5 compare le bus partagé avec deux types de NoCs complexes qui sont le crossbar et les réseaux multi-étages (MIN) pour *Multistage Interconnetion Network*. Dans ce tableau, les « Pi » désignent les processeurs et les « Mi » désignent les mémoires. D'après ce tableau, nous constatons que les réseaux multi-étages représentent une solution intermédiaire entre le crossbar et le bus partagé. En effet, ces derniers permettent un niveau de performance très élevé en comparaison avec le bus et une complexité de conception réduite par rapport au crossbar.

Tableau 5. *Quelques exemples de réseaux d'interconnexion utilisés dans les MPSoCs*

Type de réseau	Architecture interne	Extensibilité	Modèle de conception	Performance
Bus partagé	 Plusieurs sources qui partagent le même ensemble des fils	Faible nombre de processeurs	Facile à mettre en œuvre	Faible bande passante
Crossbar	 Permet une liaison directe entre les sources et les destinateurs	Grand nombre de processeurs	Complexe en termes de points de connexions	Bande passante très élevée
Réseaux multi-étages (MIN)	 Les réseaux multi-étages type Oméga	Grand nombre de processeurs Facilement extensibles	Complexité de conception réduite	Bande passante importante

2.1. *Protocole basé sur le mécanisme du répertoire*

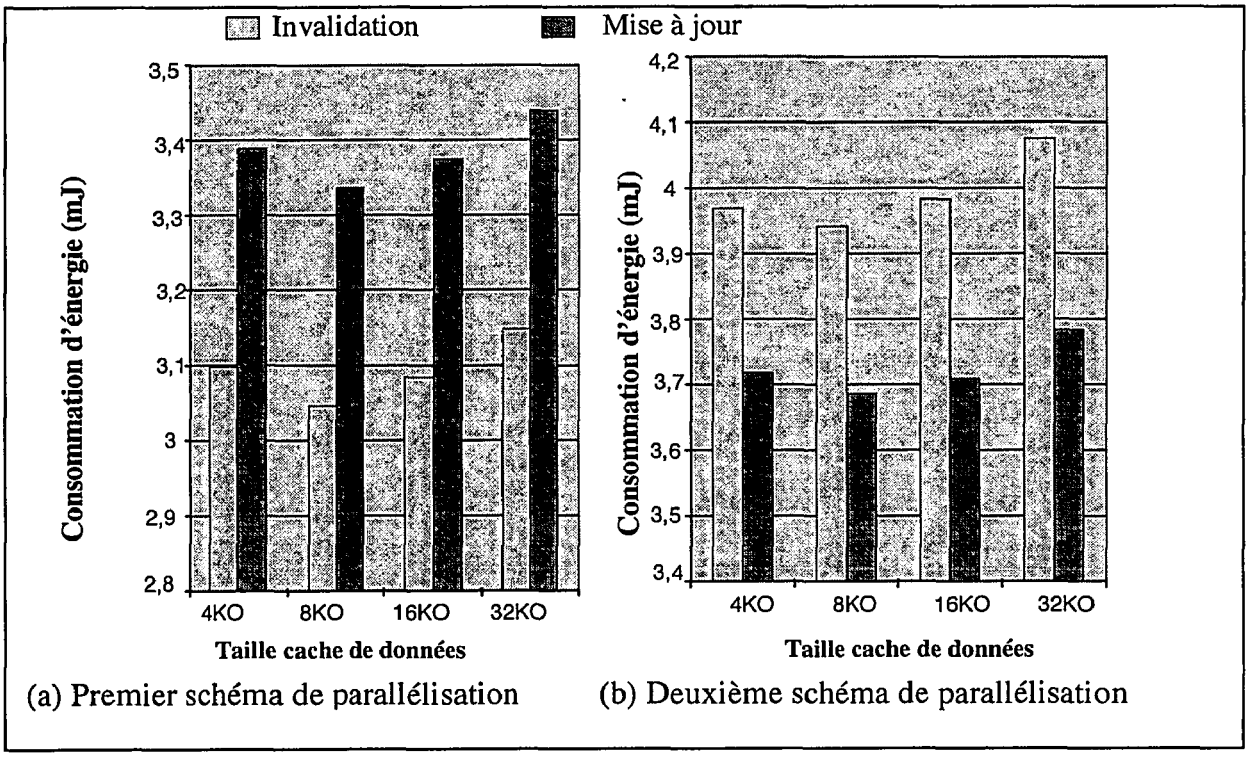
Habituellement, le partage des données dans les MPSoCs est effectué en utilisant un bus commun et en autorisant les processeurs à écouter les transactions réalisées sur le bus pour mettre à jour ou invalider les données partagées et éviter la non-cohérence des données. Malheureusement, cette solution n'est pas applicable au cas des MPSoCs utilisant d'autres types de réseaux d'interconnexion. Les architectures MPSoCs que nous visons dans ce travail sont des architectures utilisant d'autres NoCs que le bus partagé. Dans le domaine des architectures hautes performances, ce problème a donné lieu à un nouveau mécanisme qui est le mécanisme du répertoire (Censier *et al.*, 1978). La flexibilité de ce mécanisme en termes de nombre de processeurs la rend très attractive. Ceci est le cas pour nous aussi. Ainsi, nous adoptons ce mécanisme dans notre travail.

2.2. *Protocole Hybride Invalidation/Mise à jour*

En se basant sur le mécanisme du répertoire, deux méthodes sont habituellement proposées pour résoudre le problème de cohérence des données dans les caches. La première applique l'invalidation des données modifiées et la deuxième consiste à les mettre à jour. Cependant, l'inconvénient de ces deux méthodes est qu'elles ne prennent pas en compte les motifs des accès mémoires réalisés par les applications. Ceci est d'ailleurs confirmé par des expérimentations que nous avons réalisées dans notre papier (Chtioui, 2008). Ainsi, nous avons montré que selon la méthode utilisée pour paralléliser l'application Transformée Rapide de Fourier (ou FFT), l'un ou l'autre des deux protocoles donnera de meilleures performances en termes de temps d'exécution et de consommation d'énergie. Plus précisément, nous avons parallélisé cette application selon deux schémas. Notons d'abord que, afin de distinguer ces deux schémas, nous avons utilisé la classification des données partagées proposée par Gupta et Weber (Anoop *et al.*, 1992) (Wolf-Dietrich *et al.*, 1989). Il s'agit de deux types de données partagées qui sont: *Mostly-read shared data* et *Frequently read-written shared data*. Cette classification sera détaillée dans le chapitre 5. Ainsi, les deux schémas sont les suivants :

1. Dans ce premier schéma de parallélisation, chacun des processeurs travaille principalement sur ses données. Il y a ainsi très peu de partage de données en écriture. Les données partagées sont alors de type *Mostly-read shared data*.
2. Dans ce deuxième schéma de parallélisation, les données partagées modifiées par un processeur sont couramment utilisées par les autres processeurs. Les

données partagées sont alors de type *Frequently read-written shared data.*



(a) Premier schéma de parallélisation (b) Deuxième schéma de parallélisation

Figure 10. *Consommation d'énergie en fonction de la taille de cache pour deux versions de parallélisation de la FFT avec un MPSoC de 4 processeurs*

La figure 10 présente une comparaison entre le protocole par invalidation et le protocole par mise à jour, en termes de consommation d'énergie totale du système et en fonction de la taille de cache de données, pour un MPSoC contenant 4 processeurs. La figure 10.a, fournit le résultat de comparaison pour le premier schéma de parallélisation. Nous constatons de cette courbe que la consommation d'énergie avec le protocole par invalidation est moins élevée que celle avec le protocole par mise à jour. Ceci vient du fait que ce dernier réalise beaucoup de mises à jour inutiles avec ce schéma de parallélisation. Par contre, le résultat avec le deuxième schéma de parallélisation représenté par la figure 10.b, prouve que le protocole par invalidation augmente la consommation d'énergie par rapport au protocole par mise à jour. Ceci revient à l'augmentation du nombre d'échecs de cache dit à l'invalidation des données partagées.

Puisqu'une application parallèle peut présenter les deux aspects, l'utilisation d'un protocole hybride (invalidation et mise à jour) qui exploite les avantages de ces deux protocoles, devient un de nos objectifs principaux. En outre, ce choix s'explique aussi du fait que malgré que le protocole hybride fût l'objet de plusieurs travaux de recherche, très

peu de ces travaux ont été dédiés aux architectures MPSoCs.

2.3. *Protocole dynamique*

À chaque avancée de la technologie, les applications embarquées évoluent et deviennent plus variées et peuvent contenir plusieurs modèles d'accès aux données partagées. Ainsi, il devient crucial que le protocole de cohérence des données dans les caches soit capable d'adapter ses actions avec chaque modèle d'accès aux données de chacune des applications. Par conséquent, il est nécessaire d'utiliser un protocole de gestion de la cohérence des données dans les caches qui soit adaptable dynamiquement. De ce fait, parmi les objectifs de cette thèse, il y a le développement d'un mécanisme qui manipule dynamiquement le protocole de cohérence des données dans les caches selon le comportement variable de l'application parallèle. Autrement dit, le protocole proposé dans ce travail est capable de capter le changement d'un modèle d'accès dans l'application en cours d'exécution à un autre et de déterminer pour chaque modèle d'accès le protocole adéquat.

2.4. *Implémentation matérielle*

Le protocole de gestion de la cohérence des données en cache peut être implémenté sous trois formes : par un circuit matériel (ou hardware), par un programme s'exécutant sur un processeur (solution logicielle ou software) ou enfin par une combinaison des deux solutions précédentes.

L'implémentation matérielle consiste à ajouter au système un support matériel dédié à détecter et analyser les accès à la mémoire cache et mettre en place le protocole de cohérence adéquat. À l'inverse de la solution hardware, avec l'approche logicielle, le maintien de la cohérence est basé sur l'utilisation d'un compilateur qui doit générer des instructions de gestion de la cohérence des données dans les caches. Ceci est réalisé en insérant dans le programme parallèle des instructions dédiées à la cohérence. En plus de la complexité liée à la production d'un tel code dans les tâches parallèles, l'analyse du compilateur, obligatoirement statique, ne permet pas de modifier le protocole de gestion de la cohérence et d'adapter ce dernier en fonction des motifs d'accès aux données des applications parallèles. Pour toutes ces raisons, l'implémentation matérielle est plus pertinente pour le protocole de cohérence des données dans les caches que nous proposons.

2.5. *Protocole à écriture Simultanée*

La façon avec la quelle sont réalisées les opérations d'écriture mémoire, joue un rôle important dans le choix du protocole. Cette gestion des opérations peut être faite selon deux techniques (John *et al.*, 2006):

1. Écriture simultanée ou immédiate : toute opération d'écriture est réalisée en même temps dans le cache et dans la mémoire principale. Cette technique maintient une cohérence entre les données de la mémoire partagée et celles du cache. Par contre, elle est lente puisque avec chaque accès en cache il faut un accès à la mémoire.
2. Écriture différée : toute opération d'écriture est réalisée seulement dans le cache. Lorsqu'un bloc est supprimé du cache, il doit d'abord être copié du cache vers la mémoire. Pour cet objectif, un bit est retenu par bloc de cache pour savoir s'il a été modifié. Cette technique limite les écritures en mémoire, par suite elle réduit le trafic vers la mémoire. Par contre, elle exige une utilisation des mécanismes complexes pour maintenir la cohérence entre le cache et la mémoire partagée. Le tableau 6 présente une comparaison entre ces deux techniques d'écriture en cache.

Tableau 6. *Comparaison des techniques d'écriture en cache*

Technique d'écriture	Performances	Gestion du cache
Écriture simultanée	• Grande latence mémoire • Nécessite une bande passante élevée	Simple
Écriture différée	• Faible latence mémoire • Réduit le besoin en bande passante	Complexe

Vu que le bus partagé à une bande passante limitée, la plus part des architectures MPSoCs à bus partagé utilisent la technique d'écriture différée. Cependant, avec l'apparition des réseaux sur puce (NoCs) complexes, le problème de bande passante n'est plus critique. De ce fait, pour les MPSoCs utilisant des NoCs complexes, la technique d'écriture simultanée est plus intéressante. En effet, cette technique simplifie la manipulation et le contrôle de la cohérence des données. Ceci explique notre choix de la technique d'écriture simultanée dans ce travail. Notons aussi qu'en cas de défaut de cache

sur une écriture, le bloc référencé n'est pas chargé dans le cache. Cette technique est nommée *write-no-allocate*. Autrement dit, lorsqu'un échec de cache en écriture se produit, alors l'opération d'écriture est réalisée uniquement dans la mémoire principale.

3. Implémentation du protocole hybride

En se basant sur les critères de choix que nous avons précisés au dessus, nous avons implémenté un nouveau protocole hybride pour la cohérence des données dans les caches. Ce protocole est capable de détecter au cours du temps d'exécution les changements dans les motifs de partage de données entre les processeurs. Il est donc capable d'appliquer le mécanisme de gestion de la cohérence adéquat avec chaque modèle de partage. Comme les performances du protocole de cohérence sont liées à l'architecture du répertoire, la première étape dans ce travail consiste à présenter la structure du répertoire adoptée. Les différents états du répertoire seront aussi présentés.

Dans cette thèse, nous visons des architectures MPSoCs contenant quelques dizaines de processeurs. En effet, les processeurs que nous avons testés ont une certaine complexité matérielle (comme les processeurs ARM Cortex A7) qui ne permet de disposer de plus d'une centaine de processeurs par puce.

3.1. *Structure du répertoire*

Dans la littérature, plusieurs structures de répertoire pour la gestion de la cohérence des données dans les architectures multiprocesseurs ont été développées. Dans ce travail, nous adoptons la structure *full bit vector* (Censier *et al.*, 1978). Il s'agit d'associer pour chaque bloc de mémoire un vecteur de bit, où chaque bit fait rappel à l'état de ce bloc dans chaque cache et dans la mémoire centrale partagée. Dans notre cas, nous n'avons pas besoin de savoir l'état du bloc mémoire dans la mémoire partagée, puisqu'elle est toujours actualisée grâce à la technique d'écriture simultanée. La figure 11 représente la structure du répertoire que nous avons utilisée. Dans cette figure, nous supposons que l'architecture MPSoC contient « n » mémoires caches et que la mémoire partagée contient « m » blocs de mémoire. La notation « Ci » correspond au numéro du cache « i » où $(0 \leq i < n)$. La structure *full bit vector* est très intéressante du fait qu'elle est simple à implémenter et facile à manipuler.

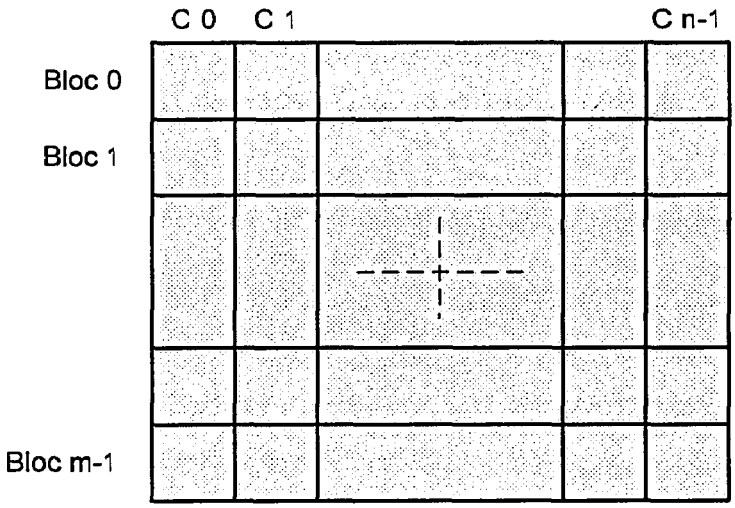


Figure 11. *Structure du répertoire dans un MPSoC*

3.2. *Protocole ESIO*

Après avoir introduit la structure du répertoire, nous définissons un protocole de gestion de la cohérence à quatre états appelé « ESIO » pour (*Exclusive*, *Shared*, *Invalid* et *Invalidated by others*). Le protocole que nous proposons manipule l'ensemble des états possibles pour un bloc de mémoire donné. En effet, chaque entrée du répertoire représente l'état du bloc mémoire correspondant dans les différents caches. Selon l'action de cohérence réalisée sur ce bloc, son état dans le répertoire évolue d'un état vers l'autre.

Avec le protocole ESIO, un bloc de mémoire peut avoir quatre états différents : *Exclusive* (E), *Shared* (S), *Invalid* (I) et *Invalidated by others* (O). La signification des états (E), (S) et (I) ressemble à celle des états (E), (S) et (I) des protocoles existant déjà comme MSI (Baslett *et al.*, 1988) et MESI (Papamarcos *et al.*, 1984). Concernant l'état (O), c'est un nouvel état que nous proposons dans ce travail. Cet état est différent de l'état *Owned* du protocole MOESI (Sweazey *et al.*, 1986). Nous expliquons dans un premier temps chacun de ces quatre états. Dans la section suivante, nous détaillons le fonctionnement du protocole ESIO à l'aide des machines d'états finis (FSM). Rappelons que l'état E, S, I ou O est stocké dans le répertoire.

3.2.1. *État Exclusive (E)*

L'état exclusif ou « *Exclusive* » noté (E) pour un bloc de cache est utilisé pour signaler que la version actualisée du bloc existe uniquement dans ce processeur et dans la mémoire partagée.

3.2.2. *État Shared (S)*

L'état partagé ou « *Shared* » noté (S) pour un bloc de cache est utilisé pour signaler que la valeur la plus récente du bloc existe dans le cache correspondant, dans les autres caches d'autres processeurs et dans la mémoire partagée.

3.2.3. *État Invalid (I)*

L'état non valide ou invalide « *Invalid* » et noté (I) indique que la copie du bloc n'est pas valide dans le cache du processeur correspondant à cette entrée du répertoire. Autrement dit, ce bloc soit qu'il n'a pas été encore chargé par ce processeur, ou bien il a été chargé mais qu'il a été remplacé par un autre bloc lors d'un échec de cache.

3.2.4. *État Invalidated by others (O)*

Le protocole que nous introduisons dans ce travail, contient un nouveau état appelé « *Invalidated by others* » et noté (O) qui est un état particulier de l'état « *Invalid* ». Cet état signifie que ce bloc mémoire n'est pas valide dans le processeur correspondant à cette entrée du répertoire, parce qu'il a subit une invalidation suite à un message d'invalidation venant d'un autre processeur.

Ce nouvel état joue un rôle important dans le protocole hybride que nous proposons dans cette thèse. En effet, il permet de distinguer les blocs mémoire qui ne sont pas encore chargés ou bien qui sont éjectés de la mémoire cache, des blocs mémoire qui ont été chargés en cache mais invalidés par d'autres processeurs. Cet état participe au choix du protocole approprié, invalidation ou mise à jour.

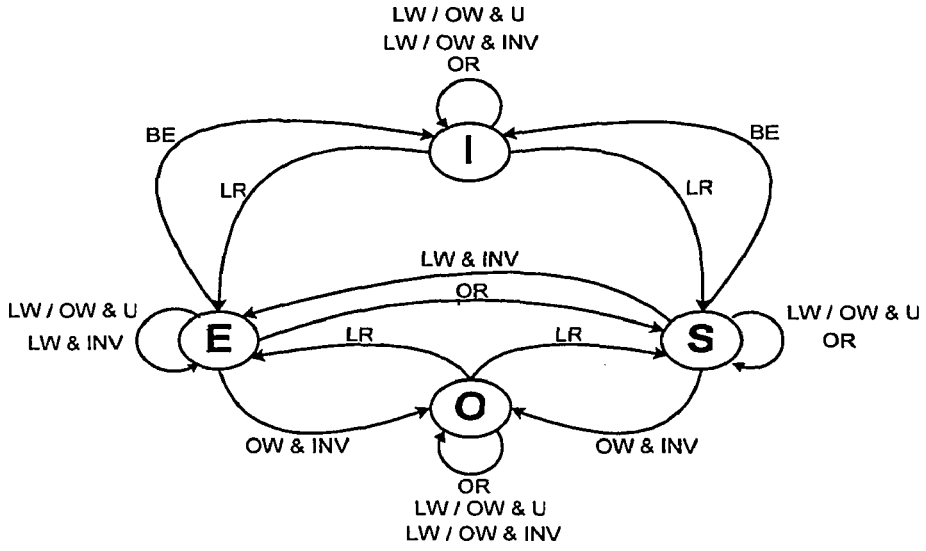
4. Fonctionnement du protocole ESIO avec une opération d'écriture

Lorsqu'un processeur réalise une opération d'écriture dans un bloc de cache, deux étapes essentielles doivent être réalisées par le protocole hybride. Ainsi, la première étape consiste à maintenir la cohérence des copies du bloc modifié dans les autres processeurs. Ceci se réalise soit par invalidation, soit par mise à jour de ces copies. L'application de tel ou tel protocole implique un changement des états du bloc dans les autres caches et dans le répertoire.

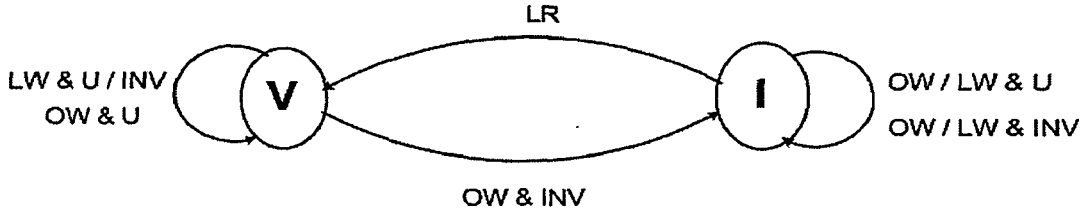
La deuxième étape consiste à modifier, selon le protocole utilisé (par invalidation ou par mise à jour), les états des différentes copies du bloc modifié dans le répertoire et également dans les autres caches. Pour cet objectif, nous avons implémenté deux machines d'états finis (FSM). La première machine est implémentée dans le répertoire, et elle est

gérée par le contrôleur du répertoire. Tandis que la deuxième FSM est implémentée dans la mémoire cache et elle est gérée ainsi par son contrôleur. La figure 12 représente la FSM du protocole hybride dans le répertoire et celle dans la mémoire cache. Dans cette figure, la notation « Ox/Ly & z » sur l'arc correspond soit à une opération de type « x » réalisée par un autre processeur, soit à une opération de type « y » réalisée localement par le processeur lui-même. La variable « z » indique le type du protocole employé.

Ici, « x » et « y » peuvent être soit une lecture (R pour *Read*) ou écriture (W pour *Write*). La variable « z », peut être soit une mise à jour (U pour *Update*) soit une invalidation (INV). L'évolution des différents états de ces FSMs suite à une opération d'écriture déclenchée par l'un des processeurs, dépend de la nature du protocole à utiliser (invalidation ou mise à jour).



(a) FSM du protocole hybride dans le répertoire



(b) FSM du protocole hybride dans le cache

Figure 12. *FSM du protocole hybride*

4.1. *Action à réaliser suite à une écriture dans le protocole par invalidation*

Lorsqu'un processeur déclenche une opération d'écriture d'un mot, deux cas peuvent se produire : Le premier cas est un succès de cache en écriture et dans ce cas le processeur modifie localement le mot mémoire. Ainsi, l'état du bloc dans la mémoire cache garde l'état Valide (V) (figure 12.b). Dans le deuxième cas, il s'agit d'un échec de cache en écriture. Dans ce cas le bloc du cache garde l'état Invalide (I). Dans les deux cas, le processeur envoie une requête d'écriture à la mémoire partagée pour la mettre à jour et envoie aussi un paquet d'invalidation vers le répertoire. Le répertoire sur réception du paquet, vérifie les états du bloc modifié chez le processeur qui a déclenché l'opération d'écriture. Pour ce processeur cette opération est dite (LW) pour *Local Write*. Le répertoire vérifie également l'état du bloc chez les autres processeurs. Pour ces processeurs, cette opération est appelée (OW) pour (*Other Write*). En conséquence, le répertoire réagit selon ces deux types d'opérations :

- ***Local Write* (LW)**: Cette opération concerne le processeur qui a déclenché l'opération d'écriture. Pour ce dernier avec une opération (LW), s'il est à l'état (O) ou bien à l'état (I) dans le répertoire, alors le même état est gardé. Différemment, s'il est à l'état (S) dans le répertoire, alors il passe à l'état (E). S'il est déjà à l'état (E), il garde ce même état (figure 12.a).
- ***Other Write* (OW)**: Cette opération concerne les autres processeurs qui se partagent le bloc modifié. Ainsi avec une opération (OW), si l'un de ces processeurs possède le bloc à l'état (E) ou bien à l'état (S) dans le répertoire, alors cet état passe à l'état *Invalidated by others* (O) (figure 12.a). Dans son mémoire cache l'état du bloc modifié passe de l'état (V) à l'état (I) (figure 12.b). Si autrement, le bloc modifié est déjà à l'état (I) ou à l'état (O) dans le répertoire pour l'un des processeurs, alors une opération (OW) ne modifie rien.

Parallèlement à la figure 12 nous simplifions l'explication du protocole ESIO à l'aide de la figure 13. Cette figure décrit le changement des états dans le répertoire avec le protocole par invalidation pour un bloc mémoire (*bloc j*) pour une architecture contenant 4 processeurs et 4 caches (C0, C1, C2 et C 3).

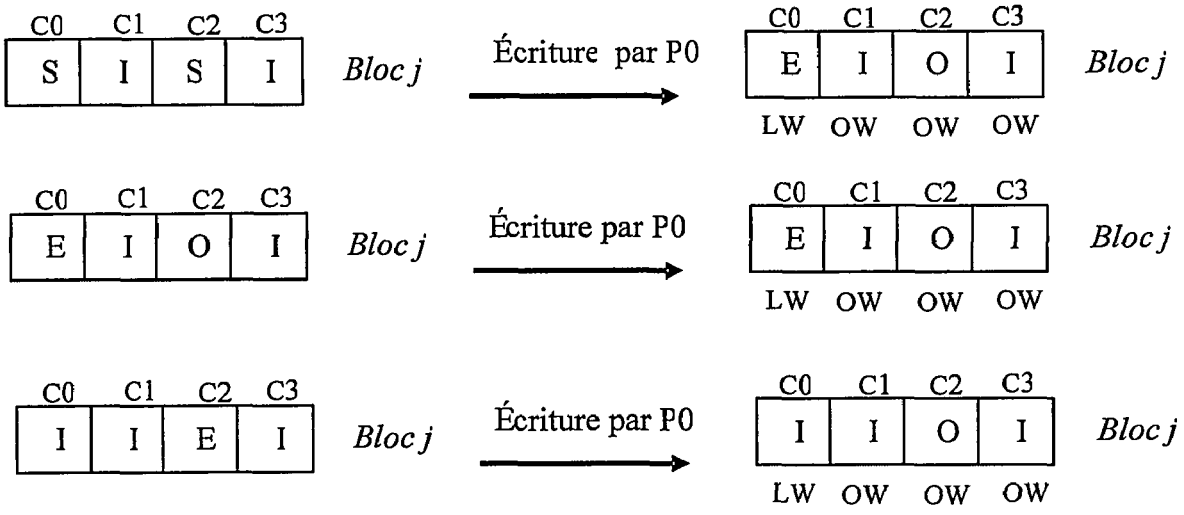


Figure 13. *Changement des états dans le répertoire suite à une écriture réalisée par le processeur 0 (P0) dans un bloc mémoire (Blocj) avec le protocole par invalidation*

4.2. *Action à réaliser suite à une écriture dans le protocole mise à jour*

Avec le protocole par mise à jour ou *update* (U), lorsqu'un processeur modifie un mot dans un bloc de cache, une demande d'écriture est envoyée vers la mémoire partagée. En même temps, une requête de mise à jour est envoyée vers le répertoire. Dans le protocole par invalidation, le répertoire réagit selon deux types d'opérations (figure 14):

- ***Local Write* (LW)**: Puisque cette opération concerne le processeur qui a déclenché l'opération d'écriture, le répertoire vérifie l'état du bloc chez ce processeur. Si ce dernier possède le bloc à l'état *Shared* (S) dans le répertoire, une copie du mot modifié est envoyée vers les autres caches des autres processeurs. A l'opposé, si le bloc à mettre-à-jour est à l'état (E), alors il garde ce même état, mais cette fois aucune requête de mise à jour n'est envoyée aux autres processeurs. Dans le cas où le bloc est à l'état (O) ou à l'état (I) dans le répertoire pour ce processeur (figure 12.a), c'est-à-dire il est à l'état invalide (I) dans son cache (figure 12.b), alors l'état est gardé le même dans le répertoire et dans le cache.
- ***Other Write* (OW)**: Pour les autres processeurs qui se partagent le bloc modifié, l'opération d'écriture réalisée est une opération (OW). Ainsi avec cette opération si l'un de ces processeurs possède le bloc à l'état (E) ou bien à l'état (S) dans le répertoire, alors cet état est gardé le même (figure 12.a). En effet, avec le protocole par mise à jour, la nouvelle valeur du mot modifié est envoyée aux autres

processeurs. De même, dans son mémoire cache l'état du bloc modifié est gardé à l'état (V) (figure 12.b). Si autrement, le bloc modifié est déjà à l'état (I) ou à l'état (O) pour l'un des processeurs, alors une opération (OW) ne modifie rien dans le répertoire.

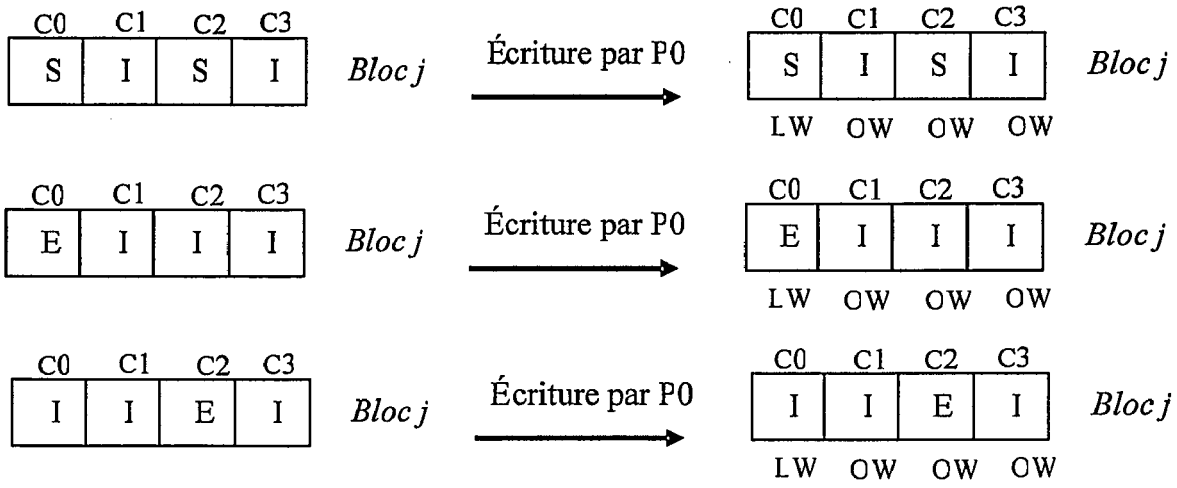


Figure 14. *Situation des états dans le répertoire suite à une écriture réalisée par le processeur 0 (P0) dans un bloc mémoire (Blocj) avec le protocole par mise à jour*

5. Fonctionnement du protocole ESIO avec une opération de lecture mémoire

Une opération de lecture d'un bloc partagé peut être de deux types (figure 15):

- ***Local Read* (LR)**: L'opération de lecture d'un bloc partagé qui est réalisée par un processeur est considérée pour ce processeur comme une opération (LR) pour (Local Read). Suite à un échec de cache, deux situations peuvent se produire : Dans la première situation, l'état du bloc recherché dans le répertoire correspond à l'état (O). Tandis que dans la deuxième situation, l'échec de cache peut être la cause de l'éjection du bloc recherché de la mémoire cache. Cette opération est notée « BE » dans la figure 12 pour (*Block Ejection*). Dans ce cas, l'état du bloc dans le répertoire est mis à l'état invalide ou (I). Dans les deux situations, après une opération (LR), le contrôleur du répertoire passe l'état de ce bloc de l'état (O) ou (I) à l'état (E) si aucun processeur ne possède ce bloc à l'état valide. Par contre, si un ou plusieurs processeurs possèdent une copie valide de ce bloc alors l'état de ce bloc dans le répertoire passe à l'état (S) (figure 12.a). Dans le cache l'état du bloc après un (LR), passe de l'état (I) à l'état (V) (figure 12.b).

- ***Other Read*** **(OR)**: L'opération de lecture d'un bloc partagé déclenchée par un processeur suite à un échec de cache, est considérée pour les autres processeurs comme une opération (OR) pour (*Other Read*). Si l'un de ces processeurs contient une copie valide du bloc recherché à l'état (E) dans le répertoire, alors après une opération (OR) cet état passe à l'état (S). Autrement, si le bloc est à l'état (S) dans quelques processeurs, alors cet état est gardé le même dans le répertoire. Pour les processeurs qui possèdent ce bloc à l'état (O) ou (I), après une opération (OR) ils gardent le même état dans le répertoire (figure 12.a).

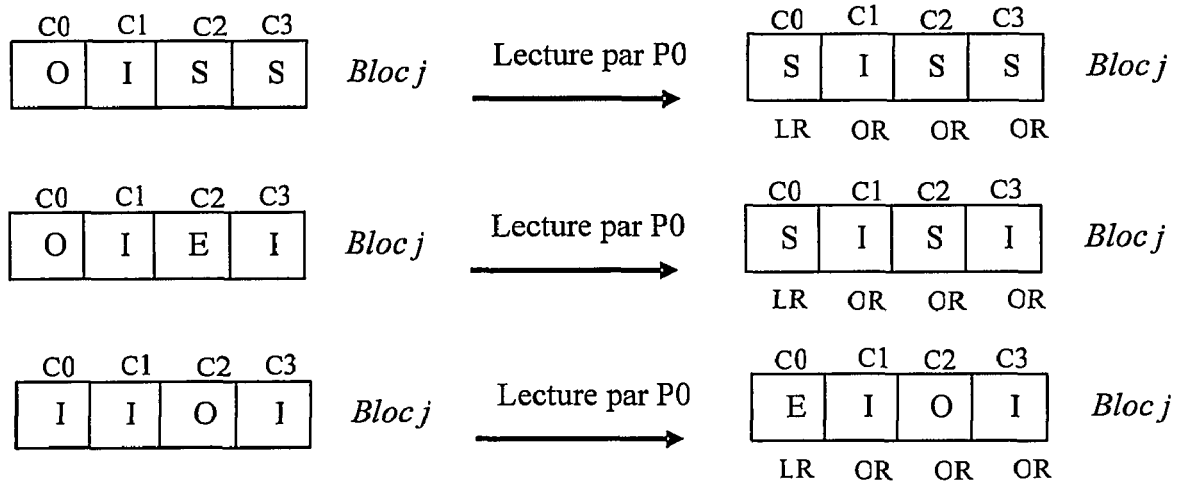


Figure 15. *Changement des états dans le répertoire suite à une lecture réalisée par le processeur 0 (P0) dans un bloc mémoire (Blocj)*

Après avoir expliqué les différents états manipulés par le contrôleur du répertoire à travers le protocole ESIO dans cette section, nous détaillons dans la prochaine section le principe de fonctionnement du protocole hybride proposé. Plus particulièrement, nous expliquons la méthodologie adoptée pour la sélection du protocole adéquat (invalidation ou mise à jour) avec un modèle d'accès donné au cours du temps d'exécution.

6. Principe du protocole hybride

La description du principe de fonctionnent du protocole hybride proposé est réalisée à l'aide de l'algorithme présenté par la figure 18. Ainsi, comme le montre cette figure, lorsqu'une opération d'écriture dans un bloc de cache est déclenchée par un processeur, deux opérations peuvent se produire : La première opération consiste à invalider les copies de ce bloc dans les autres processeurs. Tandis que la deuxième opération décide de mettre à jour ces copies. Pour ceci un bit, appelé « P », est utilisé pour représenter le protocole qui va

être utilisé pour chaque bloc de mémoire (figure 16). Si le bit « P » est mis à « INV » (0 dans l'implémentation) alors le protocole à employer est l'invalidation. Sinon, c'est le protocole par mise à jour « U » (1 dans l'implémentation) qui va être utilisé.

La stratégie que nous avons adoptée pour la sélection de l'opération de cohérence adéquate au modèle d'accès à un bloc partagé, est basée essentiellement sur l'analyse des accès précédents à ce bloc par les différents processeurs. Initialement, le protocole par invalidation est utilisé, ainsi le bit « P » est mis à 0. De ce fait, un message d'invalidation est envoyé aux processeurs possédants une copie du bloc modifié. Or, d'après le protocole ESIO présenté dans la section précédente, après l'opération d'invalidation, les processeurs qui possèdent ce bloc à l'état (S) ou (E) dans le répertoire passent à l'état (O). Ce protocole est employé pour chaque opération d'écriture jusqu'à avoir une première opération de lecture de ce bloc. En effet, un échec de cache réalisé sur un bloc qui est à l'état (O) dans le répertoire (c'est-à-dire ce bloc vient d'être invalidé), implique qu'il y a au moins un processeur qui a besoin de ce bloc. Dans ce cas le protocole adéquat est celui par mise à jour des données. Ceci introduit le passage du protocole de l'invalidation à celui par mise à jour. Par la suite le bit « P » passe de 0 à 1.

Le protocole par mise à jour est utilisé suite à un certain nombre d'opérations d'écriture. Pour revenir au protocole par invalidation, notre algorithme doit détecter des mises à jour inutiles du bloc correspondant. Dans la littérature, plusieurs solutions (Karlin *et al.*, 1986) (Anderson *et al.*, 1996) sont proposées pour déterminer le nombre nécessaire des opérations de mise à jour qui doit être atteint avant de retourner au mode d'invalidation. Ce nombre est appelé « *Update threshold* » ou encore seuil des opérations de mise à jour. Par contre, ces solutions ne tiennent pas en compte le comportement dynamique des programmes parallèles. À l'inverse, la nouvelle solution que nous proposons dans cette thèse, estime au cours de l'exécution pour chaque bloc mémoire la valeur adéquate du seuil des opérations de mise à jour. En effet, cette valeur varie dynamiquement selon les modèles d'accès au bloc mémoire.

Pour déterminer dynamiquement la valeur du seuil nous associons à chaque bloc mémoire deux compteurs. La figure 16 représente la nouvelle structure du répertoire.

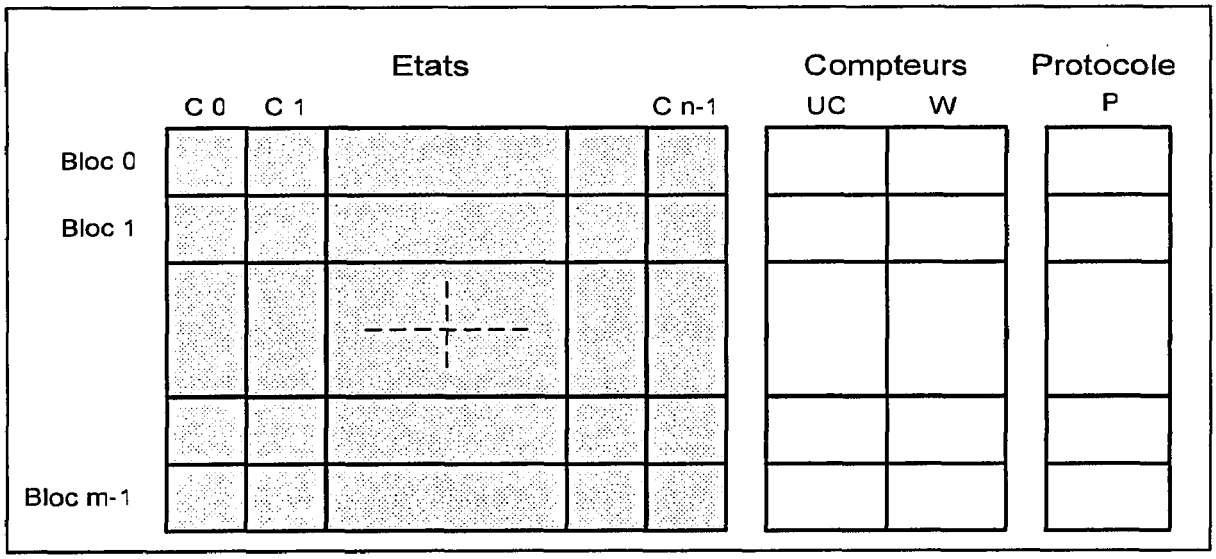


Figure16. *Nouvelle structure du répertoire*

Le premier compteur, noté « UC » pour (*Update Counter*), il représente le seuil des opérations de mise à jour qui doit être atteint avant de retourner au mode d'invalidation. Initialement « UC » est mis à zéro. Lors de l'exécution, ce compteur s'incrémente et se décrémente en fonction du modèle d'accès au bloc. Le second compteur, noté « W » pour (*Write operation*), il a deux fonctions:

(1) Avec le protocole par invalidation : le compteur « W » compte le nombre d'opérations d'écriture successives réalisé sur le bloc de mémoire par les différents processeurs. Lorsqu'une opération de lecture se produit par l'un des processeurs, le contrôleur du répertoire vérifie l'état du bloc pour ce processeur. Si le bloc est à l'état (O) dans le répertoire (figure 18, partie gauche), le compteur « W » est testé. Deux situations sont possibles :

- La valeur de « W » est faible (inférieur à Δ): Cela signifie que le nombre d'opérations d'écriture réalisé sur le bloc sans être utilisé en lecture est faible. En d'autres termes, pour un faible nombre d'opérations d'écriture successives, les échecs de cache causés par le protocole par invalidation sont très proches. Par la suite, pour minimiser ces échecs, le bloc mémoire doit être mis à jour. Pour ceci, le protocole passe du mode d'invalidation au mode de mise à jour (P est mis à 1). Le compteur « UC » est incrémenté pour indiquer que durant cette phase d'exécution de l'application, le protocole par mise à jour est le plus adapté au modèle d'accès au bloc. À ce moment, le compteur « W » prend la valeur du compteur « UC » et il se prépare à un

nouveau fonctionnement avec le protocole par mise à jour. Cette première situation est illustrée par la figure 17.a. Dans cette figure la notation « Ri » signifie une opération de lecture réalisée par un processeur « i ». De même « Wi » signifie une opération d'écriture réalisée par un processeur « i ».

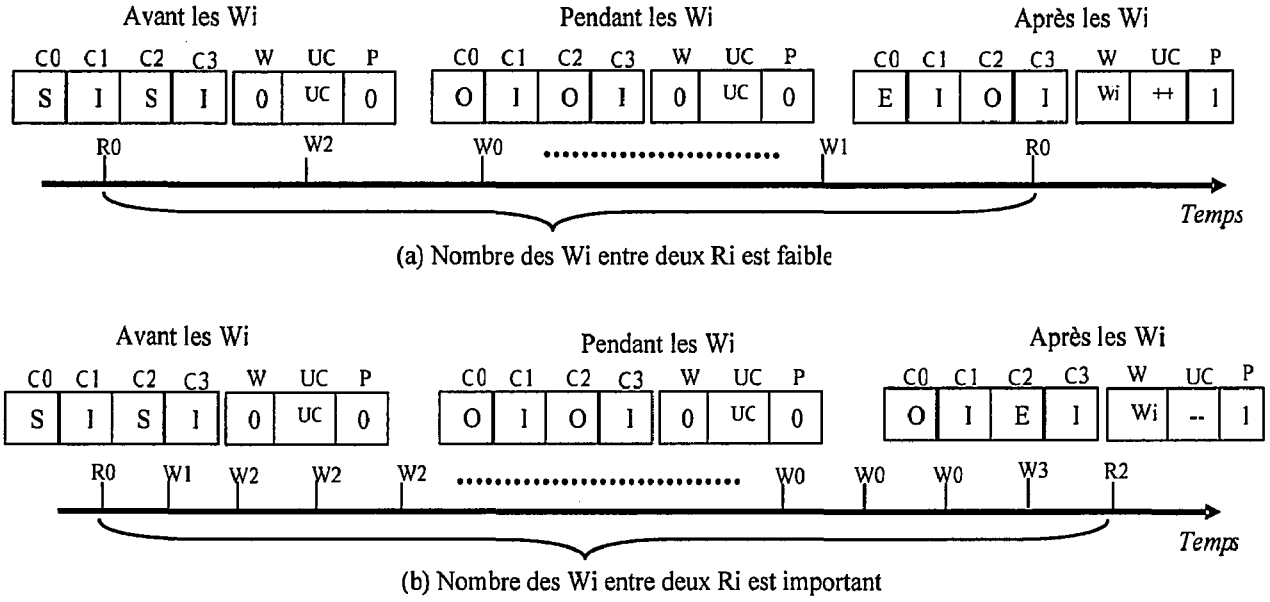


Figure 17. *Nombre des opérations totales d'écriture réalisé avec le protocole par invalidation sans avoir un échec de cache (O) au cours du temps d'exécution*

- La valeur de « W » est importante (supérieur à Δ): cela signifie que pour un nombre important d'opérations d'écriture réalisé sur le bloc, les échecs de cache causés par le protocole par invalidation sont très loin. Par la suite, le bloc mémoire est utilisé en écriture plus qu'en lecture. Par conséquent, durant cette période d'exécution, le protocole par invalidation est plus intéressant que le protocole par mise à jour. Néanmoins, le protocole doit passer au mode de mise à jour (P prend 1), puisque le protocole par invalidation a causé quand même un échec de cache. Par contre cette fois, le compteur «UC» est décrémenté (figure 17.b). De cette façon, la période dans la quelle le protocole par mise à jour va être employé est réduite. Par conséquent, après un échec de cache, le système passe rapidement au protocole par invalidation. Lorsque « UC » atteint 0, « P » est mis à 0 et le protocole passe à l'invalidation. Nous notons que également dans ce cas « W » prend la valeur de « UC » pour un objectif à réaliser au cours de l'emploie du

protocole par mise à jour.

(2) Avec le protocole par mise à jour : comme c'est déjà indiqué, après un échec de cache, le bit « P » passe à la mise à jour (P prend 1) et le compteur « W » prend la valeur du compteur « UC ». En effet, « W » sert à compter le nombre d'opérations d'écriture réalisé avant de changer le protocole du mode mise à jour à celui d'invalidation. Ainsi, « W » est employé comme un compteur intermédiaire qui prend initialement la valeur de « UC », il se décrémente après chaque opération de mise à jour jusqu'à atteindre la valeur 0, dans ce cas le protocole doit passer au protocole par invalidation (P est mis à 0) (figure 18, partie droite). Par conséquent, au mode de mise à jour « W » permet de ne pas effacer l'ancienne valeur du compteur « UC » puisqu'il va être utilisé prochainement.

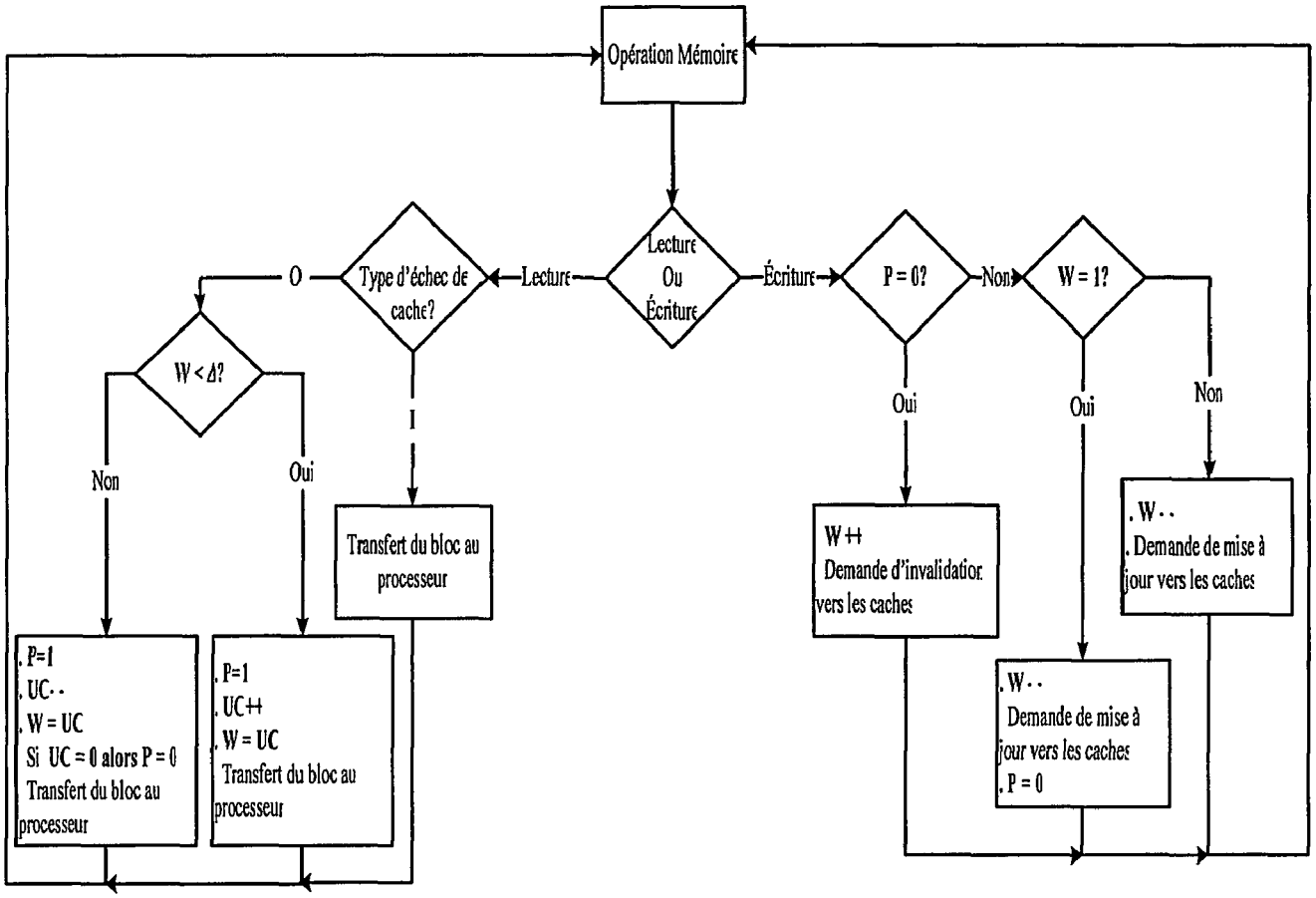


Figure 18. *Algorithme du protocole hybride*

En conclusion, avec le protocole hybride que nous proposons, le compteur « UC » représente le seuil des opérations de mise à jour qui doit être atteint avant de retourner au mode d'invalidation. Ce compteur varie avec l'intensité (ou la fréquence) des opérations de lecture réalisées dans la mémoire partagée, dans une période de temps déterminée. Cette

intensité est estimée par rapport au nombre d'opérations totales d'écriture. Ceci permet de choisir dynamiquement le protocole (mise à jour ou invalidation) adéquat à l'utilisation. La figure 19 récapitule le principe de notre protocole hybride. En effet, comme le montre cette figure pour passer du protocole par invalidation à celui par mise à jour, il suffit d'avoir un échec de cache de type (O). Pour revenir au protocole par invalidation, il suffit que la valeur du compteur W prenne la valeur 0. Nous notons ici, que durant la phase des expérimentations du protocole proposé, nous avons fixé la valeur de (Δ) à 1000 opérations d'écriture.

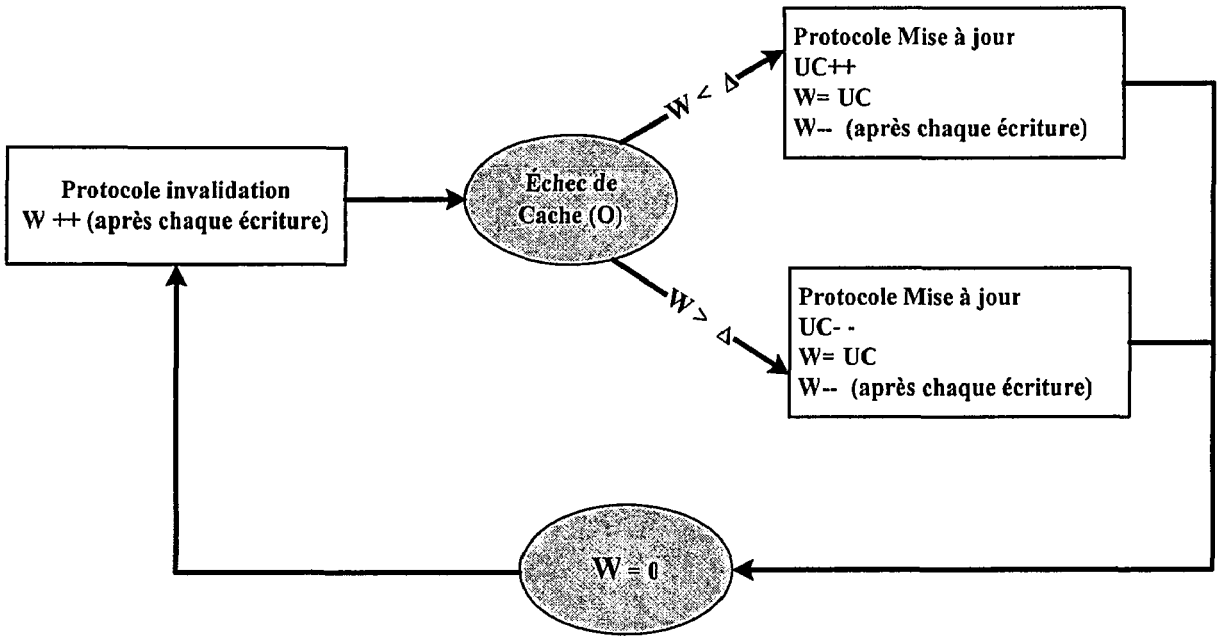


Figure 19. *Principe du protocole hybride proposé*

7. Conclusion

Pour concevoir un mécanisme performant pour le maintien de la cohérence des données dans les caches dans les MPSoCs, nous avons proposé dans ce chapitre un nouveau protocole hybride et dynamique pour la cohérence des données dans les caches. En premier lieu, une description des différentes caractéristiques du protocole proposé est présentée. En effet, nous avons expliqué le choix de ces caractéristiques, ainsi que leur influence sur l'efficacité de ce protocole. En second lieu, nous avons présenté le protocole ESIO, sur le quel est basé le protocole hybride. Ensuite, nous avons détaillé le principe de fonctionnement du protocole hybride proposé, ainsi que la stratégie adoptée pour la sélection du protocole adéquat pour chaque modèle d'accès au bloc partagé. Grâce à cette

stratégie, le protocole hybride est capable de s'adapter dynamiquement avec le comportement de l'application parallèle. Dans le prochain chapitre nous proposons une architecture matérielle qui facilite sa mise en œuvre et qui permet de le rendre plus performant.

Chapitre 4

Architecture matérielle pour le protocole proposé

1. Introduction

Afin d'améliorer l'efficacité et les performances des MPSoCs à mémoire partagée l'utilisation des mécanismes assurant la cohérence des données de façon performante devient cruciale. Une attention particulière doit être aussi portée à l'implémentation du mécanisme du protocole sur l'architecture du MPSoC et qui permet ainsi d'offrir de meilleures performances. Ainsi, en outre de la contribution principale de cette thèse qui consiste à proposer un protocole hybride et dynamique pour la cohérence des données dans les caches, nous proposons aussi une architecture matérielle qui facilite et optimise les performances de ce protocole. C'est dans ce cadre que se situe l'objectif principal de ce chapitre. En effet, nous présentons dans ce chapitre notre architecture matérielle afin de faciliter l'implémentation du protocole proposé dans le chapitre précédent et optimiser ainsi ses performances. Comme type d'MPSoC, nous visons les architectures multiprocesseurs symétriques (SMP) à mémoire partagée multi-bancs. En effet, ces architectures sont très attractives du fait qu'elles facilitent à la fois le développement des applications parallèles, grâce à leur modèle de programmation, ainsi que la possibilité d'intégrer un nombre relativement important de processeurs.

L'organisation du chapitre est comme suit : Nous présentons dans la section 2 l'architecture globale du système adopté pour l'implémentation de l'approche proposée. L'implémentation du répertoire ainsi que son contrôleur pour le protocole hybride présenté dans le chapitre précédent sont détaillées dans la section 3. Plus particulièrement, l'objectif de cette section consiste à préciser la relation de ce répertoire avec les processeurs d'un côté et la mémoire partagée d'un autre côté. Dans la section 4, nous proposons l'implémentation d'un bus additionnel dédié aux transferts des messages de cohérence entre le contrôleur du répertoire et les contrôleurs des mémoires caches dans les différents processeurs. Finalement, la manipulation de la consistance des données en mémoire à l'aide du bus proposé est présentée dans la section 5.

2. Architecture du système

Comme architecture multiprocesseur embarquée, nous visons l'architecture multiprocesseur symétrique (SMP) à mémoire partagée multi-bancs (figure 20). En effet, comme évoqué dans le chapitre de l'état de l'art, la tendance actuelle s'oriente vers la conception de ce type d'architecture. Ceci s'explique par le fait que cette architecture permet une gestion de la communication entre processeurs plus simple que celle des MPSoCs à

mémoire distribuée d'une part et par le fait qu'elle est plus extensible que les architectures MPSoCs à mémoire centralisée d'autre part.

Les processeurs utilisés dans l'architecture que nous avons adoptée sont homogènes et symétriques. Ils contiennent des mémoires caches de niveau L1. La mémoire partagée est distribuée physiquement et de manière uniforme, en plusieurs bancs de mémoires. Tous ces bancs de mémoire sont accessibles par tous les processeurs, qui voient ainsi le même espace d'adressage. L'architecture intègre aussi un réseau d'interconnexion sur puce (NoC). Il s'agit d'un réseau complexe qui peut être un crossbar, un réseau multi-étages « *multi-stage interconnection network* » (MIN), etc. Ainsi, sur cette architecture nous avons implémenté un nouveau mécanisme matériel dédié au maintien de la cohérence des données dans les caches. Une implémentation matérielle efficace qui permet d'améliorer les performances du protocole hybride proposé sera présentée dans la suite de ce chapitre.

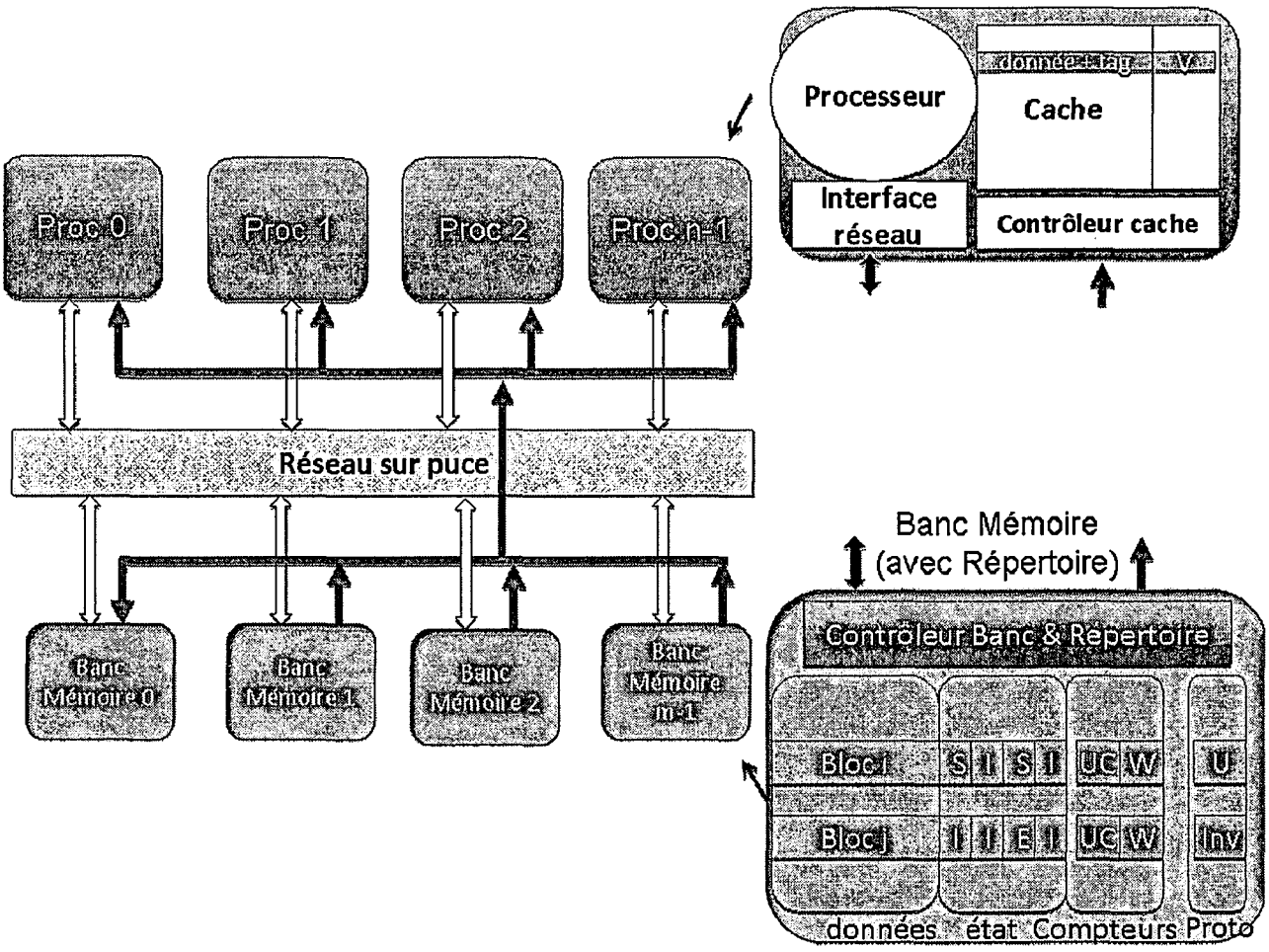


Figure 20. *Architecture globale du système*

3. Implémentation du Répertoire

Comme nous l'avons déjà évoqué dans le chapitre précédent, l'approche proposée pour la cohérence est basée sur le mécanisme du répertoire. Néanmoins, l'inconvénient majeur de ce mécanisme est le fait que pour chaque accès au cache, il est nécessaire de connaître à la fois l'état du bloc référencé et les processeurs qui le partagent avant chaque utilisation. Ce passage fréquent par le répertoire devient coûteux en termes de temps et de communication à travers le réseau surtout lorsque le nombre de processeurs est important. Pour cette raison, des études (Gustavo *et al.*, 2007) ont mis l'accent sur cet aspect. Les auteurs de (Gustavo *et al.*, 2007) ont mesuré les quantités de données transférées à travers le réseau pour réaliser les opérations de lecture et d'écriture de données pour l'exécution de l'application (les instructions chargement et stockage des données) et pour la gestion de cohérence centralisée.

Il a été constaté que pour des caches de petite taille, le nombre de paquets transférés pour le maintien de la cohérence des données reste faible par rapport au nombre de paquets utilisés pour lire ou écrire les données. Cependant, la situation devient plus complexe lorsque la taille du cache augmente. En effet, pour des grandes tailles de cache, les opérations de lecture et d'écriture diminuent du fait que le nombre d'échecs en cache diminue, alors que le nombre de paquets dus aux opérations de cohérence reste presque constant. Par conséquent, le coût de la gestion du répertoire, en termes de surcharge d'opérations de communication, dépend de la taille du cache. Cette situation devient très critique dans le contexte des architectures MPSoCs. Dans ce contexte, plusieurs solutions sont apparues afin de résoudre les coûts de gestion du répertoire. En effet, les auteurs de (Noel *et al.*, 2006) et (Evgeny *et al.*, 2007) ont changé l'architecture du NoC afin d'optimiser les performances du répertoire. Bien qu'elles montrent une baisse du temps d'exécution, ces solutions se basent sur une conception matérielle complexe et coûteuse des routeurs.

En conclusion, nous pouvons conclure que malgré sa large utilisation dans les architectures hautes performances, le mécanisme du répertoire pose des problèmes de coûts d'implémentation dans les architectures embarquées. Dans cette section, nous présentons une solution pour permettre l'utilisation d'une telle gestion de la cohérence des données, mais qui génère une surcharge réduite en termes de temps d'accès, de consommation d'énergie et de nombre d'interconnexions supplémentaires. Ainsi, nous présentons la méthodologie suivie pour l'implémentation du répertoire. L'interaction entre le répertoire et les autres composants du système sera aussi présentée.

3.1. *Position du répertoire*

La plupart des solutions matérielles existantes pour la gestion de la cohérence des données dans les caches sont basées sur le mécanisme du répertoire. Avec ce mécanisme la cohérence des données dans les caches est réalisée en utilisant un répertoire qui maintient des informations sur les états de chaque bloc de cache dans le système. Le choix de la position du répertoire dans le système multiprocesseur, joue un rôle important sur l'efficacité du protocole de cohérence. En effet, nous distinguons l'approche qui consiste à placer le répertoire au centre du système et l'approche qui consiste à le distribuer dans les différents bancs mémoires. Généralement, la première approche est adoptée aux systèmes multiprocesseurs à mémoire partagée centralisée (Dubois *et al.*, 1982) (Censier *et al.*, 1978). À l'inverse, pour les architectures MPSoCs à mémoire partagée distribuée, le répertoire est distribué à travers les différents nœuds de processeurs. Les systèmes multiprocesseurs SGI Origin (Laudon *et al.*, 1997) et DASH multiprocesseur (Daniel *et al.*, 1992) emploient cette méthode.

Bien que le répertoire soit un composant indépendant de la mémoire, nous constatons d'après ces deux alternatives fréquemment employées, que la position du répertoire dépend de la position de la mémoire partagée. Néanmoins, les auteurs de (Noel *et al.*, 2006) et (Evgeny *et al.*, 2007) ont proposé l'implémentation du répertoire à l'intérieur du NoC.

Dans notre travail, à chaque banc de mémoire est associé un répertoire qui manipule les états de ses blocs mémoire (figure 20). Ce choix s'explique par le fait que cette approche n'exige pas la modification de l'architecture du NoC comme pour (Noel *et al.*, 2006) et (Evgeny *et al.*, 2007). En outre, en comparaison avec l'alternative qui consiste à implémenter le répertoire à travers les nœuds de processeurs (Laudon *et al.*, 1997) (Daniel *et al.*, 1992), elle permet de réduire les coûts liés au routage des messages de cohérence dans le NoC. En conséquence, l'approche choisie est adéquate avec l'architecture globale du système et elle permet une implémentation facile du protocole hybride proposé.

3.2. *Répertoire associé avec la mémoire partagée*

Habituellement, le répertoire avec son contrôleur représentent un composant séparé et indépendant des mémoires caches et de la mémoire partagée. Ce composant est accessible à tous les processeurs et également à la mémoire partagée. Ceci, n'est pas le cas pour l'implémentation proposée dans ce chapitre. En effet, afin d'exploiter le fait que la mémoire

partagée est mise à jour après chaque opération d'écriture dans la mémoire cache grâce à la technique d'écriture simultanée en cache, nous associons le répertoire à la mémoire partagée. Ainsi, à chaque banc de mémoire est associé un répertoire avec son contrôleur (figure 20). Par conséquent, quand un processeur envoie une requête d'écriture à travers le NoC à ce banc mémoire, l'opération de mise à jour du répertoire est déclenchée. Ce mécanisme simplifie le maintien de la cohérence des données dans les caches, puisqu'il n'a pas besoin d'utiliser des paquets dédiés à mettre à jour le répertoire. Grâce à cette distribution des répertoires, il n'est pas nécessaire de transmettre des messages dans le NoC, et il est possible de traiter plusieurs opérations d'écritures simultanément.

En conséquence, la FSM du protocole ESIO et l'algorithme du protocole présentés dans le chapitre précédent, sont implémentés dans la mémoire partagée. Ainsi le contrôleur de la mémoire s'occupe de la gestion de cet algorithme. Il manipule également les différents états du répertoire. Ce qui permet de réduire les coûts de la gestion du répertoire.

4. Un bus pour la gestion des opérations de cohérence des données dans les caches

Dans l'objectif de mettre à jour ou d'invalider les blocs partagés dans les différents caches, généralement un paquet d'invalidation ou de mise à jour est envoyé par le contrôleur de mémoire partagée aux différents caches à travers le NoC. Par contre, cette solution encombre le trafic de données dans le NoC. D'une part, la gestion de ce trafic devient très complexe. D'autre part, le besoin en termes de bande passante et en énergie augmente également. Afin d'éviter ces inconvénients liés au transfert des messages de cohérence à travers le NoC, notre solution consiste à utiliser un réseau basse consommation dédié à la transmission des messages de mise-à-jour et d'invalidation. De ce fait, nous proposons l'utilisation d'un bus simple et unidirectionnel (figure 20). Le bus proposé est appelé «Bus de cohérence», son rôle consiste à transférer les messages de cohérence à partir des différents bancs de mémoire vers les différents processeurs. Bien que son rôle consiste simplement à actualiser les caches selon le protocole (mise à jour ou invalidation) utilisé, ce bus permet de soulager le réseau d'interconnexion des opérations de maintien de la cohérence des données dans les caches. Par conséquent, la mise à jour ou l'invalidation des données partagées est rapidement réalisée à travers le bus de cohérence. De même, le trafic de données dans le NoC devient moins important. Ce qui réduit ses besoins en termes de bande passante et de consommation d'énergie.

Cette approche hybride qui combine deux types de réseaux dont chacun est dédié à un type de transfert de données, est l'objet de notre travail dans (Chtioui, 2008). Nous notons également que dans un travail plus récent les auteurs de (Ran *et al.*, 2009) ont proposé une approche appelée BENoC pour *Bus-Enhanced Network on-Chip*. Cette approche combine aussi un bus hiérarchique (MetaBus) avec un NoC.

Comme nous allons le détailler dans le chapitre 6, l'utilisation de ce bus pour la gestion des messages de cohérence pour les systèmes considérés, offre des performances relativement intéressantes (figures 41 et 42 pages 108 et 109).

4.1. *Architecture du bus pour la gestion de la cohérence des données*

Nous notons que le bus implémenté pour le transfert des messages de cohérence des données dans les caches, travaille de manière synchrone avec le NoC et avec les autres composants du système. Bien que les processeurs soient maîtres (ou initiateurs) avec le NoC, à l'inverse, avec le bus proposé les processeurs deviennent esclaves (ou cibles). Ceci confirme l'utilité du bus qui sert à réduire les conflits d'accès au NoC. Nous précisons ici que ce bus est connecté réellement aux contrôleurs de mémoires caches de chacun de ces processeurs.

L'architecture interne du bus se compose de quatre parties. La première partie correspond à la commande appelée « CMD ». Il s'agit d'informer les caches des modifications qui se passent sur les blocs. Selon la valeur du « CMD », le type de l'opération à effectuer est soit une mise à jour du bloc modifié, soit son invalidation. Le tableau 7 explique le fonctionnement de ce signal.

Tableau 7. *Fonctionnement de l'information CMD*

Opération à effectuer	**Valeur du CMD**
Mode désactive (repos)	00
Mode active : invalidation	01
Mode active : Mise à jour	10

La deuxième partie du bus est appelée « ADR » et elle correspond aux 32 bits de

l'adresse du bloc modifié. La troisième partie du bus appelée « DATA » est utilisée dans le cas du protocole par mise à jour. Cette partie du bus sert aux 32 bits de la nouvelle valeur du mot modifié. Finalement, la quatrième partie « NUM » du bus sert à identifier le numéro du processeur qui a causé l'opération d'écriture dans le bloc. Cette dernière information, est nécessaire pour invalider ou mettre à jour toutes les copies du bloc modifié sauf la copie dans le cache du processeur qui a causé la modification.

Afin d'expliquer le fonctionnement de chaque partie du bus, nous étudions l'exemple présenté par la figure 21. Dans cette figure, le banc de mémoire 2 (Banc 2) diffuse dans le bus de cohérence une requête de cohérence, suite à une opération d'écriture réalisée dans ce banc par le processeur 1 (Proc 1). Deux cas sont possibles, selon le protocole employé :

1. Dans le premier cas, nous supposons que le protocole utilisé est celui de la « mise à jour ». Ainsi, CMD est mis à 10, ADR prend l'adresse du bloc à modifier et DATA prend la nouvelle valeur du mot modifié dans ce bloc. Concernant NUM, il prend le numéro du processeur qui a réalisé l'opération d'écriture, (« 1 » dans notre exemple. Après avoir reçu la requête venant du (banc 2), chaque contrôleur de cache vérifie la valeur de NUM. Si cette valeur est différente de son numéro, alors il vérifie si l'adresse ADR existe dans son cache. Si le cache possède une copie du bloc modifié, il remplace le mot correspond à cette adresse par la nouvelle valeur du mot fournie par le signal DATA. A l'opposé, si le cache ne possède pas une copie du bloc modifié, il n'est pas concerné par cette requête. Ainsi dans notre exemple, si la valeur de NUM est égale à «1 », le cache du processeur 1 n'est pas modifié. Pour tous les autres processeurs, le mot mémoire correspondant à ADR est mis à jour avec la donnée DATA.
2. Dans le deuxième cas, le protocole employé est celui d'invalidation. CMD prend la valeur (01). Comme dans le premier cas, chaque nœud du MPSoC doit vérifier s'il est concerné par la requête venant du banc de mémoire. Autrement dit, la valeur de NUM doit être différente du numéro du processeur. Si en plus une copie du bloc mémoire d'adresse ADR est présente dans le cache, ce dernier est invalidé. Le contrôleur du cache change l'état de ce bloc dans son cache de l'état valide (V) à l'état invalide (I).

En conclusion, cette architecture malgré sa simplicité, exige un nombre d'interconnexions et une complexité réduite.

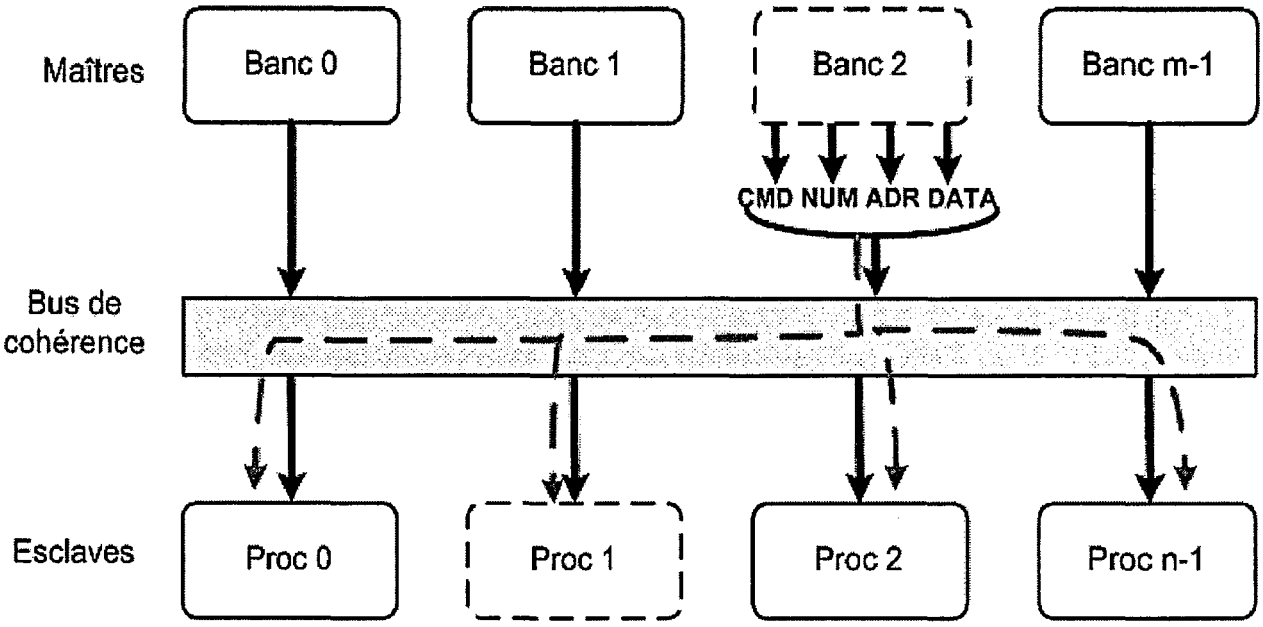


Figure 21. *Architecture du Bus de cohérence*

4.2. *Politique d'arbitrage du bus de cohérence*

Toute opération d'écriture réalisée par un processeur dans un bloc partagé entraîne l'envoie d'une requête d'invalidation ou de mise à jour aux autres processeurs. Cette requête est envoyée par le banc de mémoire qui possède ce bloc. Dans le cas où plusieurs processeurs modifient en même temps des blocs partagés, plusieurs requêtes doivent être envoyées en même temps sur le bus de cohérence. Ceci crée donc une situation de conflit sur ce bus de gestion de cohérence. Afin d'éviter ces conflits, nous avons employé une politique d'arbitrage. Cette politique permet de gérer et contrôler les conflits qui peuvent se produire dans le bus. Il s'agit de la politique d'arbitrage à priorité tournante ou *round robin*. Cette technique assure une distribution équitable de l'accès au bus par les différents bancs de mémoire. En effet, nous associons à chaque banc de mémoire un numéro. Ainsi les « m » bancs de mémoire sont numérotés de « 0 » jusqu'à « m-1 ». Initialement le banc 0 possède la plus haute priorité. Par la suite, il est le premier servi. Le prochain cycle, la priorité la plus élevée passe au banc 1 qui va être servi. Tandis que la priorité du banc 0 passe à la priorité la plus basse, et ainsi de suite. Si le banc qui a la priorité la plus élevée n'a pas besoin d'accéder à ce cycle, le banc qui a la plus haute priorité suivante se permet d'accéder.

Le choix de la politique d'arbitrage à priorité tournante est expliqué par le fait que d'un côté cette politique est facile à mettre en œuvre. De l'autre côté, nous avons réalisé des expérimentations qui consistent à mesurer le rapport (nombre de conflit/nombre de

messages) dans le bus. Ainsi, nous avons trouvé que ce rapport est très faible. En effet, nous avons testé le protocole proposé à l'application de multiplication de matrices (MM) avec une architecture MPSoC qui contient quatre processeurs. Le résultat obtenu est illustré par la figure 22. Cette figure montre que le nombre de conflits dans le bus de cohérence n'est pas significatif en comparaison avec le trafic des messages de cohérence.

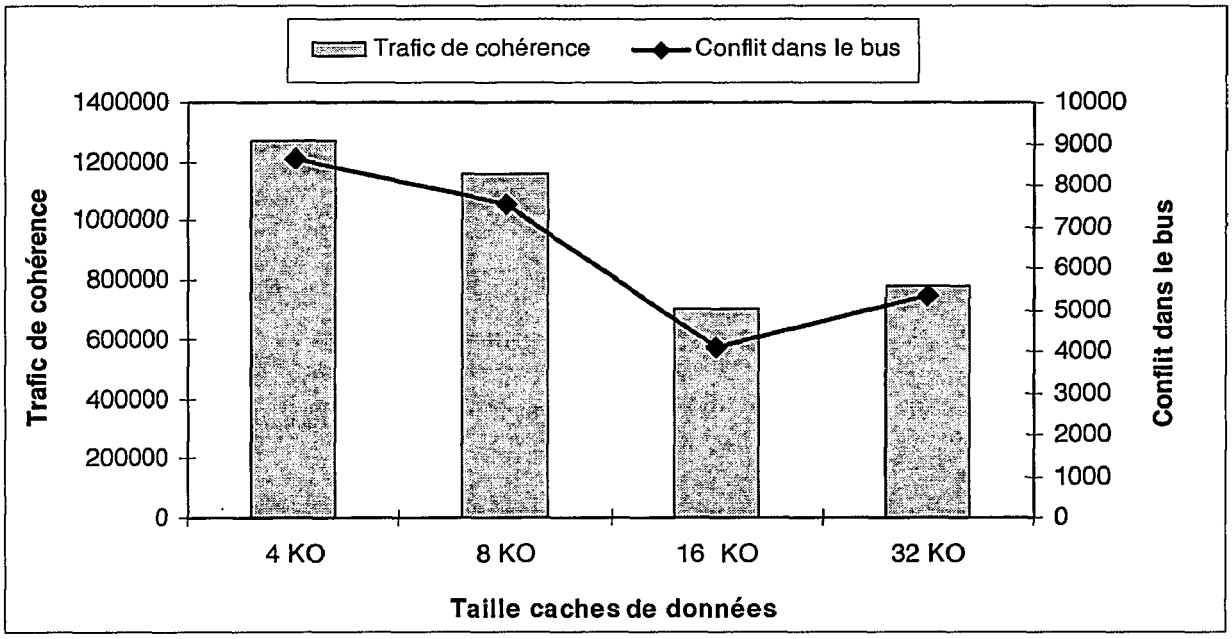


Figure 22. *Le nombre total des messages de cohérence (axe gauche) et le nombre total de conflits dans le bus (axe droit) en fonction de différente taille de cache, pour un MPSoC 4-processeurs et avec l'application MM*

5. Gestion de la consistance des données mémoire

Comme déjà évoqué dans le chapitre état de l'art, la gestion de la cohérence des données dans les caches et le problème de gestion de la consistance de ces données en mémoires sont indépendants l'un par rapport à l'autre. En effet, généralement un MPSoC doit fournir un mécanisme pour maintenir la cohérence des données en cache d'une part, et employer un modèle de consistance de mémoire d'autre part. Bien que la contribution principale de cette thèse consiste à proposer une solution efficace pour le maintien de la cohérence des données dans un MPSoC, nous proposons dans cette section d'étendre l'approche proposée pour la cohérence et la rendre capable de garantir la consistance de mémoire.

Tout d'abord, nous rappelons le problème de consistance de mémoire déjà expliqué dans le chapitre de l'état de l'art. Ainsi, un système de mémoire est dit non consistant lorsqu'un processeur réalise une opération de lecture d'un bloc partagé qui vient d'être modifié, avant que cette modification soit visible pour ce processeur et pour les autres processeurs. Pour éviter ce problème, si un processeur déclenche une opération d'écriture, il ne peut pas la considérée finie qu'après qu'elle soit visible par tous les autres processeurs. Afin de détecter la fin de son opération d'écriture, le processeur doit envoyer des messages d'acquittement (d'invalidation ou de mise à jour) aux autres processeurs. Une fois cette opération terminée, il récupère leurs messages d'acquittement, il peut alors considérer l'écriture terminée. En conclusion, l'objectif des messages d'acquittements consiste à définir un intervalle de temps nécessaire pour propager ou invalider une opération d'écriture vers tous les processeurs.

Dans ce cadre, nous proposons une solution plus simple qui réduit la complexité du système d'acquittement que nous venons de décrire. L'idée consiste à retarder l'opération d'écriture déclenchée par un processeur, jusqu'au moment de l'actualisation des autres processeurs. En effet, lorsqu'un processeur déclenche une opération d'écriture dans un bloc partagé, il ne peut pas la considérée finie qu'après avoir reçu la requête de mise à jour ou d'invalidation venant par le banc de mémoire contenant ce bloc. Lorsqu'il reçoit cette requête, ce processeur vérifie si la valeur du signal NUM coïncide avec son numéro. Si c'est le cas, ce processeur peut considérer l'écriture qu'il l'a déclenchée finie. Dans le cas où le bloc est à l'état *Exclusive* dans le répertoire, d'après le protocole ESIO aucune requête ne sera renvoyée aux processeurs. Par la suite, le processeur attend pendent un cycle, s'il ne reçoit rien alors il considère son écriture finie.

6. Conclusion

Pour répondre aux besoins des MPSoCs en termes de performance et de consommation d'énergie, il est indispensable de concevoir des mécanismes de cohérence des données dans les caches performants et adéquats avec l'architecture du système. Notre contribution dans ce chapitre consiste à proposer une architecture matérielle nouvelle qui facilite et optimise les performances du protocole décrit dans le chapitre précédent.

Dans ce travail, bien que le protocole proposé puisse être implémenté sur n'importe quel type d'architecture MPSoC, nous l'avons adopté avec une architecture multiprocesseur

symétrique (SMP) à mémoire partagée multi-bancs. Avec l'implémentation du répertoire à l'intérieur de chaque banc de mémoire d'une part et l'implémentation d'un bus dédié au transfert des messages de cohérence d'autre part, ce protocole est performant et facile à mettre en œuvre. Afin d'évaluer ce mécanisme de cohérence des données dans les caches, nous devons le tester sur plusieurs applications embarquées. Comme nous allons le montrer dans le chapitre suivant, une phase de parallélisation des applications est d'abord réalisée. Dans cette phase de benchmarking, nous avons envisagé des applications avec différents modèles d'accès aux données partagées.

Chapitre 5

Parallélisation des benchmarks pour architecture MPSoC

1. Introduction

Les nouvelles applications de traitement de signal intensif (télécommunications, traitement multimédia, traitement d'images et de la vidéo, etc.) sont de nos jours les applications les plus fréquemment exécutées sur les systèmes mobiles et embarqués. Or, ces applications travaillent sur des structures de données de grandes tailles (des tableaux de plusieurs millions d'éléments). En outre, elles doivent effectuer en temps réel un volume important d'opérations arithmétiques. Or, la solution qui consiste à augmenter les performances de puces, contenant un seul processeur, n'est plus utilisée. En effet, cette solution bien qu'elle améliore les performances, elle entraîne l'augmentation de la consommation d'énergie. De ce fait, cette solution est remplacée par une nouvelle approche qui consiste à augmenter le nombre de processeurs intégrés sur la puce en gardant la même vitesse qu'un monoprocesseur mais en travaillant avec des fréquences plus faibles. Par conséquent, seules les architectures parallèles contenant plusieurs processeurs permettent de répondre aux besoins de ces applications en termes de performances. Ainsi, chaque processeur réalise en parallèle avec les autres processeurs une partie du travail de l'application. Ces systèmes permettent un traitement puissant et une gestion de grandes masses de données. De ce fait, la tendance actuelle s'oriente vers la conception des systèmes embarqués multiprocesseurs (MPSoCs). Parallèlement aux besoins en puissance de calcul de ces applications, l'augmentation des capacités d'intégration de transistors sur une même puce permet aujourd'hui de fabriquer des systèmes très complexes, ce qui favorise ainsi la tendance vers les architectures parallèles.

Afin de bien exploiter le parallélisme et de réduire le temps d'accès à la mémoire, chaque processeur de l'architecture MPSoC est équipé d'une mémoire cache. Cependant, avec l'ajout des mémoires caches à ces architectures, il faut faire face au problème de cohérence des données dans les caches. Ceci se réalise en utilisant un mécanisme de cohérence des données dans les caches. Pour qu'il soit efficace, ce mécanisme doit prendre en considération la façon avec laquelle les données partagées ont accédé aux différents processeurs. En d'autres termes, le protocole de cohérence des données dans les caches doit prendre en compte la méthode de parallélisme employée avec l'application. C'est dans ce même contexte que se situe l'objectif principal de ce chapitre. En effet, nous envisageons dans ce chapitre d'étudier les différentes méthodes de parallélisme, ainsi que les modèles d'accès aux données partagées issus de ces méthodes. Nous visons également de préciser le protocole de cohérence des données dans les caches le plus adéquat avec chaque type de ces

modèles d'accès.

Le plan de ce chapitre est comme suit : La section 2 détaille les différentes façons de parallélisation d'une application donnée. Ensuite, dans la section 3 nous déterminons pour chacune de ces façons de parallélisation, la classe de données partagées qui lui correspond. Nous précisons ainsi, pour chaque classe de données partagées le protocole de cohérence des données dans les caches le plus adéquat. Finalement, dans la section 4, nous montrons comment nous avons parallélisé un ensemble d'applications de traitement de signal intensif afin d'avoir plusieurs modèles d'accès aux données partagées. Ceci est dans le but de bien évaluer le protocole proposé.

2. Formes du parallélisme

La croissance continuelle des besoins des applications modernes de multimédia et de télécommunication, en capacité de traitement et en performances a mis en cause l'émergence des architectures parallèles. Nous citons par exemple les multiprocesseurs Power 4 d'IBM (Tendler *et al.*, 2001) et le MPCore d'ARM (ARM). Avec cette évolution vers les systèmes multiprocesseurs, les programmeurs des applications s'orientent également vers le développement des applications parallèles. Dans ce contexte, plusieurs outils de programmation parallèles sont apparus comme OpenMP (Dagum *et al.*, 1998), ou MPI pour *Message Passing Interface* (MPI, 1994). Ainsi, les programmeurs des applications et les concepteurs des systèmes visent une exploitation efficace du parallélisme. En effet, pour exploiter finement le parallélisme dans une application, il est nécessaire de bien identifier l'architecture parallèle envisagée pour exécuter ces applications. Comme ceci a été évoqué dans le chapitre de l'état de l'art, l'une des premières classifications des architectures parallèles est celle proposée par Flynn (Flynn, 1966). Le système de classification de Flynn est basé sur la notion de flots d'information. Il existe deux types de flots d'informations dans un processeur qui sont les instructions et les données.

1. Le flot d'instructions est défini comme la séquence d'instructions exécutée par le processeur.
2. Le flot de données est défini comme le trafic de données échangé entre la mémoire et le processeur.

Selon ces deux types de flots d'informations, l'exploitation du parallélisme dans une application peut être obtenue sous deux formes : Parallélisme d'instructions ou encore de contrôle et parallélisme de données.

2.1. *Parallélisme de données*

Avec ce premier type de parallélisme, un seul flot d'instruction traite un ensemble de flots de données. Par la suite, les processeurs exécutent en même temps la même instruction mais sur des données différentes. Ce type de traitement est dit traitement régulier ou encore systématique puisque les données sont traitées de la même manière. Pour trois données différentes (D1, D2 et D3), la figure 23 explique le modèle de programmation avec parallélisme de données.

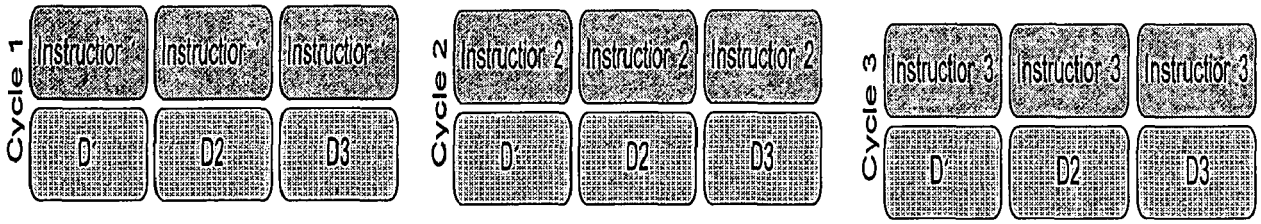


Figure 23. *Modèle de programmation avec parallélisme de données*

Généralement, ce type de parallélisme est exploité par les machines SIMD (*Single Instruction stream Multiple Data stream*). C'est une machine dans laquelle tous les processeurs sont synchrones. Un seul processeur (le maître) exécute le programme et diffuse la même instruction à tous les processeurs de calcul appelés PE pour (*Processing Element*). Cette instruction est appliquée aux données réparties dans les mémoires locales des PE, telle qu'illustrée dans la figure 24.

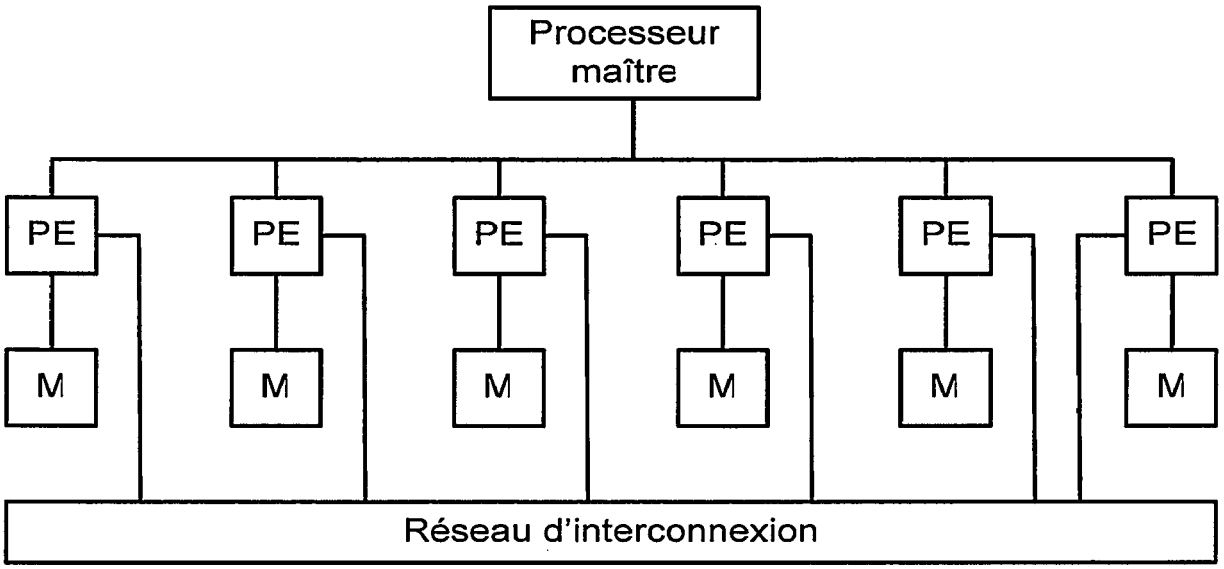


Figure 24. *SIMD à mémoire distribuée*

Avec les architectures SIMD, les modèles de programmation sont faciles à mettre en œuvres. En effet, il n'y a pas une dépendance entre les données puisque la même opération est effectuée par chaque processeur sur des données différentes. Cela permet d'éviter le contrôle des accès réalisés par les processeurs à ces données. En plus de cet avantage, ces architectures offrent un rapport coût/performances très important avec le parallélisme de données. Or, ce type de parallélisme n'est pas toujours compatible avec tous les problèmes. Nous citons l'exemple de l'application JPEG-2000 (Adams, 2001). C'est un format standard de compression d'images. Le fonctionnement de cette application suit le même schéma en deux phases. La première partie (du prétraitement à la décomposition en ondelettes) est systématique. C'est au niveau la deuxième partie de l'encodage qu'apparaissent des traitements irréguliers (quantification, deux étages d'encodage) (Philippe, 2005). Un traitement est dit irrégulier si les données sont traitées de manières différentes. Pour ce type de traitement, le parallélisme de contrôle est plus intéressant à exploiter.

2.2. *Parallélisme de contrôle*

Avec ce type de parallélisme appelé encore « parallélisme de tâches », plusieurs flots d'instructions associés à plusieurs processeurs traitent en parallèle un ou plusieurs flots de données. Nous parlons également, du modèle MIMD (*Multiple Instruction stream Multiple Data stream*). Ce modèle, contrairement au modèle SIMD, permet d'exploiter le parallélisme de tout type de traitement et plus précisément les traitements irréguliers.

Dans la suite de ce chapitre, nous focalisons l'étude des architectures MIMD à mémoire partagée. En effet, ce type d'architecture est caractérisé par une souplesse totale dans l'exploitation du parallélisme dans l'application. Il s'agit ainsi, d'appliquer plusieurs tâches de natures différentes sur plusieurs flots de données au même moment. À l'inverse de l'architecture SIMD, avec une architecture de type MIMD à mémoire partagée (figure 25), les processeurs interagissent entre eux en lisant et écrivant des variables partagées. Il est donc obligatoire d'employer des primitives de coordination et de synchronisation pour gérer cette compétition.

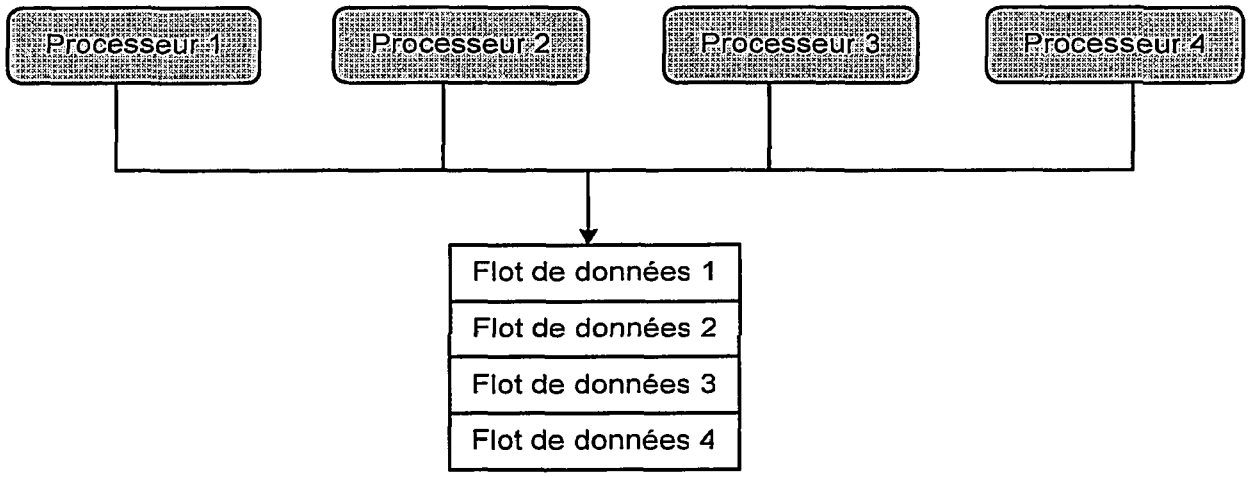


Figure 25. *Modèle de parallélisation avec une architecture MIMD à mémoire partagée*

En conclusion, nous illustrons à travers le tableau 8 les points majeurs faisant la différence entre le modèle de programmation à parallélisme de données et celui du parallélisme de contrôle. Nous distinguons deux types de parallélismes de contrôle : le modèle producteurs/consommateurs et le modèle décomposition de tâches.

Tableau 8. *Comparaison entre les modèles de programmation parallèle*

Modèle de programmation parallèle	Structure associée aux processeurs	Architecture(s) compatible(s)
Parallélisme de données	Données	SIMD, MIMD
Parallélisme de contrôle	Tâches	MIMD

2.2.1. *Producteur-consommateur*

Avec ce modèle de programmation, l'application est répartie en une séquence de traitement réalisée sur un flot de données. Ce type de parallélisme est appelé encore « Parallélisme de flux ». Tel que présenté par la figure 26, il s'agit d'un ensemble de tâches (T) associées à un ensemble de processeurs (Proc) qui communiquent entre eux de façon unidirectionnelle, généralement organisés sous forme de pipeline.

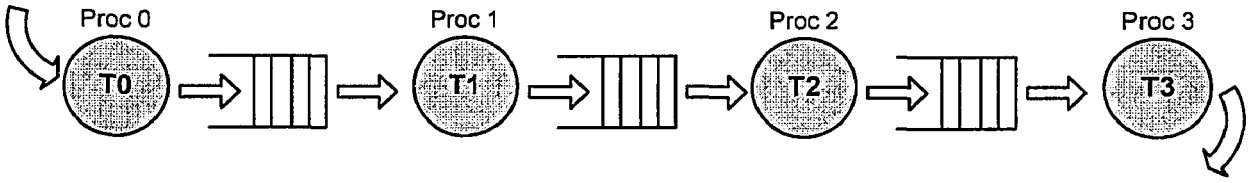


Figure 26. *Modèle de programmation avec Producteur/consommateur*

Cette structure se retrouve fréquemment dans les applications de type traitement de flux vidéo, telles que les applications MPEG (*Moving Picture Experts Group*) et JPEG (*Joint Photographic Experts Group*).

2.2.2. *Décomposition de tâches*

Avec ce type de parallélisme, l'application est décomposée en un ensemble de tâches concurrentes. Ces tâches se différent selon la quantité du travail à réaliser. La mise en œuvre de ce modèle de programmation est réalisée en attribuant au mieux les tâches aux processeurs disponibles, tel que illustré par la figure 27.

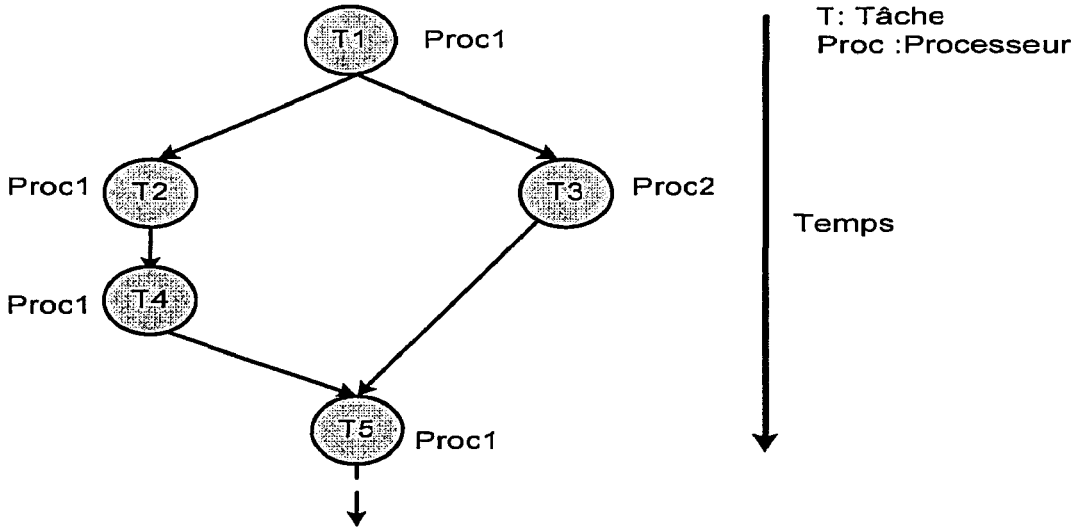


Figure 27. *Modèle de programmation avec décomposition de tâches*

Comme nous l'avons déjà indiqué précédemment, nous nous intéressons dans cette

thèse à l'étude de la parallélisation des applications sur des architectures MIMD à mémoire partagée. En effet, pour ce type d'architecture la cohérence des données dans les caches représente l'un des problèmes les plus essentiels. Ainsi, pour une bonne exploitation du parallélisme dans de telles architectures, il est nécessaire que le protocole de cohérence convienne aux modèles d'accès aux données partagées présents dans l'application.

En conclusion, dans cette section nous avons présenté en se basant sur la classification de Flynn (Flynn, 1966), trois modèles de programmation parallèles : le modèle de programmation basé sur le parallélisme de données, le modèle producteur-consommateur et le modèle de décomposition de tâches. Ces trois modèles de programmation se différent par la façon d'accès aux données partagées. Par suite, différents modèles d'accès aux données partagées sont issus de ces modèles de programmation parallèle. En conséquence, afin de bien évaluer les protocoles de cohérence des données dans les caches, il s'avère indispensable de passer de la classification de Flynn (Flynn, 1966) à une classification plus détaillée. Il s'agit de classifier les données partagées selon leurs modèles d'accès. Dans ce contexte, plusieurs classifications des données partagées sont apparues dans la littérature. L'objectif de la prochaine section consiste à étudier ces classifications et à préciser celles que nous avons utilisées dans cette thèse.

3. Classifications des données partagées

Dans la littérature, plusieurs classifications des données partagées entre des tâches parallèles sont proposées. Ces classifications sont basées sur l'analyse des accès réalisés sur chaque donnée partagée par les différents processeurs. Il s'agit de rassembler un ensemble d'informations sur la nature des accès réalisés sur cette donnée. Comme critère de classification nous citons :

- Le nombre des processeurs accédant à la donnée partagée.
- Le type de partage de la donnée : partage en lecture ou/et en écriture.

Parmi les classifications les plus connues, nous citons celle de Gupta et Weber (Anoop *et al.*, 1992) (Wolf-Dietrich *et al.*, 1989). En se basant sur leur classification, une donnée partagée peut appartenir à l'une de ces cinq classes :

- *Read-only shared data* : c'est une donnée utilisée par différents processeurs en lecture seulement.

- *Migratory shared data* : c'est une donnée qui est caractérisée par des accès exclusifs en écriture et en lecture par différents processeurs. Par la suite, la donnée, à un moment donné, est utilisée par un seul processeur. Le protocole par invalidation est plus adéquat avec ce type de données partagées.
- *Synchronization shared data* : c'est une donnée qui correspond aux primitives de synchronisations (barrière, locks).
- *Mostly-read shared data* : c'est une donnée utilisée la plupart du temps en lecture plus qu'en écriture.
- *Frequently read-written shared data* : c'est une donnée qui est utilisée en lecture et en écriture plusieurs fois par plusieurs processeurs.

Une autre classification est proposée par les auteurs de (Matts, 1995). Leur approche consiste à classifier les données partagées en prenant en compte, en premier lieu, le type de partage de la donnée (partage en lecture ou/et en écriture). En second lieu, ils prennent en compte le nombre de processeurs qui partage la donnée. Ainsi, une donnée partagée peut être l'une de ces trois classes : *Exclusive*, *Shared-by-few* et *Shared-by-many*. Dans (Sinisa *et al.*, 1997) les auteurs classifient les données partagées en se basant sur le nombre de processeurs qui réalisent les lectures et les écritures au bloc de données. Par la suite, les classes possibles pour une donnée partagée sont les suivantes : *Single Reader Single Writer* (SRSW), *Multiple Reader* (MR), *Multiple Reader Single Writer* (MRSW), *Multiple Writer* (MW), *Single Reader Multiple Writer* (SRMW) et *Multiple Reader Multiple Writer* (MRMW).

Dans cette thèse, nous adoptons les classes de données partagées *Mostly-read shared data* et *Frequently read-written shared data* de Gupta et Weber (Anoop *et al.*, 1992)(Wolf-Dietrich *et al.*, 1989). Ainsi, dans la suite de ce travail nous considérons qu'une donnée partagée peut être l'une de ces deux classes.

- *Mostly-read shared data* : cette classe de données partagées correspond aux données qui sont utilisées la plupart du temps en lecture plus qu'en écriture. Ce type de données partagées peut être présent avec le modèle de programmation basé sur le parallélisme de données. Or, comme nous l'avons déjà prouvé dans le chapitre 3 avec l'application FFT, pour ce type de données partagées le protocole par invalidation fournit de meilleurs résultats.
- *Frequently read-written shared data* : cette classe de données partagées correspond aux données qui sont utilisées en lecture et en écriture plusieurs

fois par plusieurs processeurs. Dans ce cas, tous les processeurs se coopèrent pour le calcul du résultat final pour une application donnée. Ce type de données partagées peut être présent dans les modèles de programmation avec décomposition de tâches. Pour ce type de données partagées, le protocole par mise à jour est plus intéressant que le protocole par invalidation.

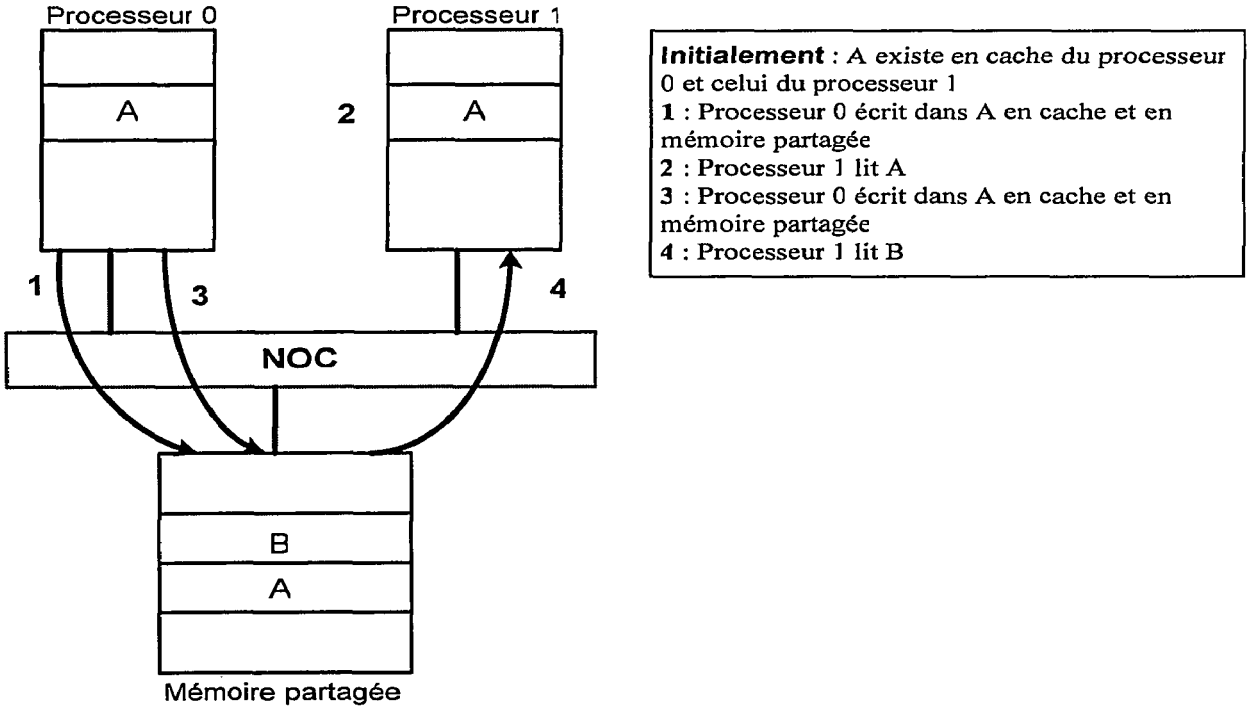


Figure 28. *Exemple de modèle de programmation producteur-consommateur*

Le tableau 9 indique pour chaque modèle de programmation parallèle, le type de données partagées qui lui correspond et le protocole de cohérence des données dans les caches le plus adéquat. Concernant le modèle producteur-consommateur, il peut contenir deux types de données partagées (Liqun *et al.*, 2007) (Abdullah *et al.*, 2010). À l'aide de l'exemple de modèle de programmation producteur-consommateur illustré par la figure 28, nous désirons analyser ces deux types de données partagées :

1. Premier type : dans la figure 28, suite à l'opération d'écriture réalisée par le processeur 0 dans « A » (opération 1 dans la figure 28), le processeur 1 réalise une opération de lecture de « A » (opération 2 dans la figure 28). Par la suite, la donnée « A » produite par le processeur 0 est immédiatement consommée par le processeur 1. Pour ne pas avoir d'échec de cache dans le processeur 1, il faut appliquer le protocole par mise à jour. Dans la suite de

notre travail, ce type de données partagées est considéré de la classe *Frequently read-written shared data.*

2. Deuxième type : dans la suite de cet exemple, le processeur 0 réalise une écriture dans « A » (opération 3 dans la figure 28), sans avoir une utilisation de cette donnée par le processeur 1. Dans ce cas, la donnée « A » qui existe encore dans le cache du processeur 1 n'est plus consommée. Ainsi, le protocole par invalidation est plus intéressant du fait qu'il élimine une mise à jour inutile de « A » dans le cache du processeur 1. Dans la suite de notre travail, ce type de données partagées est considéré de la classe *Mostly-read shared data.*

Tableau 9. *Classification des données partagées*

Modèle de programmation parallèle		**Type des données partagées**	**Protocole de cohérence de cache adéquat**
Parallélisme de contrôle	Décomposition de tâche	*Frequently read-written shared data*	Mise à jour
	Producteur-consommateur	- *Frequently read-written shared data* - *Mostly-read shared data*	Hybride
Parallélisme de données		*Mostly-read shared data*	Invalidation

En se basant sur cette classification des données partagées, nous avons parallélisé un ensemble d'applications de traitement intensif du signal et de l'image. Ce qui différent d'une application à une autre c'est la façon de parallélisation adoptée. En effet, nous avons parallélisé ces applications de façon à avoir différents comportement et différents modèles d'accès aux données partagées. De ce fait, les performances du protocole proposé sont évaluées avec différents scénarios. L'étude de ces scénarios est l'objectif de la prochaine section. Concernant les résultats de l'évaluation du protocole proposé, ils seront présentés dans le prochain chapitre.

4. Parallélisation des applications

Dans une architecture MPSoC, l'efficacité du protocole de cohérence des données

dans les caches dépend essentiellement des motifs d'accès présents dans l'application. Comme les applications des systèmes embarqués sont de plus en plus complexes, il est possible de trouver différents modèles de parallélisation dans une même application. Afin de bien évaluer notre protocole, nous avons parallélisé un ensemble d'applications comportant des motifs d'accès aux données différents. Ceci nous permet de tester notre protocole dans différents scénarios.

Les applications choisies sont : la multiplication de matrices (MM), la compression d'image (JPEG), la transformée de Fourier Rapide ou FFT pour (*Fast Fourier Transform*) et la décomposition de matrices (LU). Ces applications sont largement utilisées par la communauté pour mesurer les performances des architectures. Il faut noter que MM et LU font partie du benchmark SPLASH (SPLASH) et que FFT et JPEG font partie du benchmark Mibench (Matthew *et al.*, 2001). Le tableau 10 de la page 97 présente le type de parallélisme (de données ou de tâches) qui a été utilisé pour implémenter ces applications sur notre simulateur de MPSoC.

4.1. *Application de Multiplication de matrices (MM)*

Le programme ci-dessous présente l'algorithme de multiplication de deux matrices. Dans cet algorithme, A et B représentent les matrices d'entrées. Elles contiennent (respectivement) m lignes et n colonnes (respectivement n lignes et p colonnes). La matrice résultat de la multiplication de A par B est la matrice C, qui contient m lignes et p colonnes.

```
for (i=0; i<m; i++) // Boucle1 (Boucle externe)
{   for (j=0; j<p; j++) ++)
    {       for (k=0; k<n; k++) // Boucle2 (Boucle interne)
            {
                    C[i][j]=C[i][j]+A[i][k]*B[k][j];
            }
    }
}
```

Il existe plusieurs façons avec lesquelles l'application MM peut être réalisée en parallèle par plusieurs processeurs. Dans notre travail, nous présentons deux manières de parallélisation de l'algorithme de MM. Afin de simplifier la présentation de ces deux méthodes, nous assumons que le traitement est réalisé avec deux processeurs (Processeur 0 et Processeur 1). Ainsi, ces deux approches de parallélisation correspondent à deux classes de données partagées entre les processeurs.

- Une première manière de parallélisation de l'algorithme de MM consiste à calculer chaque élément de la matrice C par le Processeur 0 et le Processeur 1. Ceci se réalise en découpant la boucle interne (boucle2) entre le Processeur 0 et le Processeur 1, tel que illustré par la figure 29. Une explication de cette méthode est présentée également par la figure 30. Par la suite, chaque bloc de la mémoire partagée est lu et écrit plusieurs fois par chaque processeur. Avec cette méthode de parallélisation, les blocs mémoire de la matrice C sont de type *Frequently read-written shared data*. Or, pour ce type de données partagées le protocole par mise à jour fournit les meilleures performances.

Travail réalisé par Processeur 0	**Travail réalisé par Processeur 1**

for (i=0; i<m; i++)
{for (j=0; j<p; j++)
{for (k=0; k<n/2; k++) *// Boucle2*
{
C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
}
}
	for (i=0; i<m; i++)
	{for (j=0; j<p; j++)
	{for (k=n/2; k<n; k++) *// Boucle2*
	{
	C[i][j]=C[i][j]+A[i][k]*B[k][j];
	}
	}
	}

Figure 29. *Alfgorithme de multiplication de deux matrices avec la première méthode*

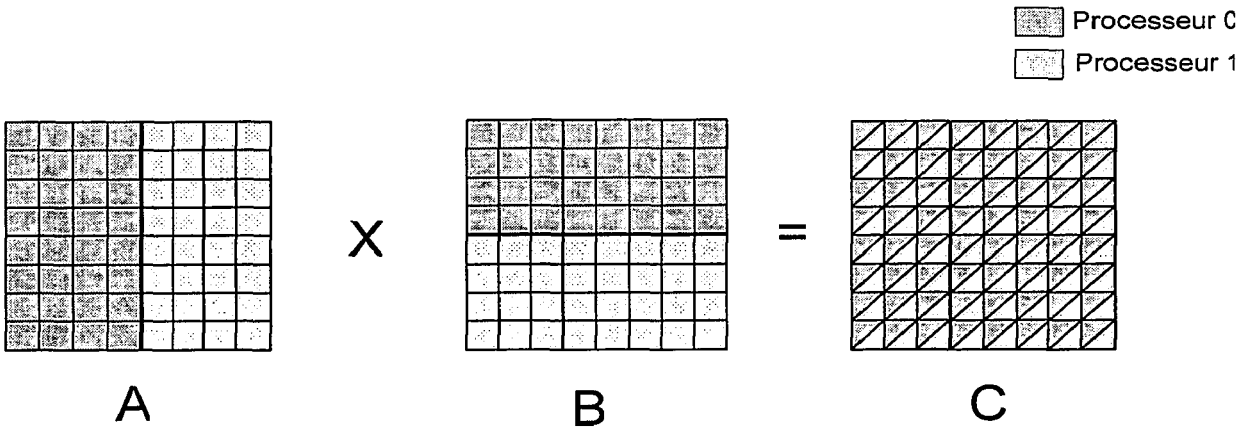


Figure 30. *Première méthode de parallélisation de l'algorithme MM*

- Une deuxième manière de parallélisation de l'algorithme de MM consiste à découper la boucle externe (Boucle 1) entre Processeur 0 et Processeur 1. L'algorithme séquentiel devient alors, comme illustré dans la figure 31. De cette manière, chaque processeur réalise le calcul pour une partie indépendante des éléments de la matrice C. Comme le montre la figure 32, le processeur 0 calcule les éléments de la partie supérieure de la matrice C, tandis que le processeur 1 calcule les éléments de la partie inférieure de C. Ainsi, les blocs mémoires de la matrice C sont de type *Mostly-read shared data*. Cette classe de données partagées favorise l'application du protocole par invalidation.

Travail réalisé par Processeur 0	**Travail réalisé par Processeur 1**

for (i=0; i<m/2; i++) // *Boucle1*
{ for (j=0; j<p; j++)
{ for (k=0; k<n; k++)
{
C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
}
}
	for (i=m/2; i<m; i++) // *Boucle1*
	{ for (j=0; j<p; j++)
	{ for (k=0; k<n; k++)
	{
	C[i][j]=C[i][j]+A[i][k]*B[k][j];
	}
	}
	}

Figure 31. *Algorithme de multiplication de deux matrices avec la deuxième méthode*

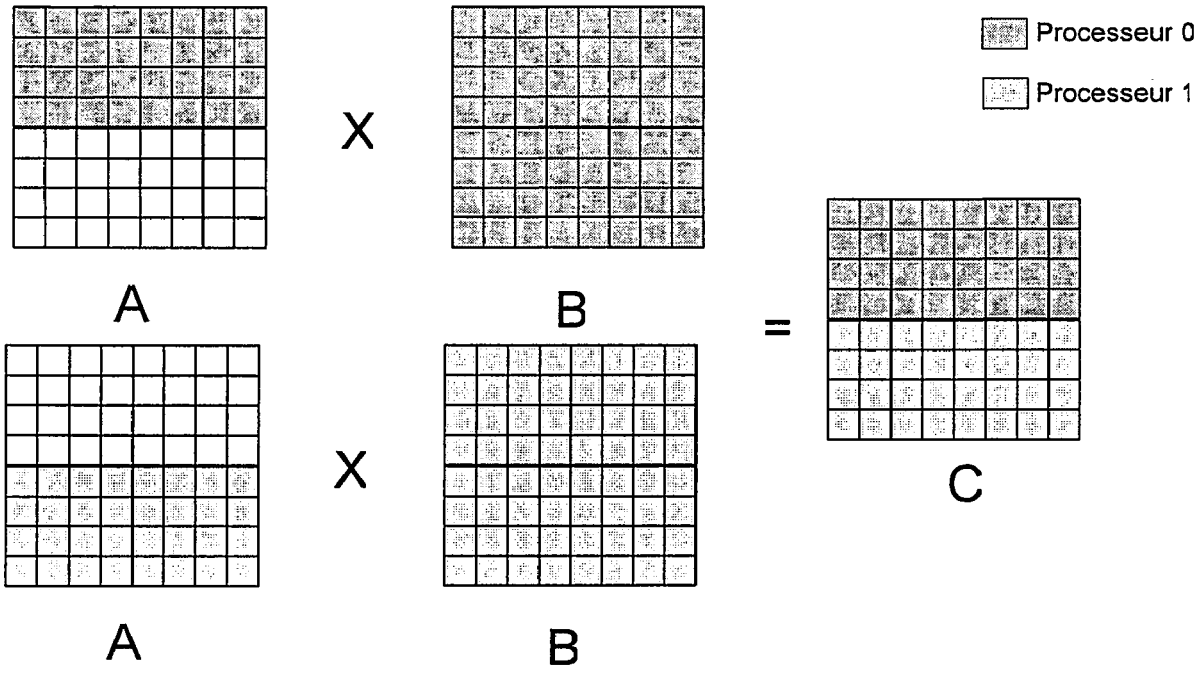


Figure 32. *Deuxième méthode de parallélisation de l'algorithme MM*

Dans le prochain chapitre, nous présenterons les résultats expérimentaux pour une version de l'application MM qui combine les deux méthodes de parallélisation précédentes (figure 33). En effet, nous avons parallélisé l'application MM sur une architecture MPSoC qui contient 4 processeurs. Avec les deux premiers processeurs, nous avons utilisé la première méthode de parallélisation. Avec les deux autres processeurs, nous avons utilisé la deuxième méthode de parallélisation. En conséquence, dans une même application, nous avons différents modèles d'accès aux données partagées.

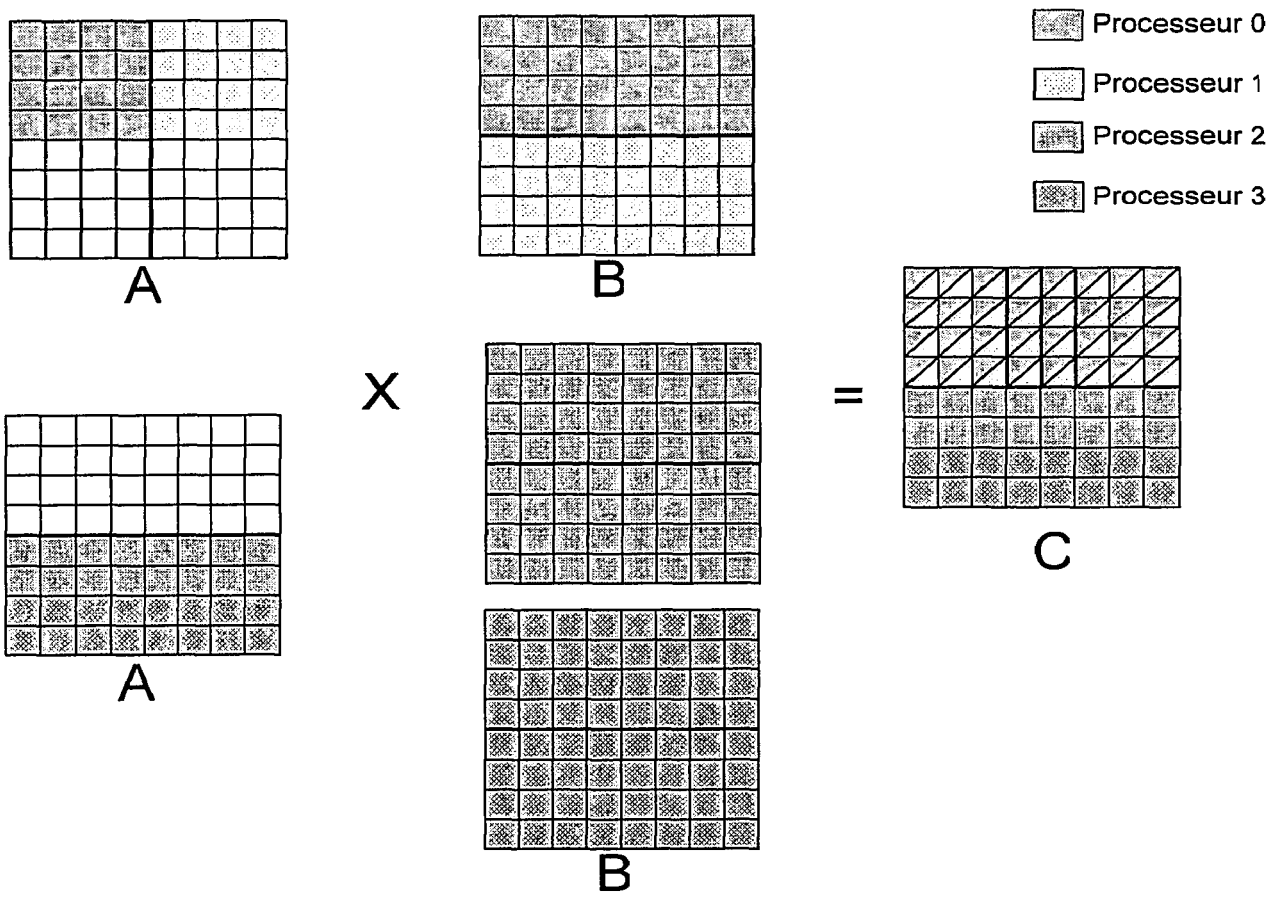


Figure 33. *Méthode de parallélisation hybride de l'algorithme MM*

4.2. *Application JPEG*

L'application JPEG (*Joint Photographic Experts Group*), est une application largement employée dans la compression d'images. Comme le montre la figure 34, cette application contient quatre tâches qui sont dans cet ordre :

- Conversion des couleurs des pixels du format RVB ou (RGB en anglais) au format luminance/chrominance (une luminance (Y) et deux chrominances (UV)).
- La transformée en cosinus discrète (TCD) ou en anglais DCT (*Discrete Cosine Transform*) est appliquée à chaque chrominance.
- La quantification pour chaque chrominance.
- Le codage Huffman pour chaque chrominance.

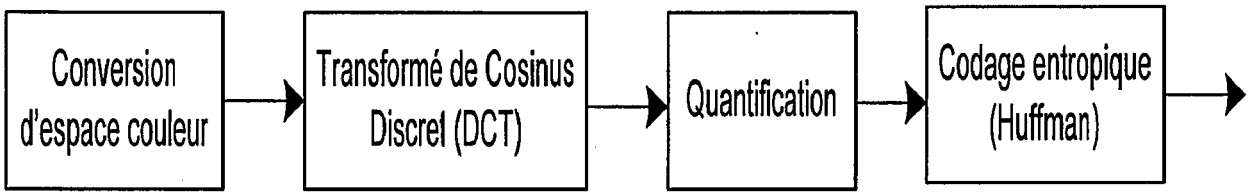


Figure 34. *Processus de compression d'images avec la norme JPEG*

Nous avons parallélisé l'application JPEG sous forme de pipeline. En appliquant cette méthode de parallélisation sur une architecture MPSoC contenant 4 processeurs, alors chaque processeur réalise l'exécution d'une seule tâche parmi les 4 tâches. Ceci est expliqué également par la figure 35. Notons ainsi, que ce schéma correspond au modèle de programmation producteur-consommateur. Comme déjà indiqué dans la section précédente, ce modèle peut contenir les deux classes de données partagées : *Mostly-read shared data* et *Frequently read-written shared data*. De ce fait, avec cette application, le protocole hybride est plus efficace que les autres protocoles.

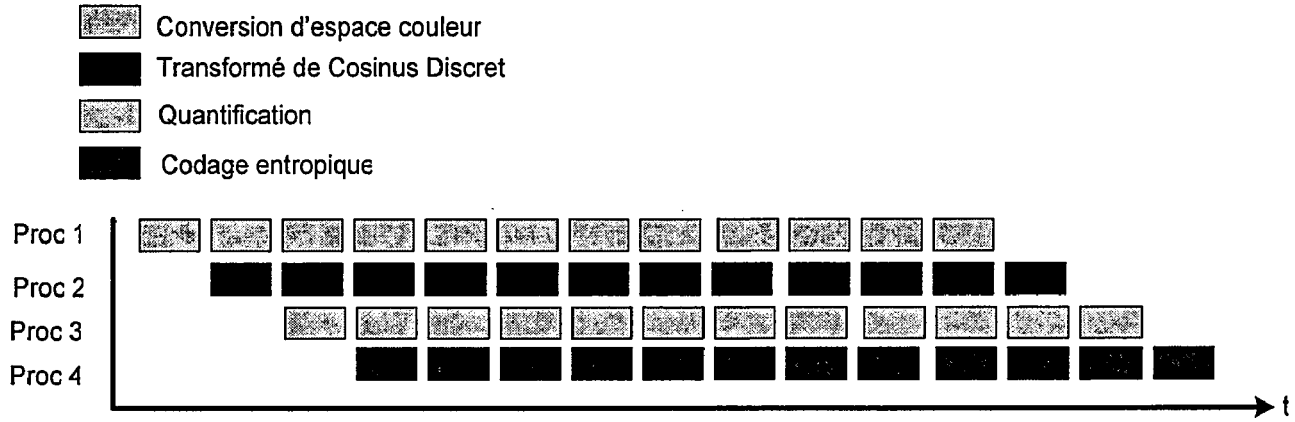


Figure 35. *Modèle de parallélisation de l'application JPEG*

4.3. *Application FFT*

L'application *Fast Fourier Transform* (FFT) est l'une des applications les plus utilisées dans le domaine de traitement de signal numérique. Elle permet d'exprimer un signal temporel dans le domaine fréquentiel. Le traitement de l'application FFT contient deux phases :

- Phase d'initialisation de matrice : consiste à remplir une matrice par les échantillons du signal temporel. Nous avons parallélisé cette partie du traitement de façon qu'elle soit réalisée par tous les processeurs en même temps. Par la suite durant cette phase

les blocs mémoire sont lus et écrits plusieurs fois par chaque processeur. Ces blocs sont alors de type *Frequently read-written shared data*.

- Phase de calcul de la FFT : après remplissage, les échantillons du signal temporel sont ensuite passés dans un algorithme de calcul FFT. Cette phase est parallélisée de façon que tous les processeurs réalisent le même calcul mais sur des parties différentes de la matrice. De cette façon, les blocs mémoire sont de type *Mostly-read shared data*.

4.4. *Application décomposition de matrices (LU)*

Le programme ci-dessous représente l'algorithme de l'application de décomposition de matrices nommée (LU). Dans cet algorithme, A est une matrice inversible et décomposable en un produit d'une matrice triangulaire inférieure L (pour *Low*) et une matrice triangulaire supérieure U (pour *Up*). Nous avons parallélisé cette application en divisant les deux boucles externes (Boucle 1 et Boucle 2) entre les processeurs. Par conséquent, les blocs mémoire partagés sont de type *Mostly-read shared data*. Or pour ce schéma de parallélisation, le protocole par invalidation est plus efficace.

```
for (i = 0; i<n; i++)   // Boucle1
 { for (j = 0; j<n; j++)  // Boucle2
    {     s = A[i][j];
          for (k = 1; k<min (i,j) -1; k++) // Boucle3
           {
                 s = s – L[i][k] * U[k][j];
           }
          if(i>j){
                 L[i][k] = s/U[j][j];
               }
          else {
                 U[i][j] = s ;
               }
    }
}
```

Finalement, le tableau 10 récapitule le travail réalisé par cette section. En effet, ce tableau représente pour chaque application le type de parallélisme employé avec elle et le type de données partagées présent ainsi dans l'application. Nous illustrons aussi dans ce tableau, le protocole de cohérence des données dans les caches adéquat avec chaque application.

Tableau 10. *Profilage de quelques applications parallèles*

Applications	**Type de parallélisme**	**Type de données partagées**	**Protocole de cohérence adéquat**
MM (Méthode 1)	Décomposition de tâche	*Frequently read-written shared data*	Mise à jour
MM (Méthode 2)	Parallélisme de données	*Mostly-read shared data*	Invalidation
JPEG	Producteur-consommateur	- *Frequently read-written shared data* - *Mostly-read shared data*	Hybride
FFT	-Décomposition de tâches -Parallélisme de données	- *Frequently read-written shared data* - *Mostly-read shared data*	Hybride
LU	Parallélisme de données	*Mostly-read shared data*	Invalidation

5. Conclusion

Dans ce chapitre, nous avons étudié les différentes formes permettant l'exploitation du parallélisme dans une application donnée. Ensuite, nous avons présenté la classification des données partagées que nous avons adoptées dans cette thèse afin d'évaluer notre protocole de cohérence des données dans les caches. Nous avons ainsi précisé pour chaque modèle de programmation parallèle la classe de données partagées qui lui correspond, et éventuellement le protocole de cohérence des données dans les caches qui permet une bonne

exploitation de ce modèle de programmation parallèle. Finalement, nous avons parallélisé quatre applications de traitements de signal intensif (MM, JPEG, FFT et LU), de façon à avoir différents modèles d'accès aux données partagées. Ce qui va nous permettre d'évaluer le protocole hybride proposé avec une variété de modèles de programmation parallèle.

Dans le prochain chapitre nous présenterons pour chacune de ces applications, une comparaison des performances entre les protocoles de cohérence des données dans les caches : hybride, invalidation et celui par mise à jour.

Chapitre 6

Environnement de simulation et résultats expérimentaux

1. Introduction

Dans ce chapitre nous allons présenter une évaluation des performances du protocole proposé dans cette thèse. Afin d'atteindre cet objectif, nous avons divisé le chapitre comme suit: dans un premier temps, nous présentons dans la section 2 l'environnement de simulation que nous avons utilisé afin d'évaluer le protocole proposé. Dans un second temps et dans la section 3, nous réalisons sur la base des résultats expérimentaux une comparaison des performances entre trois protocoles de cohérence de caches : le protocole hybride proposé, le protocole par invalidation uniquement et le protocole par mise à jour uniquement sur l'ensemble des applications (benchmark) présentées dans le chapitre précédent. Ensuite, dans la section 4, nous mettons l'accent sur les capacités d'extensibilité du protocole proposé en termes de nombre de processeurs. Finalement, dans la section 5, nous réalisons une estimation des coûts supplémentaires de ce protocole en termes de surface. Nous proposons enfin une solution qui permet de réduire ces coûts tout en garantissant un niveau relativement élevé de performances.

2. Environnement de simulation

L'objectif principal de cette section consiste à présenter l'environnement de simulation que nous avons utilisé afin d'évaluer les performances du protocole hybride proposé pour la gestion de la cohérence des données dans les caches ainsi que l'architecture proposée pour l'implémenter. Pour cela, nous présentons en premier lieu la plateforme de simulation que nous avons utilisée. En second lieu, nous expliquons la solution utilisée par cette plateforme pour maintenir la cohérence des données dans les caches.

2.1. *Plateforme de simulation*

Comme plateforme de simulation, nous avons choisi la plateforme SoCLib (SOCLIB). C'est une bibliothèque de composants matériels qui permet de réaliser un simulateur de MPSoC au niveau cycle-précis bit-précis ou CABA pour *Cycle Accurate Bit Accurate*. C'est un niveau d'abstraction qui permet de modéliser et de simuler le comportement des composants au cycle précis. Le système est alors décrit de façon précise d'un point de vue temps d'exécution des opérations. Cette description détaillée au niveau cycle par cycle permet d'améliorer la précision de l'estimation des performances du protocole proposé. Des études ont démontré que ce niveau permet d'obtenir une précision de l'ordre de 2% ou 3% par rapport à l'implémentation matérielle ou physique.

Les composants de cette plateforme sont écrits en SystemC (SYSTEMC). Ce dernier est un langage de modélisation composé de librairies de classes C++ et d'un noyau de simulation (nommé aussi moteur de simulation). Par rapport au langage C++ de base, SystemC a été étendu afin de supporter la description du matériel et intègre les notions comme la concurrence et la notion du temps.

La plateforme SoCLib permet, par instanciation de composants matériels, de réaliser une architecture MPSoC (figure 36). La version de SoCLib que nous avons utilisée contient entre-autre un processeur RISC, qui est le MIPS R3000. Le composant cache de SoCLib, nommé Xcache, est de type SRAM (*Static Random Access Memory*). Ce composant contient deux mémoires caches indépendantes : un cache pour les instructions et un cache pour les données. Cette plateforme contient également une mémoire d'instructions et une autre mémoire pour les données de type SRAM. En effet, ce type de mémoire exige plus d'espace (4 fois plus grand) sur la puce par rapport à la technologie DRAM (*Dynamic Random Access Memory*) mais il offre une simplicité d'intégration et un temps d'accès plus court.

Il y a deux types de composants dans la plateforme SoCLib : les initiateurs (I) et les cibles (T). À titre d'exemple, les processeurs sont des initiateurs et les bancs mémoires sont des cibles. En termes de réseau d'interconnexion, cette plateforme possède deux types de réseaux d'interconnexion : le crossbar et le DSPIN (*Distributed Scalable Integrated Network*) (Panades *et al.,* 2009). Le réseau DSPIN a une topologie en grille à deux dimensions permettant de relier un nombre important de composants avec un coût relativement faible par rapport au crossbar. Durant les expérimentations, nous avons utilisé le crossbar pour interconnecter les différents composants. Néanmoins, le protocole hybride que nous proposons est indépendant de la nature du réseau d'interconnexion.

Cette plateforme dispose aussi d'un composant matériel pour la gestion des verrous nommé « *Locks Engine* ». Ce dispositif permet de garantir la synchronisation entre les processeurs. Enfin, les composants que nous venons de présenter sont interconnectés par un bus et un protocole de communication qui respectent le standard VCI (*Virtual Component Interface*) (Bunker *et al.,* 2001). Nous notons également que la plateforme que nous avons utilisée contient des modèles d'estimation de la consommation statique et dynamique de l'énergie pour les différents composants de l'architectures (processeurs, caches, mémoires, réseaux d'interconnexion, etc.). Ces modèles sont détaillés dans (Ben Atitallah *et al.,* 2006).

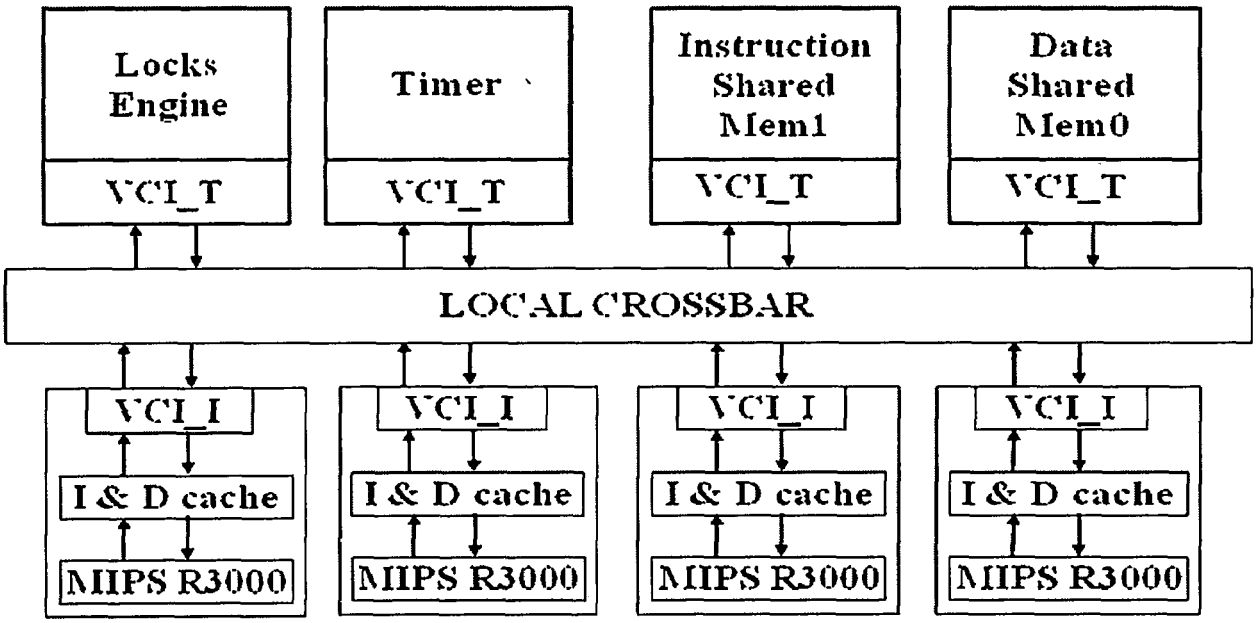


Figure 36. *Exemple d'architecture MPSoC*

Dans ce chapitre, nous supposons que le nombre de bancs mémoire est de 4. Pour exploiter cette architecture multi-bancs, nous avons parallélisé les applications de façon à avoir des accès parallèles aux différents bancs de mémoire.

2.2. *Cohérence des données dans les caches avec SoCLIB*

Afin de maintenir la cohérence des données en caches, la plateforme SoCLib utilise la solution appelée « Sans Cohérence » (Zandvelt, 1998). En effet, à partir d'indications fournies par le programmeur, les données sont réparties en deux groupes : des données locales (nommées aussi privées) à un processeur et les données partagées entre les processeurs. Seules les données locales peuvent être stockées dans la mémoire cache. Nous dirons alors que ces données sont « cachables ». À l'opposé, les données partagées entre les processeurs ne peuvent pas être chargées dans les mémoires caches. En plus de la surcharge qu'elle provoque sur le programmeur, cette solution ne permet pas une utilisation efficace des mémoires caches et augmente le temps d'exécution et l'énergie consommée. La première étape de notre travail, a été d'enrichir SoCLib par des mécanismes de gestion de cohérence « par mise-à-jour » et « par invalidation ».

3. Performances du protocole hybride

Dans cette section nous présentons les performances du protocole hybride proposé dans cette thèse. Il s'agit plus précisément d'une comparaison des performances des trois

protocoles de cohérence des données dans les caches basés sur le mécanisme du répertoire : le protocole hybride proposé dans cette thèse (HYB), le protocole par invalidation (INV) et le protocole par mise à jour (MAJ). Cette comparaison est réalisée en termes de nombre total d'échecs de cache, du temps d'exécution en cycles horloge et de consommation d'énergie en mJoule.

3.1. *Résultats expérimentaux pour l'application de multiplication de matrices (MM)*

Dans cette section, nous présentons les résultats expérimentaux pour la MM de taille (256 x 256). La parallélisation de la MM a été présentée dans la section 4 dans le chapitre 5.

3.1.1. *Réduction du nombre d'échecs de cache*

La figure 37 représente le nombre d'échecs de cache en fonction de la taille du cache de données dans les différents processeurs. Comme le montre cette figure, le protocole proposé permet de réduire le nombre total d'échecs de cache par rapport au protocole par invalidation. Cette réduction atteint 77% pour des caches de données de taille 32 KO (Kilo Octets). En effet, comme nous l'avons indiqué dans le chapitre précédent, cette application est parallélisée de façon à avoir deux types de données partagées : *Frequently read-written shared data* et *Mostly-read shared data*. Or, avec le protocole par invalidation les données partagées de type *Frequently read-written shared data* sont fréquemment invalidés inutilement. Ceci, entraîne l'augmentation du nombre d'échecs de cache. Par rapport au protocole par mise à jour, ce protocole réalise un nombre d'échecs de cache plus réduit. Ceci s'explique par le fait qu'il réalise toujours des mises à jour aux données partagées modifiées.

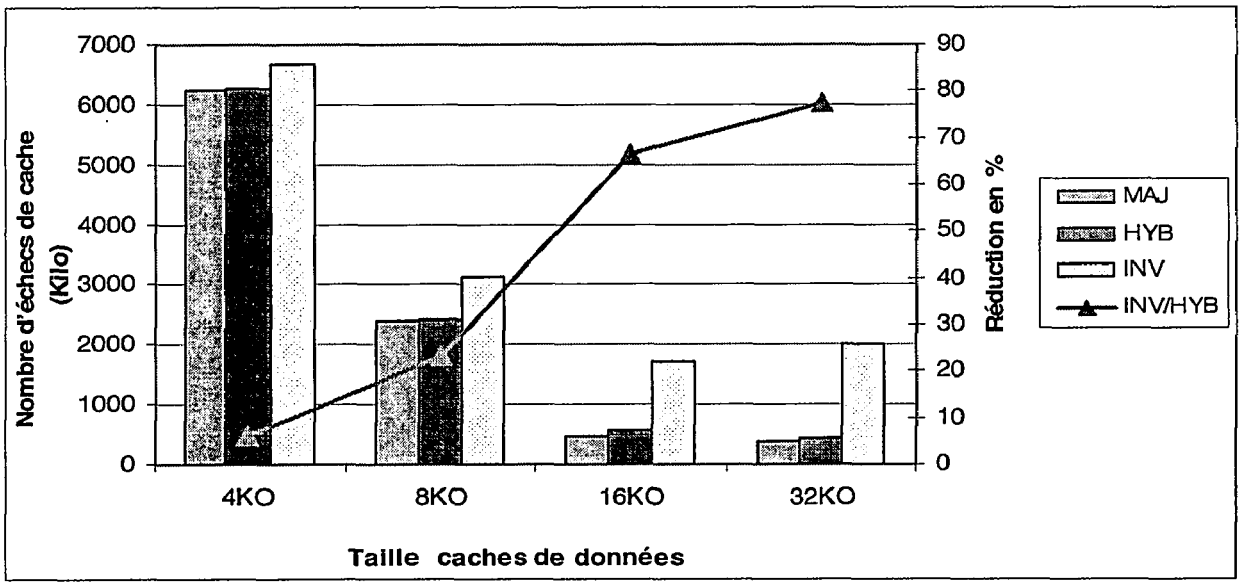


Figure 37. *Nombre d'échecs de cache (*10^3) en fonction de la taille de cache de données avec l'application MM pour un MPSoC de 4 processeurs*

3.1.2. *Évaluation du temps d'exécution dans les 3 protocoles*

Nous constatons d'après la figure 38 que le protocole proposé réduit le temps d'exécution par rapport au protocole par invalidation d'un facteur de 20% pour des caches de données de taille 32 KO. En effet, la réduction du nombre d'échecs dans les accès aux caches par le protocole hybride entraîne la réduction du temps d'exécution.

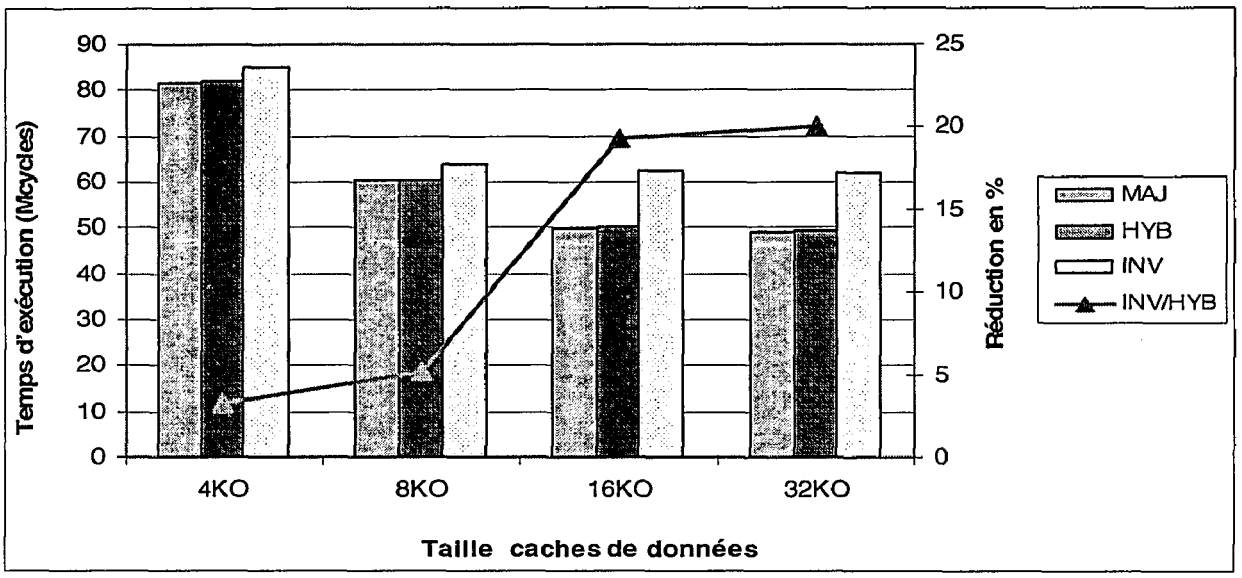


Figure 38. *Temps d'exécution en fonction de la taille de cache de données avec l'application MM pour un MPSoC de 4 processeurs*

En comparaison avec le protocole par mise à jour, nous remarquons que le protocole hybride ne réduit pas le temps d'exécution total. Ceci vient du fait que les protocoles, hybride et par mise à jour, réalisent le même nombre d'échecs en cache. En conséquence, le trafic de données dans le NoC est moins important avec ces protocoles qu'avec le protocole par invalidation. Cependant, le protocole par mise à jour réalise dans certains cas les opérations de mises à jour inutiles à travers le bus de cohérence dédié au transfert des messages de cohérence.

3.1.3. *Évaluation de la consommation d'énergie dans les 3 protocoles*

La figure 39 présente la consommation d'énergie pour les 5 composants principaux de l'architecture MPSoC. Ces composants sont : les processeurs, les caches, le NoC, le bus de cohérence et les différents bancs mémoires partagées.

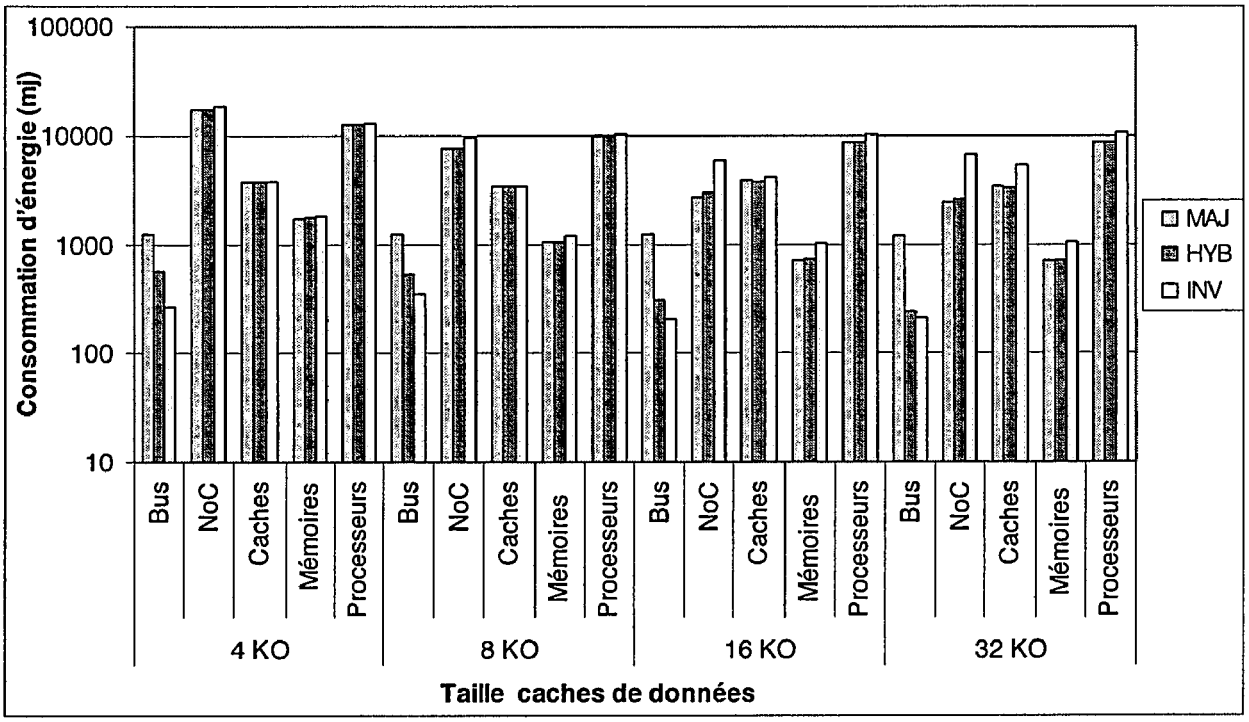


Figure 39. *Consommation d'énergie (en mJoules) des différents composants du système en fonction de la taille de cache de données avec l'application MM pour un MPSoC de 4 processeurs*

Concernant le bus et par rapport au protocole par mise à jour, nous constatons que l'énergie consommée par le bus de cohérence a été réduite grâce à l'utilisation du protocole hybride. Cette réduction peut aller jusqu'à 70% pour une taille de cache de données de 32 KO. En effet, le protocole hybride élimine les opérations de mises à jour inutiles des

données partagées de type *Mostly-read shared data* réalisées à travers le bus de cohérence.

Avec le protocole par invalidation, l'énergie consommée par le bus est moins élevée par rapport aux deux autres protocoles. Ceci vient du fait que les messages envoyés sur le bus pour le maintien de la cohérence de données avec le protocole par invalidation comportent seulement les adresses des données à invalider. Ainsi, contrairement au protocole par mise à jour qui consiste à envoyer les adresses et les nouvelles valeurs des données modifiées.

La consommation d'énergie des mémoires caches est réduite avec le protocole hybride en comparaison avec le protocole par mise à jour d'un facteur de réduction de 4% pour des caches de données de taille 32 KO. Ceci vient principalement à l'élimination des opérations d'écritures (des mises à jour inutiles) dans les caches. La valeur de réduction n'est pas assez importante puisque le protocole hybride réalise des écritures en caches lors des échecs de cache causés par l'invalidation des données partagées. Par rapport au protocole par invalidation, la consommation d'énergie des caches est bien réduite avec le protocole hybride proposé. La réduction varie de 1% pour les caches de données de petites tailles à 39% pour des caches de données de grandes tailles. En effet, avec des grandes tailles de caches, les blocs de cache résident pour plus de temps dans le cache, par suite le risque d'être invalidé devient plus important.

La consommation d'énergie du NoC est importante avec le protocole par invalidation, ceci revient au trafic de données dû aux échecs de caches causés par ce protocole. Cette valeur est bien réduite avec le protocole hybride. La réduction varie entre 4% pour des caches de données de taille 4 KO jusqu'à 60% pour une taille de cache de données de 32 KO. Avec le protocole par mise à jour, la consommation d'énergie du NoC est la plus faible par rapport aux deux autres protocoles, puisque ce protocole ne cause pas d'échecs de caches.

La consommation d'énergie des différents bancs de mémoire est aussi réduite avec le protocole hybride en comparaison avec le protocole par invalidation. Cette réduction varie de 4.5% pour une taille des caches de données de 4 KO jusqu'à 40% pour une taille des caches de données de 32KO. En effet, la réduction des échecs de caches avec le protocole hybride entraîne la réduction des opérations de lecture de ces bancs de mémoire.

Finalement, le protocole proposé réduit aussi la consommation d'énergie des différents

processeurs de l'architecture MPSoC par rapport au protocole par invalidation. Nous avons obtenu une réduction entre 2% pour des caches de données de taille 4 KO et 17% pour des caches de données de taille 32 KO. Cette réduction est introduite par la réduction du temps d'exécution. Nous notons ici, que les modèles d'estimation de la consommation d'énergie que nous avons utilisés prennent en considération la consommation statique et la consommation dynamique.

En conclusion, grâce au protocole hybride et en comparaison avec le protocole par invalidation, la consommation d'énergie totale est réduite d'un facteur de 4% pour des caches de données de taille 4 KO et jusqu'à 34% pour des caches de données de taille 32 KO (figure 40). Par rapport au protocole par mise à jour, le gain du protocole proposé est moins important et il varie de 2% pour des caches de données de taille 4 KO jusqu'à 5.5 % pour les caches de données de taille 32 KO. Ceci est dû au fait que les opérations de mises à jour inutiles réalisées par le protocole par mise à jour se font à travers le bus de cohérence au lieu du NoC. En effet, une opération de transfert de données réalisée au niveau du bus est moins coûteuse en termes de consommation d'énergie que celle réalisée au niveau du NoC. Pour confirmer ces résultats, nous avons réalisé des expérimentations qui consistent à estimer pour l'application MM les coûts en termes de temps d'exécution et consommation d'énergie, dans le cas où le bus de cohérence n'était pas utilisé (figure 41 et figure 42). Dans ce cas, le NoC est chargé de réaliser le transfert des messages de cohérence.

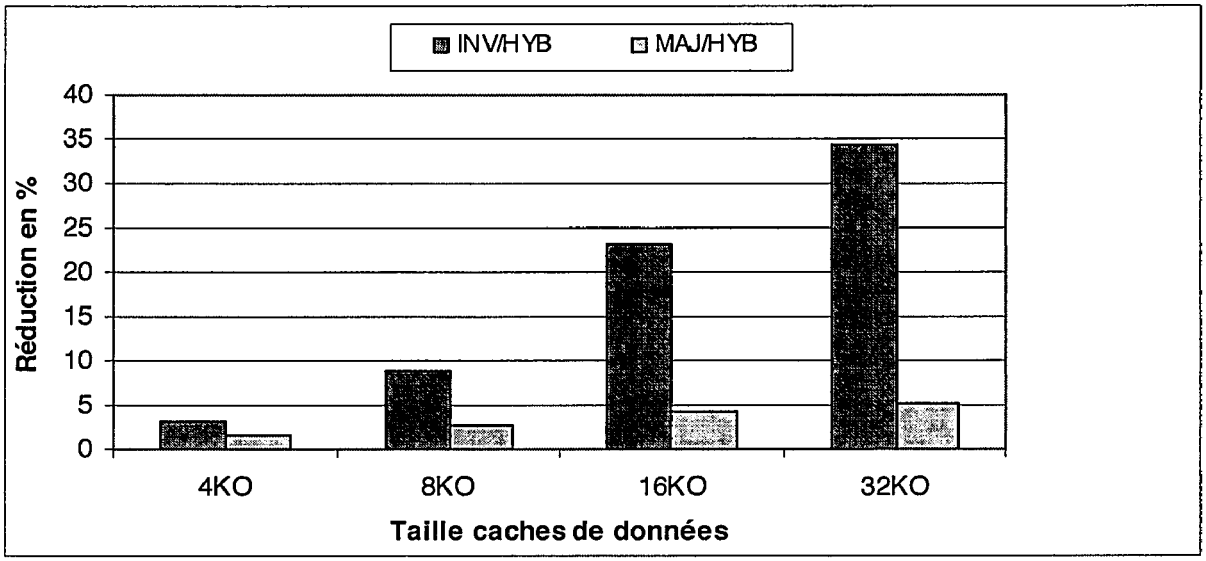


Figure 40. *Réduction de la consommation d'énergie totale avec le protocole hybride en fonction de la taille de caches de données avec l'application MM et pour un MPSoC de 4 processeurs par rapport aux 2 autres protocoles*

La figure 41 représente une comparaison de la consommation d'énergie au niveau du NoC pour deux situations. Dans la première situation, les messages de cohérence sont transférés à travers le bus de cohérence. Cette situation est désignée par (NoC avec Bus) dans la figure 41. Dans la deuxième situation, les messages de cohérence sont transférés à travers le NoC. Cette deuxième situation est désignée par (NoC sans Bus) dans la figure 41. D'après cette figure, nous constatons que l'architecture avec bus de cohérence offre une consommation d'énergie au niveau du NoC relativement réduite quelque soit le protocole utilisé. Cette réduction est plus importante avec le protocole par mise à jour. Elle atteint 27% pour une taille de cache de données de 32 KO. En comparaison avec le protocole hybride, la réduction n'est que de 6% pour des caches de données de 32 KO et 4% pour le protocole par invalidation. Nous constatons également que la réduction devient plus importante en augmentant la taille de cache. En effet, lorsque la taille de la mémoire cache augmente, le nombre d'échecs de cache et le trafic de données dans le NoC diminuent. Par conséquent, le rapport (messages de cohérence/trafic du NoC) devient plus important dans ce dernier cas.

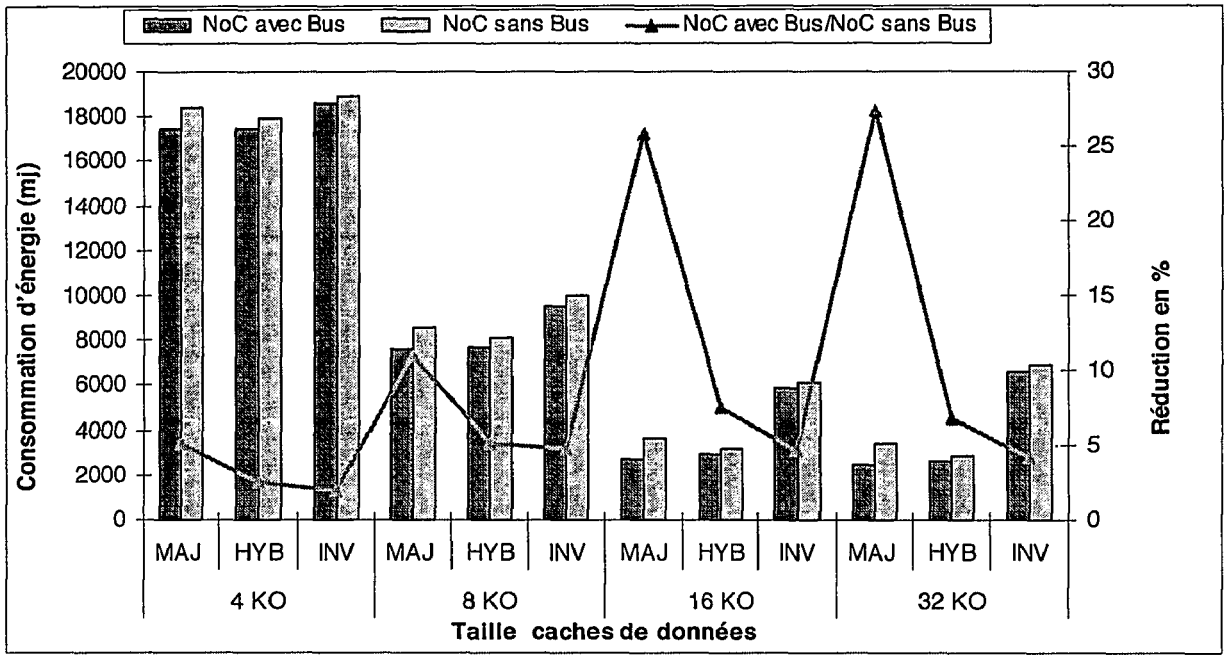


Figure 41. *Comparaison de la consommation d'énergie du système avec et sans le bus de cohérence en fonction de la taille de cache de données avec la MM pour un MPSoC de 4 processeurs*

La figure 42 montre que l'utilisation du bus de cohérence en conjonction avec le NoC pour le transfert des messages de cohérence, entraîne une réduction du temps d'exécution total. Avec le protocole par mise à jour, cette réduction varie de 5% pour des caches de données de taille 4 KO jusqu'à 27 % pour des caches de 32 KO. Nous notons que plus la taille de cache est grande plus la réduction est intéressante. Ceci vient du fait que le temps d'exécution, les échecs de cache et les accès à la mémoire partagée diminuent lorsque la taille de cache augmente. Par conséquent, l'effet des messages de cohérence sur le système devient plus important. Nous concluons d'après les figures 39 et 40 que l'utilisation du bus de cohérence avec le protocole de cohérence des données dans les caches réduit à la fois le trafic de données dans le NoC, la consommation d'énergie et le temps d'exécution.

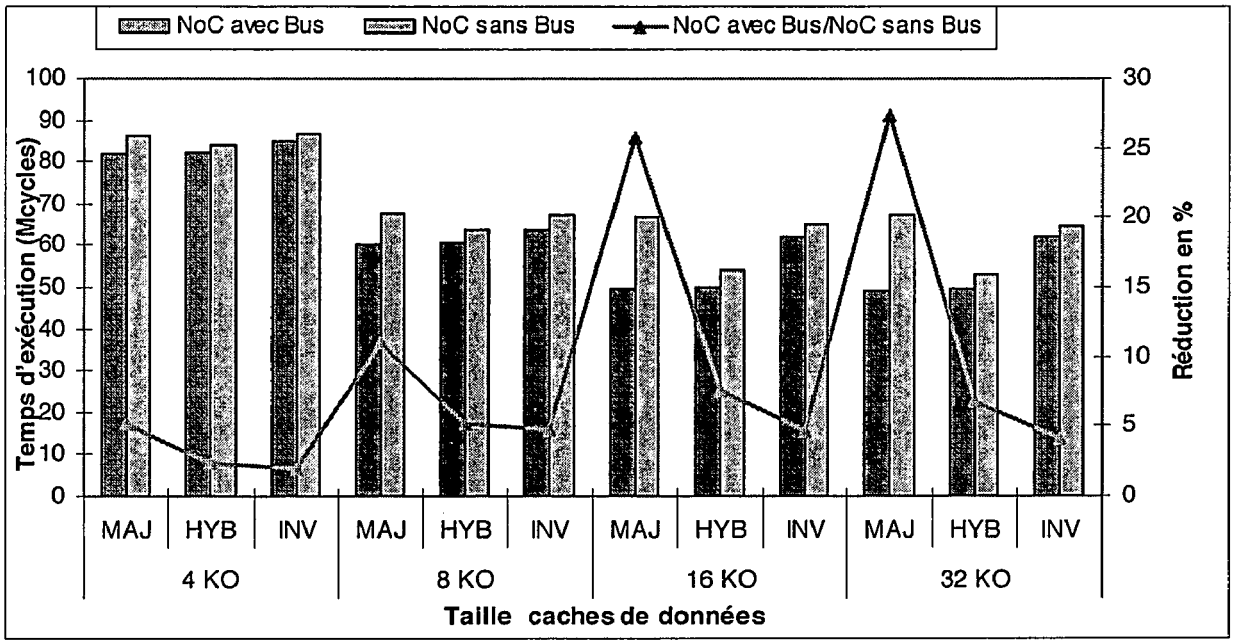


Figure 42. *Comparaison du temps d'exécution du système avec et sans le bus de cohérence en fonction de la taille de cache de données avec la MM pour un MPSoC de 4 processeurs*

3.2. *Résultats expérimentaux pour l'application FFT*

3.2.1. *Réduction du nombre d'échecs de cache*

La figure 43 fournit pour l'application FFT le nombre d'échecs de cache total avec les trois protocoles de cohérence des données dans les caches. Avec le protocole hybride, ce nombre est réduit par rapport au protocole par invalidation d'un facteur de réduction de 36% pour une taille de caches de données de 32 KO. Nous rappelons que dans cette application,

les données sont partagées entre les processeurs uniquement lors de la première phase du traitement. Il s'agit de la phase d'initialisation de la FFT. Par conséquent, dans la phase du traitement de la FFT, les processeurs réalisent le même calcul mais sur des parties différentes de la matrice. Les données ne sont ainsi partagées entre les processeurs que pendant une courte partie du temps de l'exécution totale.

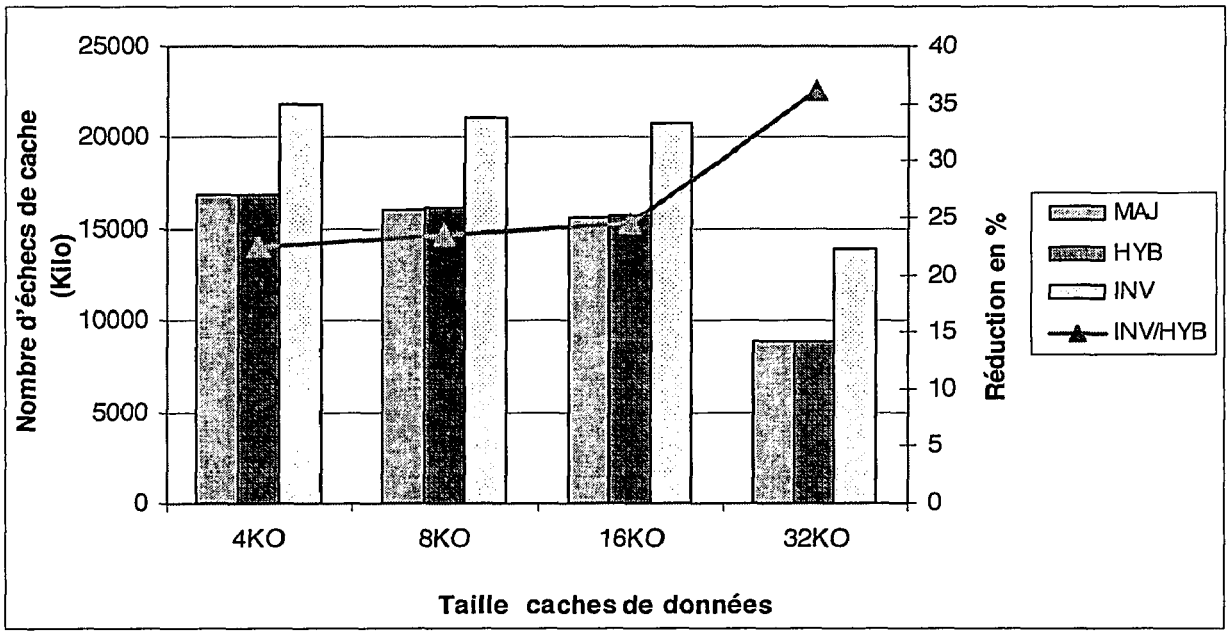


Figure 43. *Nombre d'échecs dans les caches (*10^3) en fonction de la taille de cache de données avec l'application FFT pour un MPSoC de 4 processeurs*

3.2.2. *Évaluation du temps d'exécution dans les 3 protocoles*

La réduction du nombre d'échecs dans les caches par le protocole hybride d'un facteur de 36% avec une taille de caches de 32 KO par rapport au protocole par invalidation, entraîne la réduction du temps d'exécution d'un facteur de 10% (figure 44).

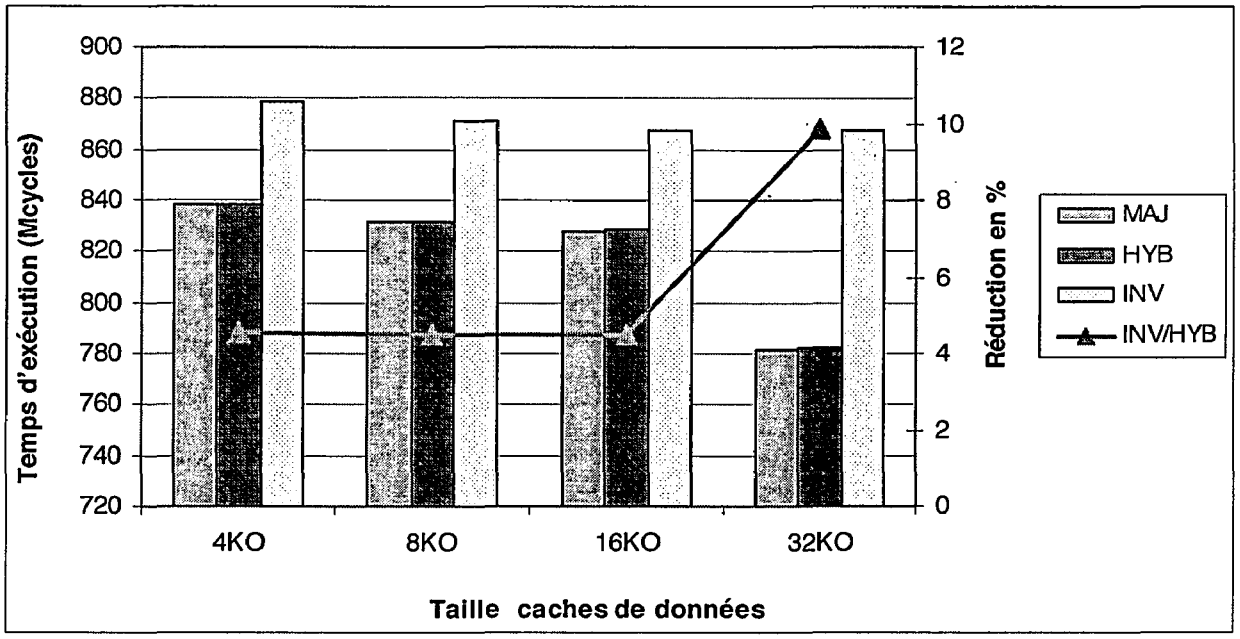


Figure 44. *Temps d'exécution en fonction de la taille de cache de données avec l'application FFT pour un MPSoC de 4 processeurs*

3.2.3. *Évaluation de la consommation d'énergie dans les 3 protocoles*

En se basant sur les résultats illustrés par la figure 45, nous constatons que le protocole proposé réduit la consommation d'énergie pour tous les composants de l'architecture MPSoC. En particulier, la consommation d'énergie du bus de cohérence est réduite en comparaison avec le protocole par mise à jour. La réduction atteint 70% pour des caches de données de taille 32 KO.

Par rapport au protocole par invalidation, le protocole proposé réduit la consommation d'énergie des caches et du NoC. Le facteur d'amélioration concernant le NoC atteint 13.5% pour des caches de données de taille 4 KO et 18% pour des caches de taille 32 KO. La consommation d'énergie des différents bancs de mémoire est également réduite avec le protocole hybride. Le facteur de réduction varie entre 7.5% pour une taille de cache de données de 4 KO et 13% pour 32 KO. Finalement, l'énergie consommée par les processeurs est réduite d'un facteur de 4% pour des caches de données de taille 4 KO et 8% pour des caches de 32 KO.

En se basant sur ces résultats, avec cette version de parallélisation de la FFT, la réduction de la consommation d'énergie totale du système avec notre protocole en comparaison avec le protocole par invalidation atteint 8% pour des caches de données de 32 KO. Par rapport au protocole par mise à jour, le gain en consommation d'énergie est

seulement de 1%. Comme nous l'avons déjà expliqué, cette faible valeur vient du fait que les mises à jour inutiles réalisées par ce protocole passent à travers le bus de cohérence au lieu du NoC, ce qui réduit la consommation d'énergie du système. Ceci montre l'avantage de l'utilisation du bus de cohérence avec le protocole de cohérence des données dans les caches.

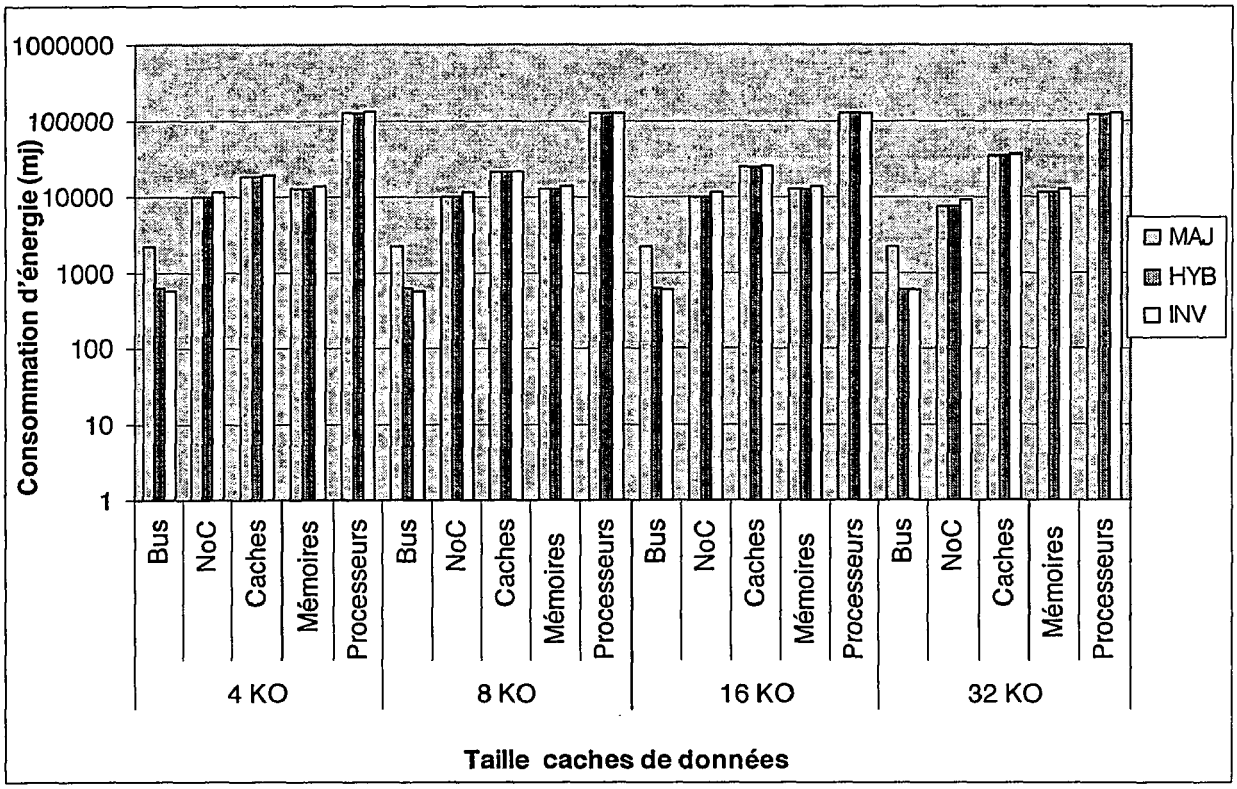


Figure 45. *Consommation d'énergie (en mJoules) des différents composants du système en fonction de la taille des caches de données avec l'application FFT pour un MPSoC de 4 processeurs*

3.3. *Résultats expérimentaux pour l'application JPEG*

3.3.1. *Réduction du nombre d'échecs de cache*

Comme le montre la figure 46, avec l'application JPEG le protocole hybride réduit le nombre d'échecs dans les caches par rapport au protocole par invalidation de 47% pour une taille de cache de données de 32 KO. En effet, cette application contient deux classes de données partagées : *Mostly-read shared data* et *Frequently read-written shared data*. Cette réduction s'explique alors par le fait que le protocole hybride élimine les invalidations des données de type *Frequently read-written shared data*.

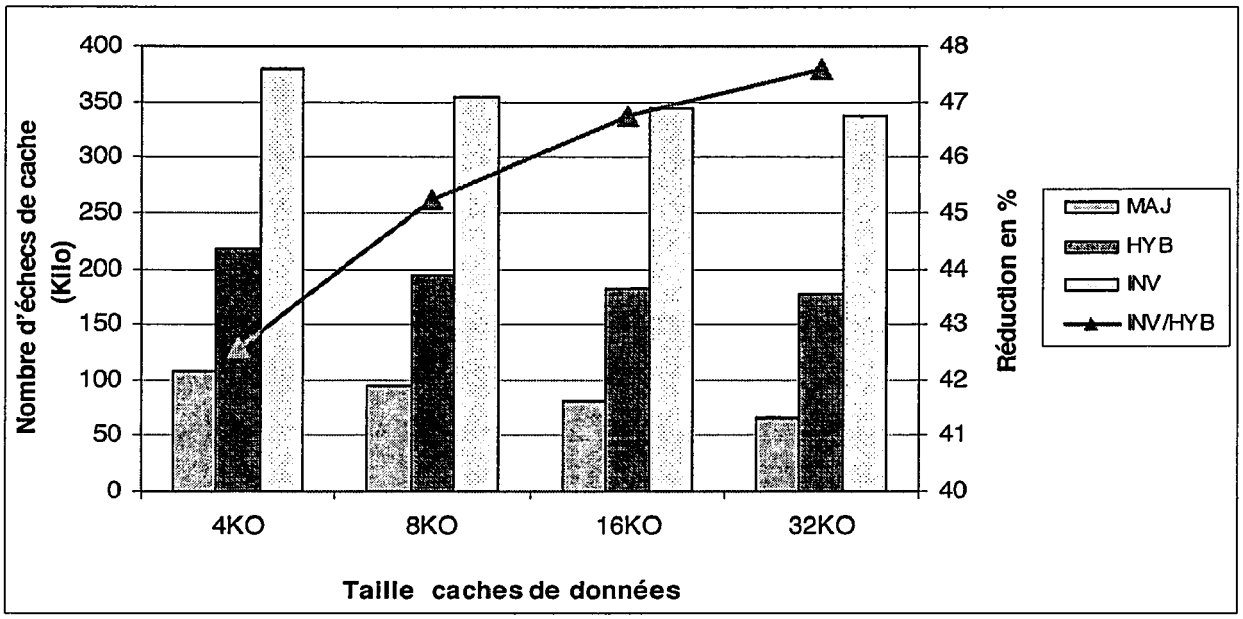


Figure 46. *Nombre d'échecs de cache (*10^3) en fonction de la taille de cache de données avec l'application JPEG pour un MPSoC de 4 processeurs*

3.3.2. *Évaluation du temps d'exécution dans les 3 protocoles*

La figure 47 montre que l'utilisation du protocole hybride permet de réduire le temps d'exécution de l'application JPEG d'une valeur de 2.5%. Cette valeur n'est pas très importante du fait que le trafic de données dans le NoC est plus important par rapport au nombre d'échecs de caches.

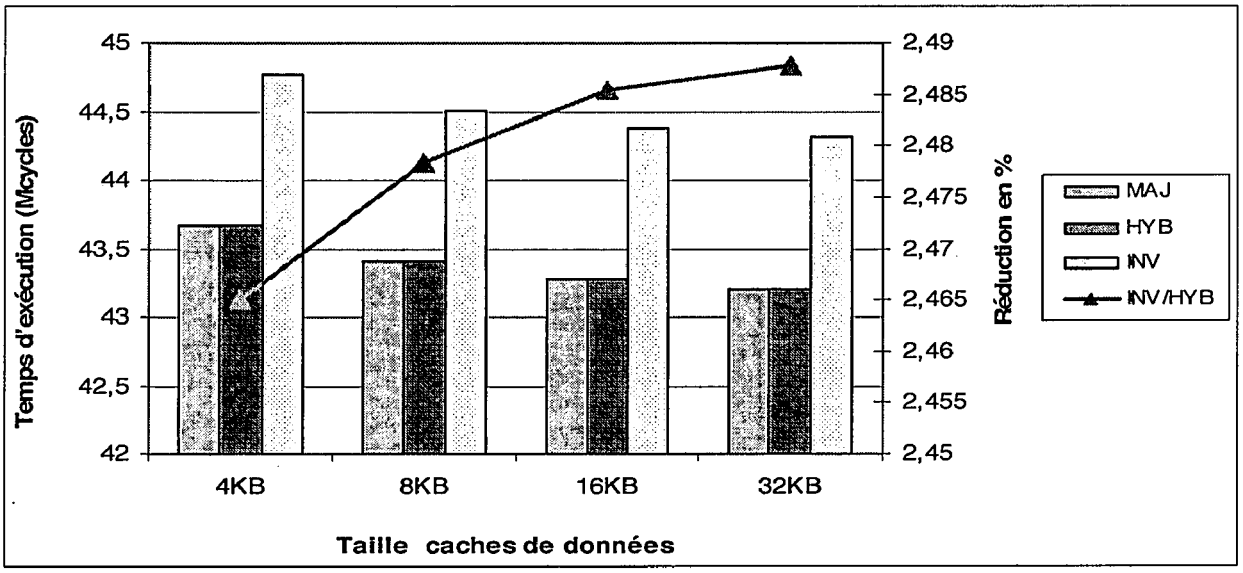


Figure 47. *Temps d'exécution en fonction de la taille de cache de données avec l'application JPEG pour un MPSoC de 4 processeurs*

3.3.3. *Évaluation de la consommation d'énergie dans les 3 protocoles*

Les résultats illustrés dans la figure 48 montrent que le protocole hybride est le protocole qui consomme le moins d'énergie. En effet, la consommation d'énergie du bus est réduite par rapport au protocole par mise à jour. Cette réduction peut atteindre un facteur de 80% pour des caches de données de 32 KO. Par rapport au protocole par invalidation, la réduction est au niveau du NoC et elle peut aller jusqu'à 41% pour des caches de 32 KO. Nous remarquons aussi que le protocole hybride réduit la consommation d'énergie au niveau des bancs mémoire par rapport au protocole par invalidation. Cette réduction atteint la valeur 11.5% pour des caches de données de 32 KO.

En conséquence pour l'application JPEG, une réduction de la consommation d'énergie globale du système de 13% est obtenue avec notre protocole par rapport au protocole par invalidation. Tandis que le gain par rapport au protocole par mise à jour est seulement 2% pour des caches de données de 32KO. Cette différence s'explique par le fait que les opérations de mise à jour inutiles se font à travers le bus de cohérence.

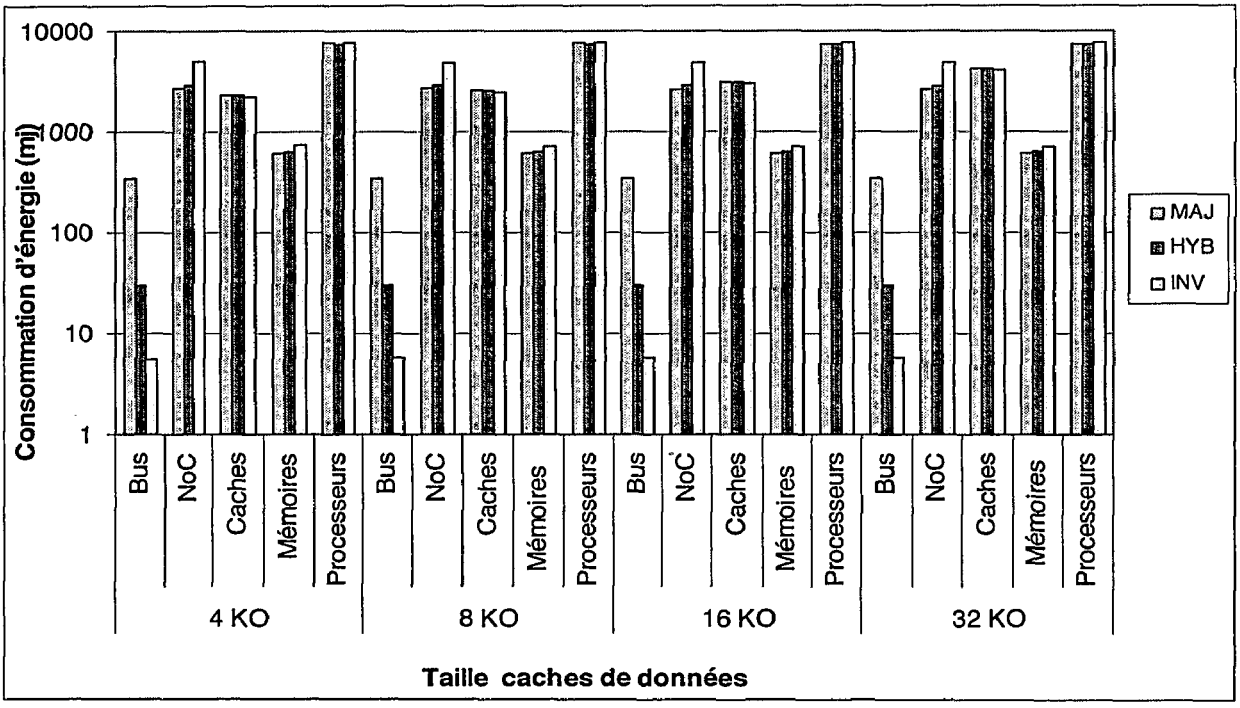


Figure 48. *Consommation d'énergie(en mJoules) des différents composants du système en fonction de la taille de cache de données avec l'application JPEG pour un MPSoC de 4 processeurs*

3.4. *Résultats expérimentaux pour l'application LU*

3.4.1. *Réduction du nombre d'échecs de cache*

L'application LU a été parallélisée de façon à avoir seulement des données partagées de type *Mostly-read shared data*. Or, comme nous l'avons expliqué dans le chapitre précédent avec ce type de données partagées, le protocole invalidation fournit les meilleurs résultats. Ceci est démontré par la figure 49 qui représente une comparaison du nombre d'échecs de cache pour les trois protocoles (hybride, invalidation et mise à jour). Cette figure montre que ce nombre est presque le même pour les trois protocoles. En effet, le protocole par invalidation ne provoque plus d'échecs de cache puisque l'application ne contient pas des données partagées de type *Frequently read-written shared data*. De même, le protocole hybride ne réalise pas d'opérations d'invalidation inutiles. Ceci démontre que le protocole proposé est capable de s'adapter au comportement de l'application.

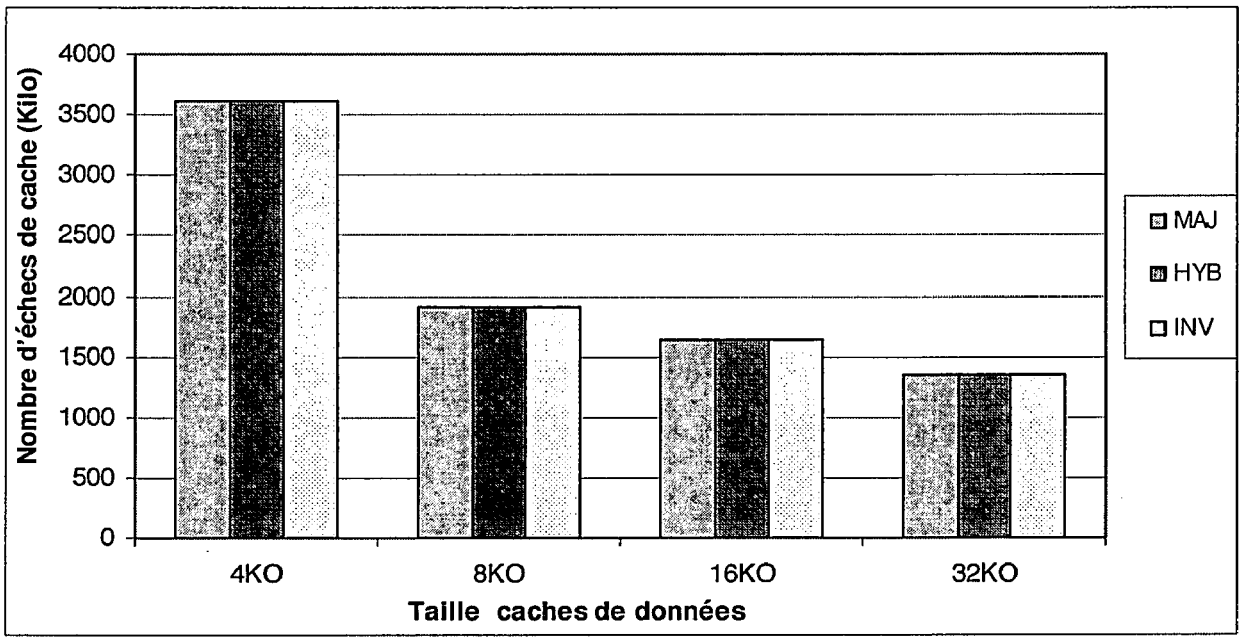


Figure 49. *Nombre d'échecs de cache (*10^3) en fonction de la taille de cache de données avec l'application LU pour un MPSoC de 4 processeurs*

3.4.2. *Évaluation du temps d'exécution dans les 3 protocoles*

Comme le montre la figure 50, le temps d'exécution de l'application LU est presque le même avec les trois protocoles (invalidation, mise à jour et hybride). Ceci est due au nombre de défauts dans les caches qui est identique dans les trois protocoles.

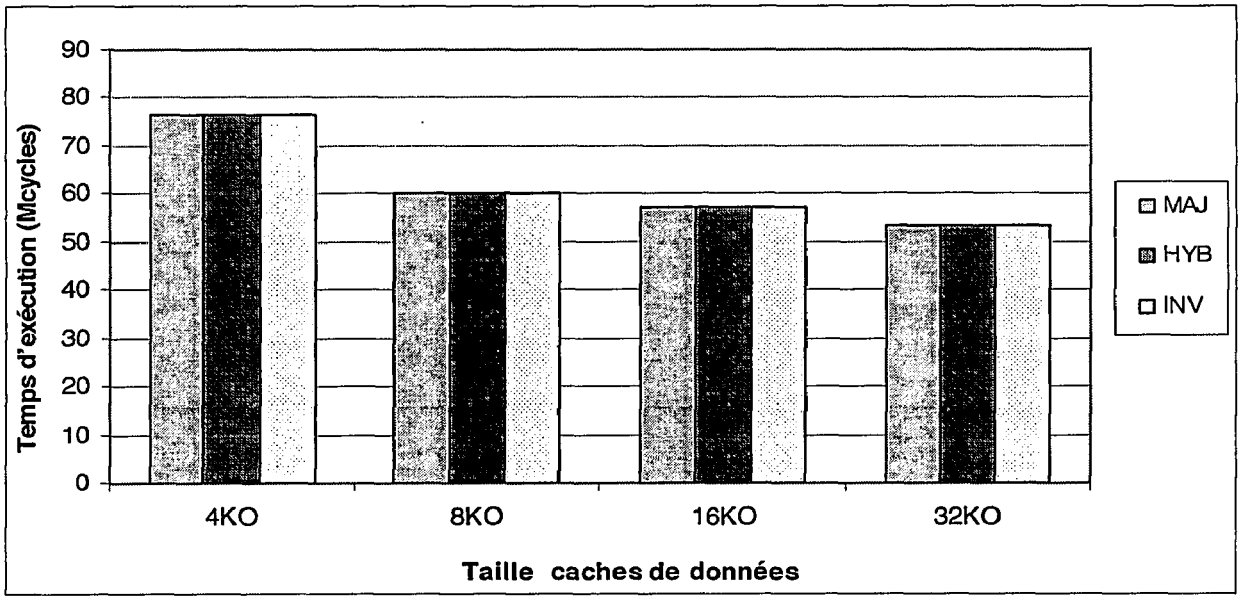


Figure 50. *Temps d'exécution en fonction de la taille de cache de données avec l'application LU pour un MPSoC de 4 processeurs*

3.4.3. *Évaluation de la consommation d'énergie dans les 3 protocoles*

La consommation d'énergie pour chaque composant du système lors de l'exécution de l'application LU, est donnée par la figure 51. Cette figure indique que pour certains composants le protocole hybride réduit légèrement la consommation d'énergie. À l'opposé, au niveau du bus, la consommation d'énergie est réduite jusqu'à 90% en comparaison avec le protocole par mise à jour. Ceci est dû au fait que ce dernier réalise un nombre important des mises à jour inutiles des données de type *Mostly-readshared data.*

En conclusion de cette section sur la comparaison des 3 protocoles, nous avons montré que le protocole proposé peut réduire, pour les différentes applications testées, le nombre des échecs par rapport au protocole par invalidation et le nombre des mises à jour inutiles par rapport au protocole par mise à jour. Il permet de réduire le temps d'exécution et la consommation d'énergie lorsque l'application peut profiter d'un changement dans le protocole de gestion de cohérence. En outre, dans des cas particuliers comme pour l'application LU et MM le protocole hybride proposé adapte son fonctionnement au cours du temps d'exécution.

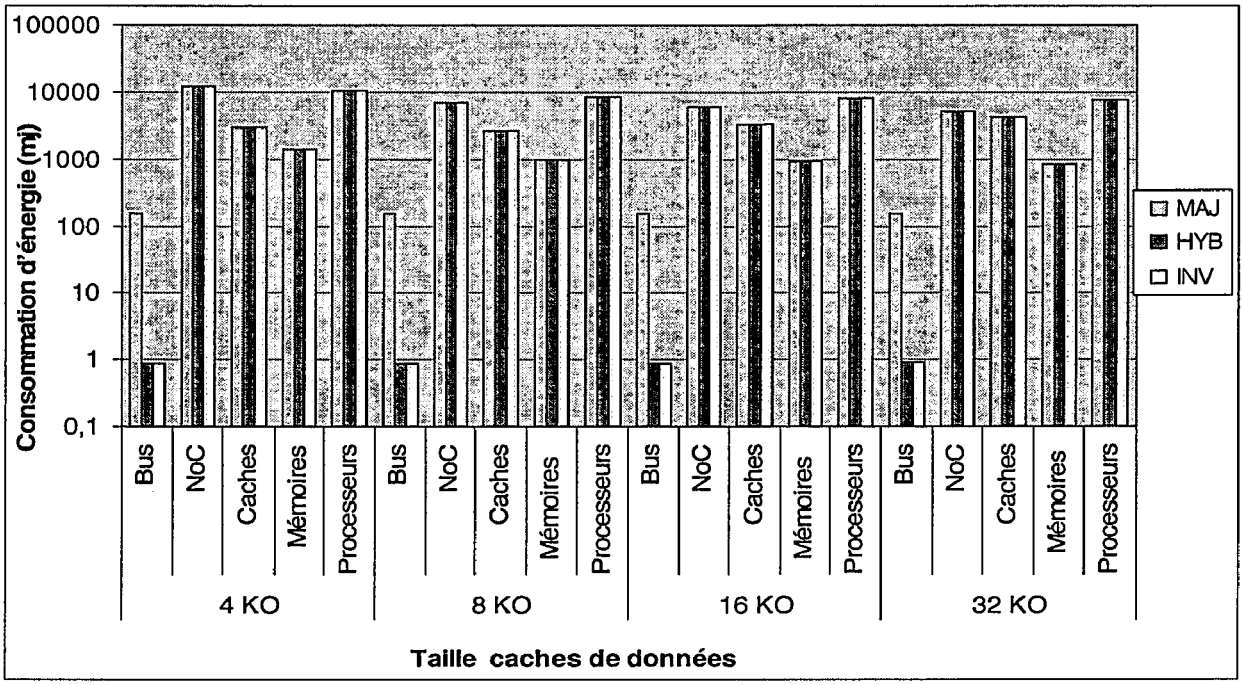


Figure 51. *Consommation d'énergie (en mJoules) des différents composants du système en fonction de la taille de cache de données avec l'application LU pour un MPSoC de 4 processeurs*

4. Extensibilité du protocole hybride pour un nombre de processeurs plus important

Afin de déterminer l'extensibilité du protocole proposé en termes de nombre de processeurs intégrés dans la plateforme, nous avons réalisé une comparaison en termes de consommation d'énergie du système global, entre les trois protocoles : hybride, invalidation et mise à jour. Il s'agit des simulations réalisées avec l'application FFT et pour une architecture MPSoC de 4, 8, 12 puis 16 processeurs et en variant la taille des caches de données. Les résultats obtenus sont exposés dans la figure 52.

La figure 52 montre qu'en augmentant le nombre de processeurs, la consommation d'énergie augmente légèrement pour les trois protocoles. Ceci revient du fait que plus le nombre de processeurs utilisés est important, plus grande sera la taille totale des mémoires caches.

Nous constatons également d'après cette figure qu'en augmentant le nombre de processeurs le protocole hybride fournit toujours la consommation d'énergie la moins élevée. De ce fait, nous pouvons dire que notre protocole de cohérence permet de supporter un plus grand nombre de processeurs avec un certain niveau d'efficacité. L'amélioration est

relativement plus importante en comparaison avec le protocole par invalidation qu'avec le protocole par mise à jour.

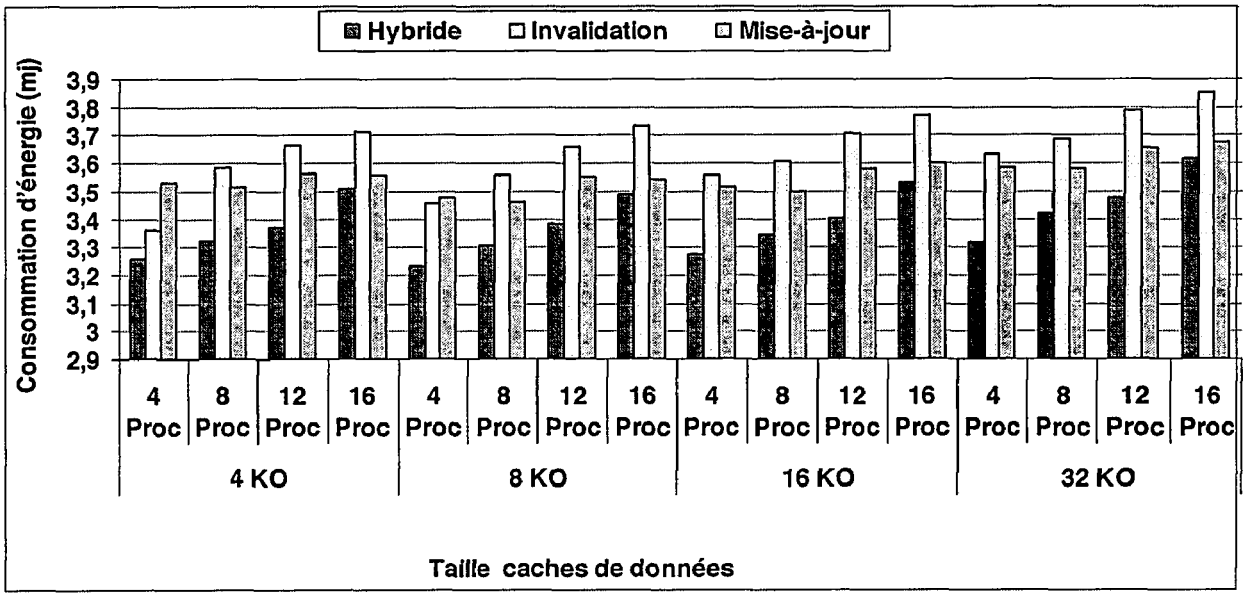


Figure 52. *Consommation de l'énergie en fonction de la taille de cache de données et du nombre de processeurs pour le protocole hybride, invalidation et mise à jour avec l'application FFT*

5. Les surcoûts du protocole proposé

Afin de déterminer les coûts supplémentaires du protocole hybride proposé en termes de surface sur la puce, nous avons utilisé la version 4.1 de l'outil CACTI (CACTI). Ce dernier est un outil qui permet l'estimation des performances, le temps d'accès, la consommation d'énergie, et de la surface des mémoires. Les résultats obtenus indiquent que 25% de la surface de la mémoire partagée est utilisée pour implémenter le répertoire du protocole proposé. Nous rappelons ici, que dans notre approche, deux compteurs de taille 32 bits chacun (UC et W) sont associés à chaque bloc de la mémoire dans le répertoire centralisé. Dans le but de réduire ce coût, nous proposons une solution qui consiste à utiliser une seule paire de compteur (UC, W) pour plusieurs blocs voisins dans un banc mémoire partagée. Cette solution est intéressante du fait qu'elle permet de profiter du principe de la localité spatiale des mémoires caches. Ce principe se base sur le fait que les modèles d'accès aux blocs mémoire d'une même zone de mémoire peuvent être semblables. Ainsi, les blocs mémoires voisins auront tendance à avoir le même comportement et le protocole de gestion de la cohérence optimal sera probablement le même pour ces blocs. En d'autres termes, au

lieu d'associer deux compteurs (UC et W) et un bit « P » pour chaque bloc mémoire, il s'agit ici d'associer deux compteurs (UC et W) et un bit « P » pour « B » blocs de mémoire consécutifs. Par conséquent, ces « B » blocs consécutifs utilisent en cours de l'exécution de l'application le même protocole de cohérence de cache. Pour simplifier l'implémentation nous avons fixé $B = 2^x$ avec x variant entre 1 et 5.

La figure 53 représente les résultats expérimentaux de la consommation d'énergie pour différentes valeurs de B, avec l'application MM et avec une architecture MPSoC de 4 processeurs. Cette figure montre qu'en plus du gain au niveau de la surface du silicium, il y a une amélioration de la consommation d'énergie si nous associons une seule paire de compteurs (UC, W) et un seul bit « P » pour 4 blocs de mémoire consécutifs. Notre estimation montre qu'en mettant B à 4, la surcharge du protocole en termes de surface est seulement 6% de la surface totale de la mémoire partagée.

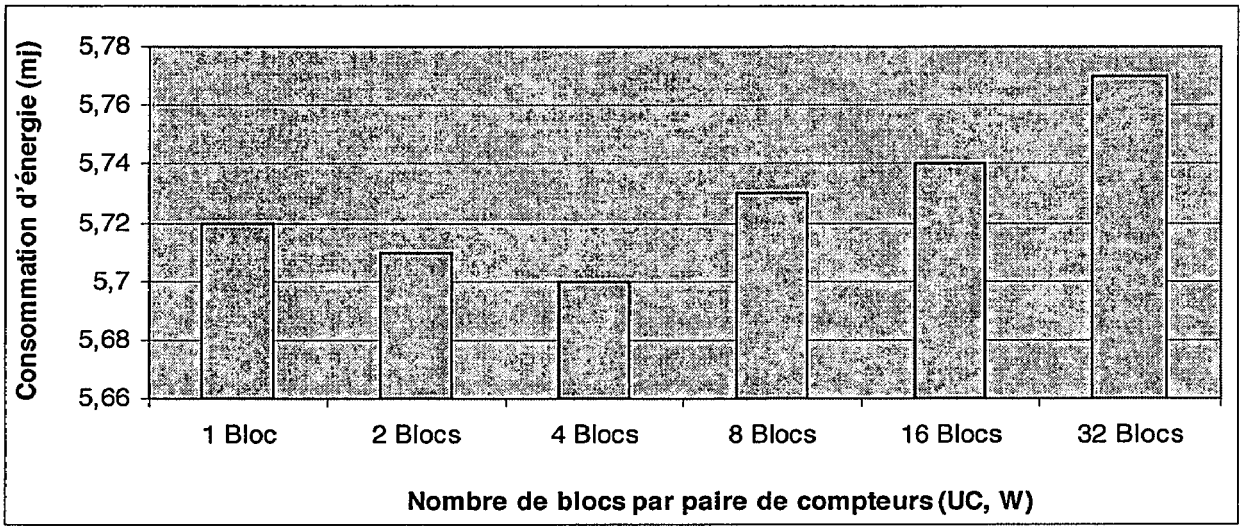


Figure 53. *Consommation de l'énergie en fonction du nombre de blocs associés à chaque pair de compteurs (UC, W) avec l'application de MM pour un MPSoC de 4 processeurs*

Nous remarquons d'après la figure 53 que la consommation d'énergie augmente à partir de B = 8. Cela signifie que les modèles d'accès aux 8 blocs de mémoire consécutifs se diffèrent. Par la suite, l'utilisation du même protocole de gestion de la cohérence des données dans les caches pour ces blocs entraîne la dégradation de la consommation d'énergie du système. Cette situation se produit aussi dans les cas où B prend la valeur 16 ou la valeur 32.

6. Conclusion

Dans ce chapitre, nous avons présenté une évaluation des performances du protocole hybride et de l'architecture que nous avons proposée dans les chapitres précédents. Nous avons présenté en premier lieu l'environnement de simulation que nous avons utilisé. En second lieu, nous avons montré à l'aide des résultats expérimentaux que le protocole hybride (invalidation/ mise à jour) proposé dans cette thèse, en coopération avec l'architecture proposée pour l'implémenter, offre un niveau de performance et de consommation de puissance intéressant. En effet, notre protocole permet de réduire le nombre des défauts dans les mémoires caches par rapport au protocole par invalidation et le nombre des mises à jour inutiles par rapport au protocole par mise à jour. La réduction de la consommation d'énergie obtenue grâce à ce protocole peut atteindre un taux de 34% avec l'application MM, 13% avec le JPEG et 8% avec la FFT.

En outre, nous avons montré que le protocole hybride est capable de s'adapter automatiquement aux patterns d'accès aux données partagées entre les processeurs, au cours de l'exécution de l'application. En effet, bien que l'application LU favorise l'utilisation du protocole par invalidation, le protocole proposé ne dégrade pas les performances mais il adapte son fonctionnement avec le comportement de cette application.

Chapitre 7

Conclusion et perspectives

1. Bilan

Les travaux présentés dans cette thèse visent à fournir une solution matérielle efficace pour le problème de la cohérence des données dans les caches des systèmes sur puce multiprocesseurs. Les applications visées sont les applications de traitement de signal intensif. En particulier, nous nous sommes intéressés aux systèmes sur puce multiprocesseurs utilisant des réseaux sur puce complexes. Cela a été fait tout d'abord par l'intermédiaire d'une étude bibliographique des solutions de la cohérence de caches les plus répondues dans la littérature. Sur la base de cette étude, nous avons regroupé ces solutions en deux familles : le protocole par invalidation et le protocole par mise à jour. Nous avons ensuite montré à l'aide des résultats expérimentaux que ces deux protocoles, ne prennent pas en compte les patterns des accès mémoires réalisés par les applications. Par la suite, nous avons montré la nécessité de disposer d'un nouveau protocole hybride. Ce dernier utilise de façon intelligente les 2 protocoles par invalidation et par mise à jour. Notre protocole tire profit des avantages des deux premiers et s'adapte automatiquement et dynamiquement à la façon avec la quelle les données des applications sont manipulées par les différents processeurs. Ainsi, lorsqu'une donnée est toujours produite par un processeur puis utilisé par un autre processeur, le protocole réalisé sera par invalidation. A l'opposé, lorsqu'une donnée est mise à jour et lue par un grand nombre de processeurs simultanément, c'est le protocole par mise à jour qui sera utilisé.

Le protocole hybride que nous avons proposé permet de remédier aux limites des protocoles statiques qui existent dans la littérature. Le nouveau protocole de cohérence de caches hybride et adaptatif est dédié aux architectures MPSoCs utilisant des NoCs complexes.

Nous avons détaillé le principe de fonctionnement de ce protocole à l'aide des machines d'états finis (ou FSMs). Nous avons aussi proposé une architecture matérielle qui facilite l'implémentation du protocole proposé et optimise ainsi ses performances en termes de temps d'exécution et de consommation d'énergie.

Dans cette thèse, nous avons mis l'accent sur les méthodes de parallélisation des applications de traitement de signal intensif. En effet, nous avons parallélisé plusieurs applications embarquées de façon à avoir différents modèles d'accès aux données partagées ce qui permet une meilleure évaluation du protocole hybride proposé pour la cohérence des caches. En effet, la présence de différents modèles d'accès aux données partagées dans une

application est l'un des paramètres qui affectent les performances des protocoles de cohérence de caches.

Enfin, dans la dernière partie de notre thèse, nous avons évalué les performances du protocole proposé avec les applications que nous avons parallélisées et en utilisant la plateforme de simulation SoCLib. Cette évaluation a été réalisée par l'intermédiaire d'une comparaison des performances du protocole proposé avec les deux protocoles : invalidation et mise à jour. Sur la base des résultats expérimentaux nous avons montré que notre protocole élimine les faiblesses liées à l'utilisation d'un seul protocole fixe (invalidation ou mise à jour) tout le long de l'exécution de l'application.

2. Perspectives

Le travail effectué dans cette thèse peut être poursuivi suivant de nombreuses directions. Nous en présentons ici quelques-unes :

- **Évaluation des Performances**

Avec la version de SoCLib que nous avons utilisée lors du démarrage de la thèse, les tâches de l'application sont placées sur les différents processeurs manuellement. De même les données et les instructions sont placées sur les différentes mémoires disponibles. Pour garantir un ordonnancement correct des tâches entre les processeurs, des variables de synchronisation sont utilisées. Cette phase d'association, nous offre une souplesse dans la parallélisation des applications et par la suite dans l'évaluation du protocole proposé. Par contre, cette version ne permet pas de tester des applications standard comme: SPLASH-2 « *Stanford Parallel Applications for Shared Memory* » (SPLASH), ParMiBench, etc. De ce fait, parmi nos perspectives, c'est l'évaluation du protocole proposé pour des applications multithreadées et non pas uniquement des applications différentes et concurrentes.

- **Réduction de la consommation d'énergie**

Comme nous l'avons indiqué dans le chapitre 6, l'estimation de la surface occupée par le protocole proposé est selon nos estimations proche de 25% de la surface globale de la mémoire partagée. Afin de réduire cette surface nous avons proposé une optimisation qui est basée sur le principe de localité spatiale des références mémoire. Comme possible extension de nos travaux nous proposons se focaliser sur une meilleure exploitation de ce principe. Dans la nouvelle approche, il sera possible de déterminer dynamiquement le nombre de

blocs (noté « B » dans le chapitre 6) qui partagent la même paire de compteurs (UC, W) et le bit « P ». L'idée consiste à garder l'historique des accès aux blocs de mémoire, lors de l'exécution de l'application. Lorsque le nombre des blocs mémoire voisins possédant le même comportement augmente alors la valeur de B augmente. En conséquence, la consommation d'énergie peut être diminuée.

- **Évaluation du protocole proposé pour les MPSoCs hétérogènes**

Les MPSoCs hétérogènes sont des systèmes multiprocesseurs sur puce qui contiennent des processeurs d'architectures différentes. L'avantage majeur de ces systèmes est qu'ils fournissent des performances élevées. En effet, les MPSoCs hétérogènes sont difficiles à programmer, du fait que les processeurs ont des modèles de programmation différents. En particulier, le maintien de la cohérence des données dans les caches est complexe à assurer avec ces architectures. Généralement, afin de réduire cette complexité les concepteurs font recours aux solutions logicielles pour la cohérence de caches. Or, ces solutions ne permettent pas d'obtenir des performances élevés. Dans ce contexte, nous visons à implémenter et à évaluer le protocole proposé pour ces nouvelles architectures.

Bibliographies

A

Adve S., Gharachorloo K., « Shared memory consistency models: A tutorial », *IEEE Computer*, 29(12):66–76, Décembre 1996.

Anoop G., Wolf-Dietrich W., « Cache Invalidation Patterns in Shared-Memory Multiprocessors », *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

Agarwal A., Simoni R., Hennessy J., Horowitz M., « An Evaluation of Directory Schemes for Cache Coherence », *Int'l Symp. Computer Architecture*, pp. 280-289, 1988.

Alexander G., « Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors », Thèse soutenue en 2003.

Anant A., Ricardo B., David C., Kirk L. J., David K., John K., Beng-Hong L., Kenneth M., Donald Y., « The MIT Alewife Machine: Architecture and Performance », *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2-13, Santa Margherita Ligure, Italy, 1995.

Alan E., « Charlesworth. The Sun Fireplane System Interconnect », *IEEE Micro*, 22(1):36-45, Janvier 2002.

Abdullah K., Tarek E. G., « An adaptive cache coherence protocol for chip multiprocessors », *the Second International Forum on Next-Generation Multicore/Manycore Technologies*, New York, NY, USA, 2010.

Archibald J., Baer J. L., « Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model », *ACM Transactions on Computer Systems*, Novembre 1986.

Archibald J. K., « A Cache Coherence Approach for Large Multiprocessor Systems », *2nd International Conference on Supercomputing*, 1988.

Adams M. D., « The JPEG-2000 still image compression standard », *Technical Report*

N2412, ISO/IEC JTC 1/SC 29/WG 1, Septembre 2001.

Anderson C., Karlin A. R., « Two Adaptive Hybrid Cache Coherency Protocols », 2nd *IEEE Symposium on High-Performance Computer Architecture*, 1996.

ARM. http ://www.arm.com.

B

Bunker A., Gopalakrishnan G., « Formal specification of the Virtual Component Interface Standard in the Unified Modeling Language », 2001.

Ben Atitallah R., Niar S., Greiner A., Meftali S., Dekeyser J., « Estimating energy consumption for an MPSoC architectural exploration », *Architecture of Computing Systems*, Mars 2006.

Baslett F., Jermoluk T., Solomon D., « The 4D-MP Graphics Superworkstataion: Computing+Graphics= 40MIPS+40MFLOPS and 100,000 Lighted Polygons per Second », *33rd IEEE Computer Society Int'l Conference – COMPCON`88*, pp 468-471, Février 1988.

Bilir E. E., Dickson R. M., Hu Y., Plakal M., Sorin D. J., Hill M. D., Wood D. A., « Multicast Snooping: A New Coherence Method Using a Multicast Address Network ». *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

Bilir E. E., Dickson R. M., Hu Y., Plakal M., Sorin D. J., Hill M. D., Wood D. A., « Multicast Snooping: A New Coherence Method Using a Multicast Address Network », *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 294–304, May 1999.

C

Choi K., Soma R., Pedram M., « Fine-grained dynamic voltage and frequency scaling for

precise energy and performance trade-off based on the ratio of off chip access to on-chip computation times », *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, IEEE Computer Society, Washington, DC, USA, 10004.

Chtioui H., Ben Atitallah R., Niar S., Abid M., Dekeyser J. L., « Gestion de la cohérence des caches dans les architectures MPSoC utilisant des NoC complexes », *Symposium en Architecture de machines (SympA '2008)*, Février 2008.

Censier L.M., Paul F., « A New Solution to Coherence Problems in Multicache Systems », *IEEE Transactions on Computers*, c-27(12):1112-1118, Décembre 1978.

Chaiken D., Kubiatowicz J., Agarwal A., « LimitLESS Directories: A Scalable Cache Coherence Scheme », *ASPLOS-IV Proc.*, pp. 224-234, Avril 1991.

Chaiken D., Agarwal A., « Software-Extended Coherent Shared MeIEEE Std 1596-1992:

Chun-M. C., Jihong K., Dohyung K., « Reducing Snoop-Energy in Shared Bus-Based MPSoCs by Filtering Useless Broadcasts », *GLSVLSI'07*, Italy 2007.

Cox A. L., Flower R. J., « Adaptive cache coherency for detecting migratory shared data », *Proc. 20th Int'l Symp. on Computer Architecture*. San Diego, California, 98-108, May 1993.

CACTI. http://research.compaq.com/wrl/people/jouppi/cacti.

D

David C., Singh J. P., Anoop G., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.

Dragon McCreight E., « The Dragon Computer System: An Early Overview », *Tech. report, Xerox Corp*, Septembre 1984.

Daniel L., James L., Truman J., David N., Luis S., Anoop G., John L. H., « The DASH Prototype: Implementation and Performance », *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92-103, Gold Coast,

Queensland, Australia, May 1992.

Diana K., Mark O., Justin H., and Frederic T. C., « Brief Contributions Cache Coherence in Intelligent Memory Systems », IEEE Transactions on computers, vol. 52, no. 7, Juillet 2003.

Dagum L., Menon R., « OpenMP : An industry-standard API for shared-memory programming », *IEEE Computational Science & Engineering*, vol. 5, n1, pp. 46–55, janvier/mars 1998.

Dubois M., Briggs F., « Effects of Cache Coherency in Multiprocessors », *IEEE Transactions on Computers*, vol. 31, n° 11, p. 1083-1099, Novembre 1982.

Dahlgren F., « Performance evaluation and cost analysis of cache protocol extensions for shared-memory multiprocessors », IEEE Transactions on computers. vol. 47, no. 10, Octobre 1998.

Dahlgren F., Dubois M., Stenstrm P., « Sequential hardware prefetching in shared-memory multiprocessors », *IEEE Trans. Parallel and Distributed Systems*, 733-746, vol. 6, no. 7, Juillet 1995.

E

Evgeny B., Zvika G., Israel C., Ran G., Avinoam K., « The Power of Priority: NoC based Distributed Cache Coherency », *Networks-on-Chip (NOCS)*, May 2007.

F

Flynn M. J., « Very high-speed computing systems », *Proc. of the IEEE*, 54(12):1901–1909, Décembre 1966.

Frank O., « A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs », Thèse soutenue en April 2009.

Forum (Message Passing Interface). MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*,

vol. 8, n3/4, 1994, pp. 159–416.

Farnaz M. T., David. J. L., « The potential of compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics », *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470. May 1995.

G

Gustavo G., Bruno C. O., Rodrigo S., Ivan S. S., « Cache Coherency Communication Cost in a NoC-based MPSoC Platform », *SBCCI'07*, Rio de Janeiro, Brazil, 2007.

Grindley R., Abdelrahman T., Brown S., Caranci S., DeVries D., Gamsa B., Grbic A., Gusat M., Ho R., Krieger O., Lemieux G., Loveless K., Manjikian N., McHardy P., Srbljic S., Stumm M., Vranesic Z., Zilic Z., « The NUMAchine Multiprocessor », *Proceedings of the 2000 International Conference on Parallel Processing*, pages 487-496, Toronto, Ontario, 2000.

Grahn H., Stenstrom P., Dubois M., « Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models », *Future Generation Computer Systems*, Juin 1995.

Grahn H., Stenstrom P., « Evaluation of a competitive update cache coherence protocol with migratory data detection », *Journal of Parallel and Distributed Computing*, 1996.

Goodman J. R., « Using Cache Memory to Reduce Processor-Memory Traffic », *10th Annual International Symposium on Computer Architecture*, Juin 1983.

H

HP. Hewlett Packard Company. HP Superdome White Paper, May 2002. http://www.hp.com/products1/servers/scalableservers/superdome/infolibrary/whitepapers/technical wp.pdf.

I

ITRS. International Technology Roadmap for Semiconductors, 2009 update system drivers. http://www.itrs.net/links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf.

Intel. Corp., The IA32 Intel® Architecture Software Developer's Manual http://developer.intel.com/design/ pentium4/manuals/245472.htm

Intel, 2010. Teraflops Research Chip. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.2010.

IEEE. Standard for Scalable Coherent Interface », New *York: IEEE*, 1993.

IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface. New York: IEEE, 1993.

J

John L. H., David A. P., *Computer architecture: A Quantitative Approach*, 4th Edition, Morgan Kaufmann Publishers Inc., 2006. 15, 62.

Jeffrey K., David O., Mark H., John H., Richard S., Kourosh G., John C., David N., Joel B., Mark H., Anoop G., Mendel R., and John L. H., « The Stanford FLASH Multiprocessor », *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302-313, Chicago, Illinois, Juin 1994.

Jhalani D., Palsetia D., « Adaptive cache coherence protocol using migratory shared data », 2007.

K

Kourosh G., Daniel L., James L., Phillip G., Anoop G., John H., « Memory consistency and event ordering in scalable shared-memory multiprocessors », *Proceedings of the 17th*

Annual International Symposium on Computer Architecture, pages 15-26, 1990.

Kourosh G., Madhu S., Simon S., Stephen V. D., « Architecture and Design of AlphaServer GS320 », *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13-24, Cambridge, Massachusetts, Novembre 2000.

Karlin A .R., Manasse M S., Rudolph L., Sleator D., « Competitive Snoopy Caching », *27th Annual Symposium on Foundations of Computer Science*, 1986.

L

Laudon J., Lenoski D., « The SGI Origin: a ccNUMA highly scalable server », *The 24th Annual International Symposium on Computer Architecture*, Conference Proceedings pages 241-251, 1997.

Lamport L., « How to make a multiprocessor computer that correctly executes multiprocess programs », *IEEE Trans. On Computers*, C-28:690–691, Septembre 1979.

Lenoski D., Laudon J., Gharachorloo K., Weber W.-D., Gupta A., Hennessy J., Horowitz M., Lam M.S., « The stanford dash multiprocessor Computer», *IEEE Computer*, 25(3):63-79, Mars 1992.

Liqun C., John B. C., « An adaptive cache coherence protocol optimized for producer-consumer sharing », *13th Int'l Symp. on High Performance Computer Architecture (HPCA-13)*, pages 328—339, 2007.

M

Motorola. MPC 750A RISC Microprocessor Hardware Specification, http://www.mot.com/SPS/PowerPC/library/ 750_hs.pdf

Martin M. M. K., Harper P. J., Sorin D. J., Hill M. D., Wood D. A., « Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors », *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, Juin 2003.

Milo M. K. M., Daniel J. S., Mark D. H., David A. W., « Bandwidth Adaptive Snooping », *the Eighth International Symposium on High-Performance Computer Architecture* (HPCA.02), 2002.

Matts B., « SM-prof: A Tool to Visualise and Find Cache Coherence Performance Bottlenecks in Multiprocessor Programs », *Conference on Measurement and Modeling of Computer Systems*, pages 178-187, Ottawa, Ontario, May 1995.

Matthew R. G, Jeffrey S. R, Dan E., Todd M. A, Trevor M., Richard B. B., « MiBench: A free, commercially representative embedded benchmark suite », *Workload Characterization, 2001(WWC '01)*, 2001.

N

Nilsson H., Stenstrom P., « The Scalable Tree Protocol A Cache Coherence Approach for Large-Scale Multiprocessors », *Proc. Int'l Symp. Parallel and Distributed Processing*, pp. 498-506, Dec. 1992.

Noel E., Li-Shiuan P., Li S., « In-Network Cache Coherence », *International Symposium on Microarchitecture (MICRO'06)*, 2006.

P

Petrot F., Greiner A., Gomez P., « On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures », *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, DSD 2006* , pages 53-60, 2006.

Papamarcos M., Patel J., « A Low Overhead Coherence Solution for Multiprocessors with

Private Cache Memories », *11th Annual Int'l Symposium on Computer Architecture*, pp 348-354, Juin 1984.

Panades I. M., Greiner A., Sheibanyrad A., « A low cost network-on-chip with guaranteed service well suited to the GALS approach », *NanoNet*, 2009.

Philippe D., « Spécification multidimensionnelle pour le traitement du signal systématique », Thèse soutenue le 15 décembre 2005.

R

Radhika T., Amit P. S., Jaswinder P. S., Susan J., and John L. H., « An Evaluation of a Commercial CC-NUMA Architecture I The CONVEX Exemplar SPP1200 », *Proceedings of the 11th International Parallel Processing Symposium*, pages 8-17, Suisse, Avril 1997.

Ran M., Isask'har W., Israel C., Avinoam K., « Best of Both Worlds: A Bus Enhanced NoC (BENoC) », *3rd ACM/IEEE International Symposium on Networks-on-Chip*, Washington, USA, 2009.

S

Sweazey P., Smith A. J., « A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Future bus », *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, Juin 1986.

Susan O., Anant A., « Evaluating the Performance of Software Cache Coherence », *3rd international Conference on architectural Support for programming languages and operating systems*, 1989.

Sorin D. J., Plakal M., Hill M. D., Condon A. E., Martin M. M. K., Wood D. A., « Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol », *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, Juin 2002.

SOCLIB. An open platform for modelling and simulation of multi-processors system on chip. http://soclib.lip6.fr/Home.html.

Sinisa S., Zvonko G.V., Michael S., Leo B., « Analytical Prediction of Performance for

Cache Coherence Protocols », *IEEE Transactions on Computers*, 46(11):1156-1173, Novembre 1997.

SPLASH. http://www.capsl.udel.edu/splash.

SYSTEMC. http://www.systemc.org.

T

Tang C. K., « Cache System Design in the Tightly Coupled Multiprocessor System », *Proceedings National Computer Conference*, pages 749-753, Octobre, 1976.

Thapar M., Delagi B., Flynn M.J., « Linked List Cache Coherence for Scalable Shared Memory Multiprocessors », *Proc. Int'l Parallel Processing Symp.*, pp. 34-43, Avril 1993.
Thapar M., Delagi B., « Stanford Distributed Directory Protocol », Computer, vol. 23, no. 6, pp. 78-80, Juin 1990.

Tom L., Russell M. C. S., « A CC-NUMA Computer System for the Commercial Marketplace », *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308-317, Philadelphia, Pennsylvania, May 1996.

Tendler J., Dodson S., Fields S., Hung L., Sinharoy B., « Power4 System Microarchitecture», *Technical report, IBM Server Group*, 2001.

W

Wolf-Dietrich W., Anoop G., « Analysis of Cache Invalidation Patterns in Multiprocessors », *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, Boston, Massachusetts, Avril 1989.

Y

Yeimkuan C., Laxmi N. B., « An Efficient Tree Cache Coherence Protocol for Distributed Shared Memory Multiprocessors », IEEE Transactions on computers, , Vol. 48, NO. 3, Mars 1999.

Z

Zandvelt F., « On caches for a (rt) multiprocessor environment », Unpublished, prepared for the VSI/OCB working group, Avril 1998.

Gestion de la cohérence des données dans les systèmes multiprocesseurs sur puce

Résumé: Les travaux présentés dans cette thèse visent à concevoir une architecture performante et efficace pour la gestion de la cohérence des données dans les mémoires caches des systèmes sur puce multiprocesseurs (MPSoC). Dans cette thèse nous nous intéressons tout particulièrement aux architectures à mémoire partagée et aux applications de traitement de signal intensif. Plusieurs solutions ont été proposées dans le passé pour résoudre ce problème. Néanmoins, la majorité de ces solutions existantes ont été pensé pour les systèmes multiprocesseurs haute-performances. Dans ce type de systèmes les contraintes liées aux ressources matérielles et à la consommation d'énergie sont rarement prises en compte. A l'opposé, dans les systèmes embarqués qui nous intéressent ici, ces contraintes jouent un rôle de premier plan. De plus, les solutions existantes ne prennent pas en compte les modèles d'accès aux données partagées réalisés par les processeurs. Nous proposons dans cette thèse un nouveau protocole de gestion de la cohérence de cache basé sur deux protocoles simples, nommés les protocoles par invalidation et par mise à jour. De plus le protocole proposé s'adapte automatiquement aux modèles d'accès aux données. Une architecture matérielle qui facilite son implémentation et qui optimise ses performances est également proposée. Les résultats expérimentaux montrent que le protocole proposé ainsi que l'architecture offrent un niveau de performances et de consommation d'énergie intéressant.

Mots clés: Systèmes sur puce multiprocesseurs (MPSoC), mémoire partagée, gestion de la cohérence des données en cache, performances des architectures, consommation d'énergie.

Managing cache coherency of shared data in shared memory multiprocessor systems-on-chip

Abstract: The work presented in this thesis aims to provide an efficient hardware solution for managing cache coherency of shared data in shared memory multiprocessor systems-on-chip (MPSoC) dedicated for intensive signal processing applications. Several solutions are proposed in the literature to solve this problem. However, most of these solutions are efficient only for high-performance multiprocessor systems. These systems take rarely into account hardware resources and energy consumption limitations. In MPSoCs architectures these constraints are very important. In addition, these solutions do not take into account access patterns from the different processors to shared data. In this thesis, we propose a new approach for treating cache coherency problem. It consists on a new hybrid (invalidation/update) adaptive cache coherence protocol. A hardware architecture that facilitates its implementation and optimizes its performance is also proposed. The experimental results show that the proposed protocol in conjunction with this architecture provides an interesting level of performances and energy consumption.

Key-words: Multiprocessor systems-on-chip (MPSoC), shared-memory, cache coherency, performances, energy consumption.