Hiérarchie mémoire dans les systèmes intégrés multiprocesseurs construits autour de réseaux sur puce

Hela Belhadj Amor

▶ **To cite this version:**

Hela Belhadj Amor. Hiérarchie mémoire dans les systèmes intégrés multiprocesseurs construits autour de réseaux sur puce. Architectures Matérielles [cs.AR]. Université Grenoble Alpes, 2017. Français. NNT : 2017GREAM049 . tel-01762643

HAL Id: tel-01762643
https://theses.hal.science/tel-01762643

Submitted on 10 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel du : 7 août 2006

Présentée par

Hela BELHADJ AMOR

Thèse dirigée par **Frédéric PÉTROT**
et codirigée par **Hamed SHEIBANYRAD**

préparée au sein du laboratoire **TIMA**
et de l'École Doctorale **Mathématiques, Sciences et Technologies de l'Information, Informatiques (MSTII)**

Hiérarchie mémoire dans les systèmes intégrés multiprocesseurs construits autour de réseaux sur puce

Thèse soutenue publiquement le **5 octobre 2017**
devant le jury composé de:

M. Smail NIAR
Professeur, Université de Valenciennes, Président
M. Daniel ETIEMBLE
Professeur émérite, Université Paris Sud, Rapporteur
M. Jean-Philippe DIGUET
Directeur de recherche CNRS, Laboratoire LAB-STICC, Rapporteur
M. Quentin MEUNIER
Maître de Conférences, Université Pierre et Marie Curie, Examinateur
M. Frédéric PÉTROT
Professeur, Grenoble-INP, Directeur de thèse
M. Hamed SHEIBANYRAD
Chargé de recherche CNRS, LIP6, Co-Directeur de thèse

Abstract

Multi/many-cores parallel systems for high-power computing at low energy costs are nowadays a reality. However, exploiting the performance of these architectures depends on the efficiency of the system in managing data accesses. The aim of our work is to improve the efficiency of these accesses by exploiting the hardware architecture characteristics.

In a first part, we propose a new cache hierarchy organization that aims at maximizing the use of the available storage space at each level. This solution, based on non-uniform cache access architectures (NUCA), supports inter and intra-level transfers of the hierarchy. It requires a cache coherency protocol that suits its specifications.

Obviously, the transfer of data in the hierarchy is also a determinant of the system performance. In a second part, we consider the specific communication needs of the protocol. We suggest the use of a virtualized network as an *ad-hoc* communication medium to manage consistency traffic at a lower cost. It links the caches of the same level to support intra-level transfers, which are a specificity of our protocol, in order to reduce the average access latency.

Keywords : Multi-Processor System-On-Chip (MPSoC), Memory hierarchy, Consistency, Network-on-Chip (NoC), Non Uniform Cache Access (NUCA).

Résumé

Les systèmes parallèles de type multi/pluri-cœurs permettant d'obtenir une grande puissance de calcul à bas coût énergétique sont de nos jours une réalité. Néanmoins, l'exploitation des performances de ces architectures dépend de l'efficacité du système à gérer les accès aux données. Le but de nos travaux est d'améliorer l'efficacité de ces accès en exploitant les caractéristiques de l'architecture matérielle.

Dans une première partie, nous proposons une nouvelle organisation de la hiérarchie des mémoires caches qui maximise l'utilisation de l'espace de mémorisation disponible à chaque niveau. Cette solution, basée sur les architectures à accès non uniforme au cache (NUCA), supporte les transferts inter et intra-niveau de la hiérarchie. Elle requiert un protocole de cohérence de caches qui s'adapte à ses spécifications.

Certes, le transfert des données au niveau de la hiérarchie est aussi un déterminant de la performance du système. Dans une seconde partie, nous prenons en compte les besoins de communication spécifiques du protocole. Nous proposons un réseau virtualisé comme support de communication *ad-hoc* afin de gérer le trafic de cohérence à moindre coût. Ce dernier relie les caches d'un même niveau pour supporter les transferts intra-niveaux, qui sont une spécificité de notre protocole, en vue de réduire la latence moyenne d'accès.

Mots Clés : Système Multiprocesseur sur Puce (MPSoC), Hiérarchie mémoire, Cohérence, Réseau sur Puce (NoC), Non Uniform Cache Access (NUCA).

Table des figures

1.1 Tendance en nombre de processeurs/SoC dans le futur 2

2.1 Hiérarchie de la mémoire 6
2.2 Différentes architectures (a) UMA, (b) NUMA, (c) NUCA 8
2.3 Performance des multi-diffusions dans un réseau sur puce 12

3.1 Structure de cache coopératif à contrôle (a) : centralisé, (b) : distribué . . 22
3.2 Exemple de topologies semi-régulière (irrégulière) qui ne supporte pas un routage *DOR* 32
3.3 2x4x3 core de topologie grille avec un routage ZYX, et 2x1x2 flanc X-, et 2x2x3 flanc X+. La flèche décrit le chemin du paquet du nœud situé dans le flanc X- jusqu'au nœud dans le flanc X+ en traversant le core 32

4.1 Structure de la hiérarchie des mémoires locales 36
4.2 Configuration logique des clusters 37
4.3 Aperçu du mécanisme de recherche de donnée haut niveau : (a) donnée dans un banc L2 différent de la cible (b) donnée dans un banc L3 différent de la cible 39
4.4 Scénarios de l'écriture d'une donnée résidente dans un banc de cache L2 différent de la cible (a) État de la ligne MT (b) État de la ligne M (c) État de la ligne SS 45
4.5 Scénarios de la lecture d'une donnée et/ou d'une instruction résidente dans un banc de cache L2 différent de la cible (a) Lecture d'une donnée ou d'une instruction dans l'état MT (b) Lecture d'une donnée dans l'état M (c) Lecture d'une donnée dans l'état SS (d) Lecture d'une instruction dans l'état M ou SS 46
4.6 Mise à jour de la liste des caches au niveau d'un banc L2 différent de la cible lors de plusieurs lectures d'un même bloc (a) d'instructions (b) de données 48
4.7 Scénario d'écriture dans une ligne partagée (Upgrade) 49
4.8 Évincement d'une ligne résidente dans le cache L1 50

4.9 Évincement d'une ligne du banc L2 dans différentes états (a) partagé, (b) modifié et aucun cache L1 ne possède la ligne, (c) modifié et la copie à jour est dans un autre cache L1 . . . 51
4.10 Phénomène d'interblocage . . . 52
4.11 Scénario de redirection . . . 53
4.12 Architecture logique . . . 54

5.1 Exemple de réseau virtuel . . . 58
5.2 Exemple de réseau logique ayant comme topologie un tore inclus dans un réseau physique de topologie maille . . . 59
5.3 (a) Exemple de méta-routage logique au niveau d'un réseau logique de topologie tore inclus dans un réseau physique de topologie maille (b) Ajout et suppression de l'en-tête temporaire au niveau des routeurs du réseau logiques . . . 60
5.4 L'algorithme de routage au niveau (a) du port d'entrée local et (b) des ports d'entrée (E,W,N,S) . . . 61
5.5 Machine d'états du port local du routeur . . . 62
5.6 Architecture du port local du routeur . . . 63
5.7 Machine d'états pour les ports d'entrées 2D . . . 63
5.8 (a) Circuit de suppression de l'en-tête temporaire (b) Architecture du port d'entrée 2D . . . 64
5.9 (a) Violation dans le réseau physique (b) Restriction dans le modèle tour pour l'algorithme X-first dans une topologie maillé NoC . . . 65
5.10 (a) Un VN de topologie anneau contourne la région défectueuse pour diriger la paquet vers un chemin sain (b) Algorithme de routage tolérant aux fautes . . . 67

6.1 Architecture d'un routeur . . . 71
6.2 Format du paquet de multidiffusion . . . 73
6.3 Architecture de l'interface réseau . . . 75
6.4 Architecture des liens . . . 76
6.5 Exemple de recherche de données . . . 77
6.6 Les paquets générés pour chaque direction . . . 78
6.7 Les bases du routage implémentant la diffusion . . . 79

7.1 Support de communication autour du système mémoire . . . 82
7.2 Flot de simulation . . . 83
7.3 Un réseau physique 4x4 sur lequel est sont projetés deux réseaux logiques de taille 2x2 et 3x3 . . . 85

7.4 Latence moyenne des paquets en utilisant des trafics synthétiques pour un réseau physique de taille 4x4 86
7.5 Un réseau physique 8x8 sur lequel sont projetés deux réseaux logiques de taille 4x4 et 5x5 86
7.6 Latence moyenne des paquets en utilisant les trafics synthétiques pour un réseau physique de taille 8x8 87
7.7 Plateforme utilisée pour les simulations 89
7.8 Configuration de l'architecture à simuler 91
7.9 Taux de défauts du cache L2 pour deux configurations différentes : une avec 4 bancs et l'autre avec 16 bancs 92
7.10 Pourcentage des blocs évincés de l'ensemble des bancs du cache L2 pour différentes applications 92
7.11 Pourcentage de blocs évincés par banc de cache L2 93
7.12 Pourcentage de blocs évincés par bancs de cache L2 pour l'application *Radix* 93
7.13 Latence moyenne du réseau 95
7.14 Distribution des accès aux bancs de cache L2 95
7.15 Accélération pour les différentes applications 96
7.16 Consommation énergétique au niveau du réseau d'interconnexion normalisé à S_NUCA 98
7.17 Taux de défauts de cache L2 100
7.18 Latence moyenne du réseau 100
7.19 Accélération pour les différentes applications 101

Liste des tableaux

7.1 Configuration du simulateur 85
7.2 Surface et puissance consommées par différents types de routeurs 88
7.3 Paramètres des architectures modélisées 89
7.4 Paramètres des applications 90

Table des matières

Abstract i

Résumé iii

Remerciements v

Table des figures vii

Liste des tableaux xi

1 Introduction 1

1.1 Problème général et objectifs de la thèse 2
1.2 Contributions 3
1.3 Plan de la thèse 3

2 Problématique 5

2.1 Performance des systèmes multiprocesseurs intégrés 5
2.1.1 Organisation de la mémoire 6
2.1.2 Accès aux données 7
2.1.3 Structure de communication 7
2.2 Hiérarchie des mémoires caches 7
2.2.1 Mémoire partagée hiérarchique 7
2.2.2 Partagé-distribué : NUCA 8
2.2.2.1 Localité des données 9
2.2.2.2 Gestion des données 10
2.2.3 Cohérence des données 10
2.3 Diffusion : base du protocole de cohérence 11
2.3.1 Support matériel 11
2.3.2 Algorithme de routage 12
2.4 Conclusion 13

3 État de L'Art 15
3.1 Hiérarchie des mémoires caches 16
3.1.1 Généralités 16
3.1.1.1 Forme de la hiérarchie 16
3.1.1.2 Politiques de gestion de la hiérarchie 17
3.1.1.3 Organisation : caches privés ou partagés 17
3.2 Architecture NUCA 19
3.2.1 Réplication 19
3.2.2 Migration 21
3.2.3 Cache Coopératif 21
3.2.4 Discussion : limitations de la gestion du contenu des caches 23
3.3 Autour des NoCs 24
3.3.1 Co-conception mémoire et NoC 25
3.3.2 Cohérence de caches et Réseau 26
3.3.3 Discussion 26
3.4 Virtualisation 27
3.4.1 Généralité 27
3.4.2 Domaines de virtualisation 27
3.4.3 Virtualisation du réseau sur puce 28
3.4.3.1 Utilisation des ressources 28
3.4.3.2 Rendement 29
3.4.3.3 Consommation d'énergie 29
3.4.3.4 Domaine de cohérence 29
3.4.4 Stratégies de virtualisation 30
3.4.4.1 Canaux virtuels 30
3.4.4.2 Partitionnement des ressources 30
3.5 Positionnement 33
3.6 Conclusion 34

4 Protocole de cohérence pour une nouvelle organisation hiérarchique des mémoires caches 35
4.1 Structure de la hiérarchie des mémoires caches 36
4.2 Métriques d'évaluation des performances 37
4.3 Gestion de mémoires caches 38
4.3.1 Placement des données 38
4.3.2 Mécanisme de recherche de la donnée 39
4.3.3 Problème lié à la structure de la hiérarchie 40
4.4 Protocole de cohérence des caches 40
4.4.1 Principe 40

4.4.2 Implantation du protocole 41
4.4.2.1 États du premier niveau du protocole 41
4.4.2.2 États de second niveau du protocole 42
4.4.2.3 Requête L2_Fwd_GET(X/S/_INST) 45
4.4.2.4 Requête L2_Fwd_Upgrade 49
4.4.2.5 Mécanisme d'évincement dans le cache L1 50
4.4.2.6 Mécanisme d'évincement dans le cache L2 50
4.4.3 Interblocage 51
4.5 Virtualisation de la hiérarchie 53
4.6 Conclusion 54

5 Un réseau virtualisé comme support de communication 57
5.1 Concept du réseau virtuel 58
5.2 Routage Logique 58
5.2.1 Exemple de méta-routage 60
5.2.2 Routage au niveau du port d'entrée local 61
5.2.3 Routage au niveau des ports d'entrées (E,W,N,S) 61
5.3 Micro architecture des ports d'un routeur 62
5.4 Interblocage 64
5.4.1 Situation 64
5.4.2 Recouvrement d'interblocage 65
5.5 Exemples d'utilisation du concept de réseau logique 66
5.5.1 Applications qui requièrent différentes topologies 66
5.5.2 Sécurité vis-à-vis des applications malveillantes 66
5.5.3 Routage tolérant aux fautes 66
5.5.4 Protocole de cohérence de cache 67
5.6 Conclusion 68

6 Stratégie de multidiffusion utilisant le concept du réseau virtualisé 69
6.1 Modèle de diffusion 69
6.1.1 Calcul des différents voisins 69
6.1.2 Abstraction de l'algorithme 71
6.2 Stratégie de multidiffusion 71
6.2.1 Micro architecture du routeur 71
6.2.1.1 Structure du paquet de multidiffusion 72
6.2.1.2 Canaux virtuels et politique d'arbitrage 73
6.2.2 Contrôleur de l'interface réseau 74
6.2.3 Contrôle de flux au niveau des liens 76
6.3 Exemple de recherche de données 77

6.4 Conclusion 79

7 Expérimentations & Résultats 81
7.1 Plateforme d'expérimentation : Gem5 82
7.2 Expérimentations I : Réseau virtualisé 83
7.2.1 Garnet en mode autonome 83
7.2.2 Trafics synthétiques utilisés 84
7.2.3 Expérimentation I.1 84
7.2.4 Résultats I.1 : Performance du réseau 85
7.2.5 Résultats I.2 : Passage à l'échelle 86
7.2.6 Surface et puissance consommées 87
7.3 Expérimentation II : Protocole de cohérence exploitant le concept de VN 88
7.3.1 Architecture simulée 88
7.3.2 Applications utilisées 90
7.3.3 Résultat II.1 : Taux de défauts de cache 90
7.3.4 Résultat II.2 : Latence moyenne du réseau 94
7.3.5 Résultat II.3 : Temps d'exécution 96
7.3.6 Consommation énergétique 97
7.3.7 Surcoût matériel 99
7.3.8 Passage à l'échelle 99
7.3.8.1 Taux de défauts de cache 99
7.3.8.2 Latence moyenne du réseau 100
7.3.8.3 Temps d'exécution 101
7.4 Conclusion 101

8 Conclusions et Perspectives 103
8.1 Conclusions 103
8.2 Perspectives 105

Glossary 107

Publications 109

Annexe : Protocole de cohérence 111

Références 119

Chapitre 1: Introduction

La révolution technologique de ces dernières décennies doit en grande partie son effervescence à l'informatique. En fait, cette science est devenu indispensable au quotidien de chacun : téléphone intelligent, console de jeux, voiture autonome, etc, ... Son évolution rapide envahit le marché grand public mais aussi des domaines plus critiques tels que la prévision météorologique, l'accélération des calculs scientifiques (analyse financière, simulation physique, ...) ou encore le contrôle des systèmes complexes (industrie avionique et sous marine). Le cœur des systèmes informatiques est le couple processeur mémoire auquel nous apporterons notre attention.

Le développement qu'a connu le marché des processeurs doit son essor à l'augmentation de la densité d'intégration des transistors due à la diminution de la finesse de gravure. Par conséquent, les ordinateurs sont moins coûteux et plus puissants. Néanmoins, l'augmentation de la fréquence est limitée pour des raisons liées à la consommation énergétique et à la dissipation thermique. Ainsi, dans le but de gérer cette dissipation thermique et de garantir une grande puissance de calcul, des processeurs multi-cœurs regroupant plusieurs unités de calcul avec des fréquences de fonctionnement plus basses ont vu le jour. Ensuite, face à la demande de marché qui cherche à améliorer les performances à moindre coût, une nouvelle génération de systèmes massivement parallèles apparaît : les many-cœurs[TKM+02] (voir figure 1.1). Ces architectures many-cœurs permettent une grande capacité de calcul à basse consommation énergétique grâce à l'intégration de plusieurs dizaines (voir centaines) de cœurs relativement simples sur une même puce.

En dépit des capacités physiques offertes par les architectures précédentes, l'exploitation de la puissance de calcul disponible n'est pas toujours triviale. En effet, profiter de la puissance de calcul ne pourra se faire qu'à la condition que les données soient accessibles efficacement et simplement du point de vue des applications. Ainsi, la définition et l'implantation de systèmes mémoires efficaces, tant du point de vue de la rapidité d'accès aux données que du point de vue de la simplicité d'utilisation par les programmes parallèles s'exécutant sur ces plateformes, est un défi scientifique et technique de première importance.

L'utilisation de la mémoire dans ce genre de système est basée sur une approche hiérarchique pour laquelle chaque niveau de mémoire représente un coût et une capacité différents. C'est une approche largement utilisée dans des systèmes à large échelle. Le rôle de cette hiérarchie est de combler l'écart de vitesse entre le processeur et la mémoire principale, relativement plus lente. Certaines niveaux peuvent être partagés, ce qui les rend sujets à des conflits d'accès et d'état. Afin de réduire les conflits, une solution est de subdiviser les caches en un ensemble de bancs. Différents critères de

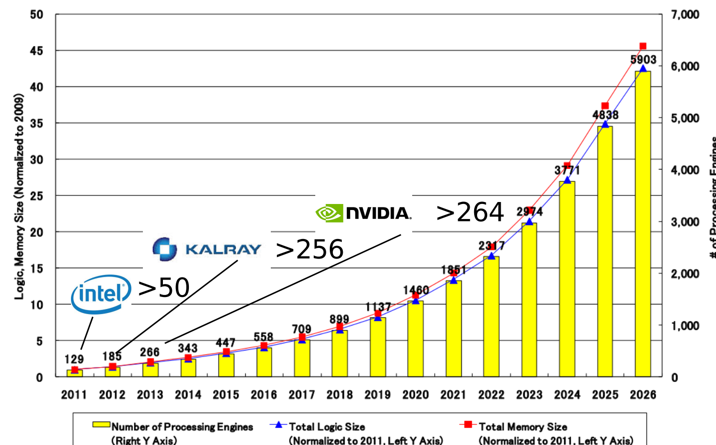


FIGURE 1.1 – Tendance en nombre de processeurs/SoC dans le futur

subdivision existent, et leur but est d'aboutir à une utilisation efficace de l'espace de cache disponible sans induire des latences additionnelles. Toutefois, ces subdivisions peuvent être recouvrantes, ce qui dicte le besoin de veiller au maintien de la cohérence des données qui résident dans les bancs.

Un des aspects les plus problématiques lié aux architectures sur lesquelles nos travaux s'appuient est la communication cohérente de tous les composants et son impact sur les performances du système. L'infrastructure de communication de ces systèmes est un réseau intégré sur la puce (NoC) autour duquel s'articulent les caches. Ainsi, le temps d'accès à ces derniers peut différer de la perspective d'un processeur à un autre et ceci reflète l'aspect des architectures NUCA. Certes, la bande passante n'est plus le point critique de ces systèmes. Ainsi, la latence, la consommation et la complexité sont les déterminants de l'efficacité des transferts. Il convient donc de se concentrer sur ces aspects pour exploiter efficacement les potentialités du NoC au profit de la hiérarchie des mémoires caches. Une des solutions permettant d'aider à traiter les problèmes aussi bien de calcul que de communication est la virtualisation. Cette technique, appliquée au réseau, offre de nombreuses opportunités telles qu'une souplesse architecturale, une capacité à hiérarchiser le trafic pour résoudre les performances souhaitées, et une utilisation moindre des ressources, réduisant de ce fait la consommation d'énergie. En s'appuyant sur ces avantages, nous pensons que cette approche est pertinente si on l'adapte au besoin des protocoles de cohérence associé à de nouvelles organisations de la hiérarchie mémoire.

1.1 Problème général et objectifs de la thèse

Les travaux de cette thèse se penchent sur l'aspect mémoire des architectures multi-processeurs, vu que l'accès aux données constitue généralement un point critique de tout système.

L'obtention de hautes performances dépend de la bonne utilisation de l'espace de mémorisation sur puce (réduction des défauts de cache) et de l'efficacité de gestion de la cohérence des données, étant donné que les accès par plusieurs cœurs mènent

souvent à des problèmes de cohérence et de conflit d'accès. Il ne faut cependant pas négliger l'infrastructure de communication qui doit assurer les transferts de données efficacement.

Notre objectif est d'utiliser au mieux la capacité de mémorisation des niveaux hiérarchiques de la mémoire cache tout en garantissant la cohérence des données qui y résident et en réduisant les latences moyennes d'accès.

1.2 Contributions

Les contributions de cette thèse sont les suivantes :

- Proposition d'une nouvelle structure de la hiérarchie mémoire permettant d'exploiter l'espace de mémorisation disponible sur la puce et offrant une flexibilité au niveau de l'organisation des données manipulées. Le problème de cohérence de ces données met en évidence le besoin d'un nouveau protocole qui réponde aux exigences de cette hiérarchie.
- Introduction du concept de réseau virtualisé, une redéfinition de la notion de réseau virtuel pour s'adapter au mieux aux besoins des applications. Il permet de répondre à la problématique de gestion des communications pour une architecture distribuée.
- Combinaison du concept de réseau virtualisé avec la nouvelle organisation de la hiérarchie et son protocole de cohérence, pour renforcer la localité des accès et donc réduire les latences, tout en se contentant des ressources matérielles disponibles.

1.3 Plan de la thèse

Le manuscrit est organisée comme suit : dans le chapitre 2, nous abordons les raisons qui motivent ce travail et identifions les problèmes à résoudre. Le chapitre 3 est voué à l'état de l'art sur la hiérarchie des mémoires caches autour d'un réseau sur puce. Il évoque les notions de base et les principaux travaux existants qui s'y rapportent. Le chapitre 4 présente notre première contribution liée à la question de l'organisation des données dans les mémoires caches. Géré par les divers cœurs, un protocole de cohérence de caches qui s'adapte aux spécifications de la hiérarchie est défini et développé.

Cette première partie a mis en évidence le besoin d'un support de communication *ad-hoc*. Ainsi, le chapitre 5 détaille un nouveau concept de virtualisation appliqué aux réseaux sur puce, qui, en plus de répondre aux exigences de notre protocole, pourrait également être utile dans d'autres domaines d'applications.

Le chapitre 6 est un cas d'application du concept décrit dans le chapitre précédent. Il présente en détail l'adaptation de notre concept de réseau virtualisé au protocole de cohérence qui est la base de l'organisation de la hiérarchie. Le chapitre 7 illustre les résultats des expériences couvrant aussi bien la partie réseau que la partie protocole.

Enfin, sur la base des contributions apportées, le chapitre 8 conclut ce travail en répondant aux questions posées au niveau de la problématique. Il en résume également les limitations, et propose des pistes de recherche en vue de travaux futurs.

Chapitre 2: Problématique

Sommaire

1.1 Problème général et objectifs de la thèse 2
1.2 Contributions 3
1.3 Plan de la thèse 3

Dans ce chapitre, nous présentons tout d'abord le contexte de nos travaux qui portent sur la performance des systèmes multiprocesseurs et les facteurs qui sont liés. Ensuite, nous introduisons les problématiques autour desquelles s'articulent les différentes contributions de ce travail, qui sont liées à l'organisation des mémoires caches, à la gestion des données et au support matériel assurant la circulation et l'échange des données entre les différentes mémoires. Nous terminons par une conclusion qui regroupe l'ensemble des questions auxquelles nous apportons des réponses dans cette thèse.

2.1 Performance des systèmes multiprocesseurs intégrés

La recherche de performance dans les systèmes électroniques rend les systèmes intégrés sur puce de plus en plus complexes. Les unités de traitement (IP) sont maintenant intégrées pour satisfaire les besoins des applications. Ceci entraîne *de facto* l'augmentation des communications pour le contrôle des traitements et pour les échanges des données.

- En s'appuyant sur l'architecture des systèmes actuels, ces données sont stockées dans des mémoires. Hors les concepteurs sont confrontés à une barrière technologique baptisée mur mémoire [WM] qui fait référence à une différence d'évolution de la rapidité des processeurs face à celle des mémoires. Pour limiter son impact, les systèmes utilisent des hiérarchies de caches.
- En outre, à l'heure actuelle, le facteur décisif permettant d'atteindre de hautes performances réside dans la rapidité d'accès aux données par les unités de traitement des processeurs. Hennessy et Patterson ont évalué de 20 à 40% les instructions faisant référence à la mémoire dans la majorité des programmes [HP07, HGP+03]. De telles proportions révèlent l'importance de ces accès et la nécessité de se focaliser sur les stratégies minimisant le temps d'accès aux données.
- Favoriser une exploitation performante des systèmes multiprocesseurs nécessitent de plus une structure de communication adéquate. Cette dernière est le support de base des schémas d'accès assurant l'échange des données. Ainsi, la qualité de

l'architecture d'interconnexion devient prépondérante pour la performance du système.

Pour résumer, l'un des facteurs freinant l'obtention de hautes performances de calcul est *l'accès aux données*, car les délais associés à ces accès dépendent fortement de *l'organisation mémoire* et de *la structure supportant la communication*.

2.1.1 Organisation de la mémoire

La mémoire est d'une grande importance dans la conception des manycœurs. Elle est représentée par plusieurs types de technologies et d'organisations, fournissant des performances variées. Le problème majeur est lié à sa capacité et l'exploitation de cette dernière par les applications. Le dilemme vient du fait que l'on désire exploiter la capacité de mémoire très importante alors que le besoin en performance dicte l'utilisation de mémoires de capacités moins importantes ayant des temps d'accès plus courts. La solution est de ne pas se limiter à un seul composant ou à une technologie, mais plutôt d'adopter une hiérarchie mémoire comme l'illustre la figure 2.1.

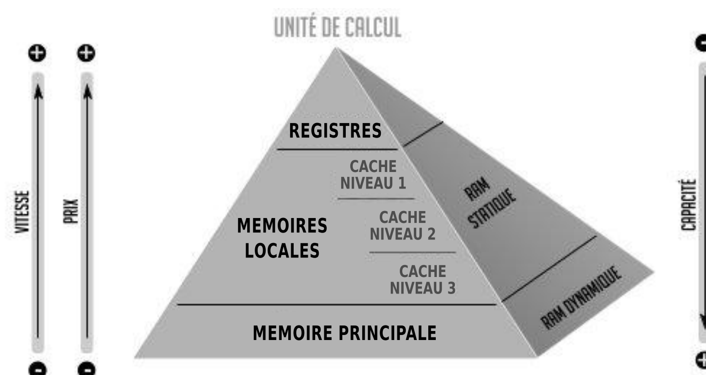


FIGURE 2.1 – Hiérarchie de la mémoire

La mise en place d'une telle structure hiérarchique est une solution qui a été largement explorée. Les niveaux sont de plus en plus rapides au fur et à mesure que la donnée se rapproche de l'unité de calcul.

2.1.2 Accès aux données

Dans le cadre de nos travaux, un accès efficace est déterminé par la latence d'accès. Cette latence dépend, d'une part, de l'éloignement physique du banc mémoire auquel on souhaite accéder (les caches ($1 \sim 10$ cycles), mémoire dynamique DRAM (une centaine de cycles)). D'autre part, le franchissement des interfaces tels que l'interface de bus, pont d'accès entre deux bus et files d'attentes influe sur la durée d'un accès. Une autre particularité concerne la bande passante des liens de communication. Dès que la charge sur le lien dépasse un seuil, la latence d'accès croit [PGJ+05]. Ainsi, la conception des liens de communication doit tenir compte des besoins du type d'application, des performances souhaitées et des contraintes de coût (coût d'implantation et de consommation).

En ce qui concerne la consommation, son évaluation en amont est primordiale afin de concevoir des solutions économiques.

2.1.3 Structure de communication

Les structures de communication représentent un élément important qui déterminent les performances globales du système. En effet, la complexification croissante des applications qui induit une augmentation des communications dans le circuits pour l'échange de données et pour le contrôle des traitements, doit être gérée efficacement. Le but est d'augmenter les débits d'échanges, de réduire la latence du traitement, d'assurer la flexibilité de l'architecture et notamment de s'adapter aux futurs systèmes incluant des centaines d'unités de traitement. Une gestion efficace de la communication sur puce qui offre une large bande passante et limite la consommation d'énergie est primordiale.

Les structures de communications classiques ne sont performantes que pour un nombre d'unités à interconnecter réduit. Avec l'avènement des circuits intégrant un grand nombre de ressources les communications classiques deviennent incompatibles. Ce genre de problème a été étudié et des solutions ont été proposées dans le contexte des réseaux d'interconnexion de machine parallèles [CGS99]. L'idée est d'adapter les solutions des réseaux non intégrés au contexte des systèmes sur puce en tenant compte de la surface, de la latence et de la consommation d'énergie. Ainsi, le concept du réseau sur puce (NoC) apparaît [GG00].

Dans ce contexte, nous nous intéressons aux mémoires caches hiérarchisés (mémoires locales) et avec la difficulté de passage à l'échelle des manycœurs, l'étude de leurs organisations autour d'un réseau sur puce est nécessaire. En effet, la hiérarchie des caches n'est performante que si les accès y sont localisés, c'est à dire si les données sont souvent réutilisées. Or, ceci dépend de l'application induisant des schémas d'accès divers et variés. Donc, nous nous focalisons sur une organisation des caches qui tire profit des potentiels du réseau sur puce.

2.2 Hiérarchie des mémoires caches

2.2.1 Mémoire partagée hiérarchique

La conception des puces multi-cœurs et les efforts pour surmonter les limitations des multiprocesseurs symétriques ont conduit à l'émergence des architectures hiérarchiques, construites autour d'un sous-système mémoire hiérarchique. Une architecture à mémoire partagée hiérarchique est toute plate-forme multiprocesseur disposant d'unités de calcul et de mémoires organisées de manière hiérarchique, les unités de calcul partagent la mémoire globale. Des exemples typiques de machine avec de semblables architectures sont les machines UMA, NUMA, COMA et NUCA.
UMA (*Unified Memory Access*) est le cas où les processeurs partagent une mémoire unique et le temps d'accès à cette mémoire est identique. Le problème de ce type d'architecture est que la mémoire ne supporte qu'un seul accès à la fois. NUMA (*Non Uniform Memory Access*) est l'architecture avec mémoire partagée présentant des coûts d'accès mémoires non uniformes. Les éléments de calcul, dans ce type d'architecture, sont connectés à plusieurs bancs de mémoire physiquement distribués. À l'instar des NUMA, COMA (*Cache Only Memory Access*) présente des latences d'accès très inférieures vu que le banc

mémoire est un banc de cache et que la donnée peut être répliquée ou migrée. Ce type d'architecture est coûteux car il nécessite des mécanismes matériels pour assurer la cohérence. Pour NUCA (*Non Uniform Cache Access*), c'est l'avènement des many-cœurs qui en a dicté le besoin. Dans cette configuration, la mémoire cache est beaucoup plus large vu qu'elle est organisée en un ensemble de bancs qui sont logiquement partagés.

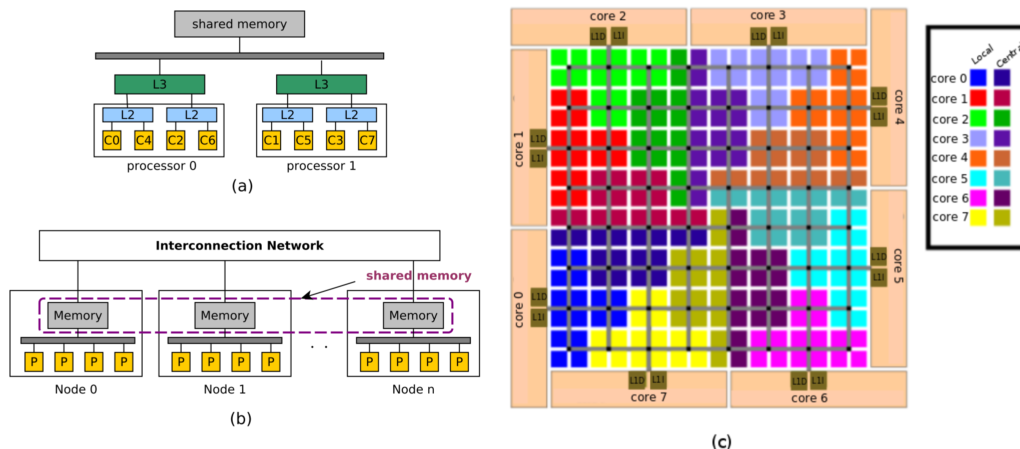


FIGURE 2.2 – Différentes architectures (a) UMA, (b) NUMA, (c) NUCA

Dans cette thèse nous nous sommes intéressés à des architectures qui présentent des coûts d'accès mémoire non uniformes : les plates-formes NUCA. Ces dernières présentent des topologies complexes qui doivent être bien exploitées afin de tirer le maximum de performance applicatif.

2.2.2 Partagé-distribué : NUCA

L'architecture NUCA est une architecture de système multiprocesseur dans laquelle les unités de calculs sont connectées à des bancs mémoire logiquement partagés et physiquement distribués. La notion de logiquement partagé est expliquée par le fait que la mémoire cache est perçue par les unités de calcul et le système d'exploitation comme une mémoire unique partagée. Ces bancs donnent à l'architecture des latences d'accès variées. NUCA, similaire à l'architecture NUMA, présente un accès non uniforme à un certain niveau de la mémoire cache. Ainsi, l'accès aux données est conditionné par la distance entre le cœur et le banc de cache dans lequel réside la donnée. Dans ce type d'architecture, les processeurs utilisés sont dotés de plusieurs niveaux de cache, afin de réduire le temps d'accès. De plus, un réseau d'interconnexion est nécessaire pour relier le nombre élevé de cœurs et de bancs mémoires.

Pour résumer, dans notre contexte, la hiérarchie mémoire associée à chaque cœur est formée de plusieurs niveaux de cache. Maximiser l'utilisation de la mémoire sur puce pour des meilleurs performances permet de réduire le nombre d'accès, coûteux en temps et en consommation, à la mémoire externe. Donc, une gestion efficace de la capacité mémoire sur puce est nécessaire [SSP+07].

2.2.2.1 Localité des données

Les mémoires caches tirent profit de la notion de localité en maintenant des copies récemment utilisées. Les prochains accès à ces données sont effectués directement dans ces caches. La notion de localité est basée sur l'idée que les données ne sont pas accédées de façon uniforme durant toute l'exécution. La localité peut être spatiale ou temporelle.

– **Localité spatiale** : le bloc de données prochainement demandé est probablement situé à proximité des données en cours d'utilisation.
– **Localité temporelle** : un bloc de données récemment accédé sera probablement utilisé dans un futur proche.

Néanmoins, le principe de localité ne répond pas à tous les problèmes liés aux accès mémoires et en pose de nouveaux tels que :

– **Congestion sur un banc de cache** : Un grand nombre de processeurs accédant à des données stockées physiquement dans un même banc mémoire peut créer un point de congestion. Ce dernier aura des conséquences dramatiques sur les performances du système. Cette congestion augmente la latence d'accès comme c'est le cas lors des accès sur un bus. Ce problème est d'autant plus crucial pour des systèmes comprenant des dizaines de nœuds de calcul. Il convient donc de trouver une solution.
– **Pénalité d'échec** : Notre contexte d'étude se focalise sur les architectures multiprocesseurs (NUCA), ainsi la distance entre la donnée et le nœud demandeur peut être très grande. L'usage de niveaux de cache permet de réduire la pénalité d'échec mais avec des limites. En fait, plus la demande est servie par un niveau inférieur plus la pénalité est importante. Tenter d'augmenter le nombre de niveaux de cache est une mauvaise solution car cela implique une croissance de la taille des mémoires caches dont le coût en temps d'accès et en surface deviendrait ruineux. Donc au lieu d'ajouter des niveaux de cache, il serait intéressant d'optimiser l'utilisation des différents niveaux.
– **Surface Utile** : Comme il a été mentionné ci-dessus, les mémoires caches occupent une surface non négligeable de la puce. Une utilisation massive de ces caches réduit la surface utile à cause de la duplication des données. Le rangement des données à exemplaire unique permet de réduire la taille de la puce ou d'avoir des processeurs supplémentaires. L'idéal est de maximiser la surface utile de la puce en gardant les mêmes performances d'accès.

Renforcer la localité des accès aux données dans une architecture many-cœurs est une problématique primordiale à résoudre [CRM07] car elle impacte le temps d'exécution, la scalabilité et la consommation énergétique. Pour cela, la donnée doit être placée à proximité du processeur qui l'utilise.

2.2.2.2 Gestion des données

Afin de gérer efficacement l'espace de mémorisation au niveau de la puce, l'exploration des techniques de localisation est nécessaire. Ces techniques aident à repérer l'emplacement des données pour répondre aux accès. La répartition statique des données sur les caches implique un emplacement unique pour chaque donnée, ce qui facilite sa localisation. À part sa simplicité et sa rapidité, cette répartition peut causer des goulots d'étranglement dans le cas où les données sont affectées au même cache. Une autre approche est d'autoriser les données à changer d'emplacement en fonction de leur utili-

sation. Une telle situation nécessite la connaissance globale des traces de mouvements des données. Ainsi la dynamique de cette approche représente les limitations d'une gestion centralisée tel que la réduction du parallélisme et la surcharge du système.

Récemment, d'autres systèmes ont autorisé la migration et la réplication des données afin d'assurer les accès locaux et la répartition des charges. Cependant, cette technique impacte l'espace de mémorisation et le coût de la migration est important.

– En s'appuyant sur les capacités des réseaux sur puce, quelle organisation de la hiérarchie mémoire doit-on adopter pour avoir des accès efficaces ?
– Comment placer les données dans les bancs mémoires afin de minimiser le trafic sur la puce, en gardant à l'esprit la contrainte de capacité limitée de ces bancs ?

En dépit des capacités que l'architecture NUCA peut offrir, l'exploitation de la capacité de mémorisation sur puce n'est pas toujours triviale. D'un point de vue organisationnelle, une gestion efficace des conflits d'accès à chaque niveau de mémoire est nécessaire. Nous nous intéressons aux problèmes de la cohérence des données au niveau de cette architecture.

2.2.3 Cohérence des données

L'architecture des systèmes à mémoire partagée distribuée ayant plusieurs niveaux de cache nécessite une gestion correcte et efficace du partage des données. En effet, les cœurs en possession d'une copie (donnée dans leur propre cache) peuvent modifier la valeur en local, ce qui implique un problème d'incohérence entre les différentes copies. La gestion de la cohérence dans les systèmes many-cœurs nécessite la considération des contraintes liées à l'organisation de la mémoire dans ce type d'architecture. En fait, les mécanismes de cohérence peuvent être organisés de différentes manières en fonction du nombre de niveaux de cache et de la structure des différents niveaux. Cette cohérence de données peut être gérée de façon logicielle ou à travers des mécanismes matériels. L'avantage de cette dernière stratégie est que le programmeur n'a plus à se soucier de la gestion des caches. Il est vrai que le maintien de la cohérence par le matériel induit un surcoût en surface et consommation, mais ceci est justifié par son efficacité et par sa simplicité d'usage.

Dés lors que l'accès aux données devient critique, l'évolution technologique remet en cause les solutions proposées. Ainsi les protocoles n'échappent pas à cette évolution et il convient d'en ré-évaluer certains.
Les deux classes de protocoles les plus couramment utilisés sont :

– Protocole d'espionnage de bus
  L'idée est que chaque cache espionne les requêtes de cohérence émises par les autres caches afin de mettre à jour l'état de ses lignes. Cette classe de protocole est simple à implémenter et sa latence de mise à jour est faible. Néanmoins, il fait l'hypothèse d'un bus partagé comme moyen d'interconnexion et c'est le principal désavantage. En fait, la bande passante d'un bus est limitée et elle est partagée entre les cœurs. Le nombre de ces derniers dépasse difficilement la dizaine.
– Protocole à base de répertoire
  Implanté du côté du contrôleur mémoire, un répertoire global gère l'état de partage des lignes de cache. Ce répertoire contient le nombre et la localisation de copies de chaque ligne. C'est le point de sérialisation des requêtes de cohérence. La plupart des architectures multiprocesseurs industrielles ayant plus d'une dizaine de cœurs

implémentent ce type de protocole. Cette scalabilité a un coût de mémorisation important (en terme de surface).

La tendance des systèmes actuels est l'intégration d'un grand nombre de cœurs utilisant des interconnexions de types NoC. L'architecture de ces systèmes ne supporte pas le protocole de diffusion pour deux raisons simples. La première concerne le nombre de cœurs qui ne cesse d'augmenter et le trafic qu'ils généraient sur les liens d'interconnexion. La deuxième est due au réseau qui ne sérialise pas les transactions et qui nécessite des supports matériels dédiés permettant la transmission des requêtes dans l'ordre. Ainsi, les protocoles de diffusion et de multi-diffusion ont été très vite écartés.

Cependant, des approches hybrides basées sur la diffusion ont révélé leur efficacité. L'hypothèse sur laquelle ces approches se sont appuyées est que seuls un petit nombre de nœuds impliqué dans la cohérence se prête à la diffusion (invalidations, gestion par région...). Dans le même contexte, la diffusion peut servir comme une technique de localisation de données dans un système distribué. Cette technique n'est qu'une partie intégrante du protocole de cohérence.

De ce fait, nous pensons que dans un système à plusieurs niveaux de cache (privée ou partagé), une forme de cohérence basée sur la diffusion ou la multidiffusion peut être efficace. Ainsi, la question qui se pose est la suivante :

– Comment assurer la cohérence des données en utilisant la diffusion dès lors qu'on s'appuie sur la nouvelle structuration de la hiérarchie mémoire ?

2.3 Diffusion : base du protocole de cohérence

Dans la section précédente, nous avons soulevé le problème de cohérence vis-à-vis de la nouvelle organisation des caches. Se pose alors un deuxième problème lié au protocole proposé comme solution : Le coût de la diffusion en terme de bande passante, latence et stockage.

Il est indispensable d'adapter le support de communication à la diffusion afin de garantir que la solution proposée n'est pas trop coûteuse.

2.3.1 Support matériel

Le support de communication dans les architectures multiprocesseurs doit s'adapter à la nature du trafic dont il a la charge pour des gains en performance. Ce support doit être efficace et flexible. Il est d'une forte importance puisqu'il gère les transactions entre les différents modules de l'architecture.

Le trafic induit par les protocoles de cohérence de caches et plus précisément la diffusion est un type de communication auquel on doit prêter attention. Puisque cette dernière peut être une source importante de dégradation de performance (en bande passante et en latence), y adapter le support de communication s'avère nécessaire. En effet, le trafic élevé engendré par la diffusion peut mener à la congestion du réseau : des paquets entrent en concurrence pour accéder aux mêmes ressources (ports de sortie, canaux physiques ou virtuels, ...). Ceci dicte, de prime abord, le besoin d'un réseau *ad-hoc*. Son enjeu majeur est de réduire la latence des transferts au niveau de la hiérarchie.

D'autre part, l'architecture classique des routeurs dans les réseaux sur puce n'est pas capable de gérer efficacement ce type de communication et ceci peut impacter le reste du trafic.

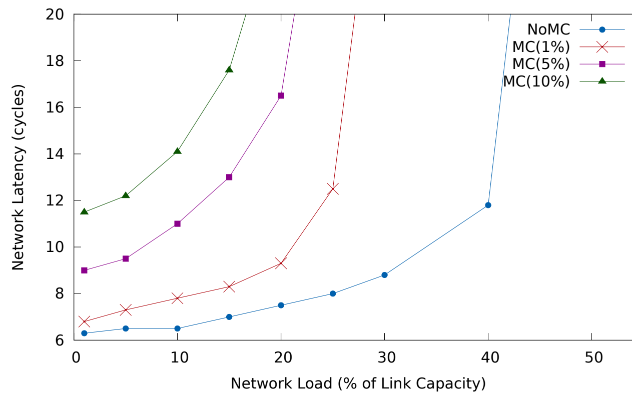


FIGURE 2.3 – Performance des multi-diffusions dans un réseau sur puce

La figure 2.3 illustre la performance d'un NoC 4x4 en maille 2D en effectuant des campagnes d'injections de paquets de type *uniform random* [JPL08]. Le réseau est très performant en l'absence de multi-diffusion. Une fois que l'on injecte des paquets destinés à des nœuds multiples, on remarque la dégradation. MC 1% consiste à convertir 1% des paquets en des paquets de multi-diffusion. Dans ce cas le point de saturation baisse de 40% à 25%. Ce point est la valeur de charge seuil à partir de laquelle le réseau atteint le régime congestionné. Pour un MC de 5% et un autre de 10% le réseau sature à 20% et 15% respectivement. Étant donné que le réseau n'est pas conçu pour gérer plusieurs destinations d'un seul paquet, la multi-diffusion est décomposée en un ensemble de mono-diffusions au niveau de l'interface réseau.

L'absence d'un routeur dédié qui gère la diffusion ou la multi-diffusion entrave le déploiement des protocoles qui nécessitent un tel type de trafic. L'enjeu de ce routeur est de réduire la latence moyenne du réseau, d'être économe en bande passante et d'être aussi simple que possible.

2.3.2 Algorithme de routage

Le routage n'est qu'une fonction d'ordonnancement spatiale sur le réseau, qui permet l'acheminement des messages de la source jusqu'à la destination. Les algorithmes de routage peuvent être classés en deux catégories :

– routage déterministe : ce type englobe les fonctions de routage qui associent une seule route reliant chaque paire (source, destination). Il est implémenté soit par des tables de routages, soit par des opérations arithmétiques sur les adresses. Son avantage réside dans la simplicité d'implémentation, mais son manque de flexibilité dans l'attribution des routes le rend sensible aux phénomènes de congestion ;
– routage adaptatif : la différence par rapport au précédent est que la route empruntée par le message n'est pas connue à l'avance, mais est déterminée au fur et à mesure de la propagation. Les critères guidant les choix de la route sont liés à la charge du réseau et l'idée est que les paquets évitent les régions du réseau déjà chargées. La flexibilité de ce type de routage vise une amélioration des performances globales, néanmoins son implémentation nécessite beaucoup de ressources matérielles, ce qui est un problème pour les réseaux intégrés, et il est

victime d'interblocages.
Concernant la diffusion et la multi-diffusion, le routage de leur paquets est d'une complexité importante. Cette complexité n'est pas liée au choix de l'algorithme de routage qu'il soit déterministe ou adaptatif. Elle est due, par contre, à la nécessité d'ajouter du matériel servant à choisir les ports de sorties pour un paquet, à modifier les destinations dans chaque en-tête de paquet et à éviter les interblocages.

En outre, le routage dépend fortement de la topologie du réseau qui traduit la manière dont les liens de communication connectent les différents nœuds du système. Plusieurs topologies plus ou moins complexes ont été proposées pour différents systèmes. La complexité de la topologie est liée au nombre de liens physiques qu'elle engendre, reflétant son coût, et la distance maximale entre deux éléments distants du système (diamètre du réseau) reflétant sa latence. Le choix de la topologie peut avoir un impact sur les performances du système notamment si le nombre d'éléments connectés est important et la latence est critique (cas du rechargement des lignes de caches dans les many-cœurs). Ces topologies convergent vers des formes régulières et structurées (maille, tore) dans le but de faciliter l'acheminement des paquets ainsi que le passage à l'échelle.
Nos questions concernant la diffusion sont les suivants :

- Comment adapter la structure du réseau afin de faciliter les interactions entre les modules de la hiérarchie mémoire ?
- Quel algorithme de routage permet d'implanter la stratégie de diffusion et quelles les modifications cela implique au niveau du support matériel ?

2.4 Conclusion

Dans ce chapitre, nous avons soulevé un certain nombre de problématiques concernant la hiérarchie mémoire qui s'articule autour d'un réseau sur puce dans le contexte des systèmes many-cœurs. Pour les niveaux de cache associés à chaque cœur, comment les manipuler afin d'augmenter l'efficacité d'accès ? Cette efficacité repose sur la latence, reliée à la localité des données, sachant que la gestion de la hiérarchie mémoire nécessite le maintien de la cohérence. Ces problèmes seront abordés par l'étude des besoins du réseau concernant la structure, le support matériel et l'algorithme de routage.

À la lumière des différentes difficultés analysées, on doit concilier les différents aspects qui accroissent les performances. Nous énumérons ainsi les questions auxquelles nous apportons des réponses dans cette thèse :

- En s'appuyant sur les capacités des réseaux sur puce, quelle organisation de la hiérarchie mémoire doit-on adopter pour avoir des accès efficaces ?
- Comment placer les données dans les bancs mémoires afin de minimiser le trafic sur la puce, en gardant à l'esprit la contrainte de capacité limitée de ces bancs ?
- Comment assurer la cohérence des données en utilisant la diffusion dès lors que l'on s'appuie sur la nouvelle structuration de la hiérarchie mémoire ?
- Comment adapter la structure du réseau afin de faciliter les interactions entre les modules de la hiérarchie mémoire ?
- Quel algorithme de routage permet d'implanter la stratégie de diffusion et quelles modifications cela implique-t-il au niveau du support matériel ?

Chapitre 3: État de L'Art

Sommaire

2.1 Performance des systèmes multiprocesseurs intégrés 5
2.1.1 Organisation de la mémoire 6
2.1.2 Accès aux données 7
2.1.3 Structure de communication 7
2.2 Hiérarchie des mémoires caches 7
2.2.1 Mémoire partagée hiérarchique 7
2.2.2 Partagé-distribué : NUCA 8
2.2.3 Cohérence des données 10
2.3 Diffusion : base du protocole de cohérence 11
2.3.1 Support matériel 11
2.3.2 Algorithme de routage 12
2.4 Conclusion 13

La gestion de la hiérarchie des mémoires caches est un problème clé pour l'optimisation des performances des systèmes dès lors que l'on aborde les architectures parallèles. Évidemment, ces architectures sont organisées généralement autour d'un réseau sur puce qui peut influencer la latence d'accès aux données. Donc, toute solution concernant l'accès aux données doit tenir compte des paramètres du réseau afin d'exploiter efficacement ses caractéristiques. Une exploration de l'espace des solutions de ces réseaux permet de souligner l'intérêt de la technique de virtualisation. Ses atouts nous ont poussé à utiliser ce concept dans le cadre de l'organisation de la hiérarchie des mémoires caches. Ce chapitre est organisé comme suit : la section 3.1 rappelle les principes généraux de la hiérarchie des mémoires caches, la section 3.2 présente l'architecture NUCA sur laquelle s'appuie nos recherches. Nous passerons en revue les différentes techniques proposées pour la gestion des mémoires caches dans ce type d'architecture et nous discuterons ses limites. Comme cette architecture est centrée sur les réseaux sur puces, la section 3.3 est focalisée sur les travaux qui s'y rapportent. Ensuite, la virtualisation, ses stratégies et les solutions qui visent à améliorer les performances du système feront l'objet de la section 3.4. Finalement, nous nous positionnerons vis-à vis de ces solutions.

3.1 Hiérarchie des mémoires caches

3.1.1 Généralités

Le but des mémoires caches est de cacher la latence entre la mémoire principale lente et les processeurs qui sont plus rapides. Les caches sont interrogés en priorité quand un processeur requiert une donnée de la mémoire. Si la donnée se trouve dans le cache, il s'agit d'un succès. Dans le cas inverse, si la donnée est absente, c'est ce qu'on dénomme défaut de cache. La requête est alors relayée à la mémoire principale. L'organisation de l'espace du cache peut conduire à trois types de défauts :

- les défauts obligatoires correspondent à une première référence à une donnée, cette dernière n'a jamais été demandée donc elle ne peut pas avoir été insérée dans le cache.
- les défauts de capacité sont dus à la taille limitée du cache. Le cache est plein, il est impossible d'insérer la donnée requise sans en évincer une déjà présente.
- les défauts de conflit sont reliés à l'associativité. Le nombre de places dans laquelle une certaine donnée peut être placée est limité. En conséquence, malgré la présence de lignes potentiellement vides dans le cache, l'ensemble accueillant la ligne est plein et il faut évincer une ligne pour accepter la nouvelle donnée. Ainsi, les caches totalement associatifs ne sont pas soumis à ce type de défauts.

Il existe toujours un compromis entre la taille du cache et sa latence. Un gros cache à l'avantage de réduire le nombre d'accès à la mémoire principale, cependant la latence d'accès est très importante. Un plus petit cache est doté d'une rapidité lui permettant de soutenir la cadence des requêtes du processeur, mais le faible espace de mémorisation favorise les accès à la mémoire principale qui sont coûteux en terme de latence. La solution adoptée est l'utilisation de plusieurs niveaux de cache.

Organisés en hiérarchie, les caches sont interrogés par le processeur dans l'ordre, du plus proche au plus éloigné. En effet, actuellement les processeurs possèdent plusieurs caches organisés en niveaux. Le niveau le plus proche d'un cœur nommé L1, est le plus petit. Sa taille se justifie par le fait que près du cœur il n'y a pas assez de place pour une grosse mémoire, et que, de plus, le temps nécessaire pour la recherche du bloc dans le cache est une fonction croissante de la taille de ce dernier. Les niveaux suivant disposent de plus d'espace étant plus éloignés des cœurs, ils sont donc de plus en plus gros. Notons que le terme LLC désigne le niveau de cache le plus éloigné des cœurs et il représente souvent le troisième niveau (L3).

La différence de performance combinée avec le besoin accru d'accéder à la mémoire fait qu'une gestion efficace de la hiérarchie mémoire est cruciale pour obtenir un système performant. Le problème de la performance d'une hiérarchie mémoire peut être attaqué dans de nombreuses directions : la forme de la hiérarchie (nombre de niveaux, cache spécialisé, ...) ; la gestion de la hiérarchie ; l'organisation des différents niveaux dans la hiérarchie.

3.1.1.1 Forme de la hiérarchie

La forme est liée aux nombre de niveaux dans une hiérarchie. Le cas le plus classique est celui d'un processeur avec deux ou trois niveaux de cache embarqués sur la puce. Chaque niveau de la hiérarchie contient un sous-ensemble de la mémoire principale, d'accès plus rapide que cette dernière. Leur contenu est déterminé par les accès du

processeur ainsi que la politique d'éviction. Ces caches, qu'ils soient séparés ou unifiés, contiennent les instructions et les données. Il est très courant que les caches de premier niveau soient séparés, alors que pour les niveaux qui suivent, ils sont généralement unifiés (et de plus grande associativité).

3.1.1.2 Politiques de gestion de la hiérarchie

Concernant sa gestion, le contenu d'un cache vis-à-vis du contenu du niveau suivant de la hiérarchie se voit contraint par des politiques telles que l'inclusion d'un niveau dans le niveau suivant ou l'exclusion des contenus de niveaux de cache différents. On parle de hiérarchie non-inclusive si aucune des précédentes politiques n'a été mise en œuvre.

3.1.1.3 Organisation : caches privés ou partagés

L'organisation des différents niveaux de cache est d'une grande importance dans la conception des systèmes multiprocesseurs. Le coût et la capacité des mémoires caches contraignent les choix du concepteur.

Cache partagé

L'approche à cache partagé consiste à créer un niveau de cache, généralement le dernier niveau, large et partagé entre plusieurs cœurs. La grande capacité de mémorisation de ce cache permet de répondre aux besoins des cœurs qui lui sont associés. Afin de minimiser la charge en accès à la zone partagée, dans la structure à cache partagé, les caches L1 privés s'occupent de filtrer les requêtes d'accès pour chaque cœur. En effet, lors d'un défaut au niveau de cache L1 la requête est envoyé vers le cache partagé du niveau inférieur.

Pour cette organisation, les données ne sont plus dupliquées au niveau du cache partagé. Cependant, une donnée peut avoir plusieurs copies dans des caches L1 privés. Ceci nécessite le maintien de la cohérence des données entre les caches L1s et le cache partagé.

Pour les systèmes à petite échelle (moins de 10 cœurs) les techniques de gestion des mises à jour des données sont les suivantes :

- Écriture différée (write-back) : la mise à jour de la donnée dans le cache partagé n'est effective que lorsque la copie de la donnée est remplacée ou invalidée.
- Écriture immédiate (write-through) : chaque modification de la copie d'une donnée dans un cache L1 est rapportée lors de l'écriture à la copie principale de la donnée dans le cache partagé.

Les limitations de ces techniques sont liées au trafic et à la dissipation thermique qui en résultent dans le cas des systèmes à grande échelle (des dizaines de cœurs).

Des mécanismes de cohérence ont été proposés avec l'introduction des techniques de communication directe entre le cache partagé et les différents cœurs. Une des techniques les plus répandues est celle basée sur des répertoires [CF78]. En fait, chaque donnée est associée à un répertoire qui se charge de garder des informations relatives à la donnée. Ce répertoire note pour chaque ligne quels sont les cœurs qui possèdent une copie valide, et qui seront donc les cibles des requêtes d'invalidation.

Les avantages de l'approche à cache partagé sont les suivants. Un premier concerne l'espace de mémorisation disponible qui peut être utilisé efficacement par plusieurs cœurs. Une seule copie est maintenue au niveau du cache partagé et celle-ci est sollicitée par plusieurs cœurs ce qui implique l'optimisation de l'utilisation de l'espace cache et l'augmentation des performances [BJM11].

Néanmoins, l'interférence due aux trafics générés par l'ensemble des cœurs influe sur les performances en engendrant des défauts de cache. La gestion de la cohérence ajoute également un surcoût. De plus, le temps d'accès aux données dans le cache partagé est en moyenne très long [DS07, ZIUN08].

Cache privé

Une alternative à l'approche précédente est l'organisation en caches privés où chaque cœur est doté d'au moins de deux niveaux de cache privées : un niveau L1 et un niveau LLC. L'accès dans ce cas est local, direct et rapide. En effet, dans le cas d'un échec au niveau du cache L1, la requête est envoyée vers le cache LLC. Vu que les caches LLC sont indépendants et ils ne sont accessibles que par leur cœur local, les accès aux données sont rapides. De plus, la structure des caches ne nécessite pas une gestion de cohérence, ce qui permet d'augmenter les performances d'accès.

Toutefois, les données accédées par les différents cœurs sont dupliquées sur plusieurs LLCs privés et ceci dégrade le taux d'utilisation de la mémoire sur la puce. Une autre limitation est liée à l'efficacité de gestion de l'espace cache qui est due à l'allocation statique du cache. En effet, le déséquilibre de charge des cœurs induit un déséquilibre dans l'utilisation des caches LLC : un cœur peut avoir besoin de plus d'espace de mémorisation que ce qui est disponible dans son cache LLC privé, tandis qu'un autre peut ne pas utiliser la totalité de son cache.

En s'appuyant sur les aspects positifs et négatifs des deux modèles d'organisation de cache précédents, la section 3.2 présente une technique qui permet de combiner le modèle à cache privé et celui à cache partagé dans le but d'augmenter l'efficacité de mémorisation sur la puce.

Approche hybride

L'approche hybride regroupant les deux modèles précédents est la solution permettant le passage à l'échelle tout en garantissant de bonnes performances. Elle permet de combiner la minimisation de la latence d'accès aux données grâce aux mémoires caches privées et la réduction du nombre de défauts de cache en autorisant, comme dans l'approche à cache partagé, l'échange des données partagées entre les caches privées distants. Une telle approche a l'avantage d'améliorer la disponibilité des données dans le système (grâce à des caches plus grands) et leur accessibilité (grâce à la qualité du réseau de communication entre les nœuds).

Dans le contexte des systèmes multicœurs sur puce, Nous nous intéressons aux approches hybrides qui sont basées sur des architectures NUCA.

3.2 Architecture NUCA

À l'ère des architectures multiprocesseurs, l'accès aux données est un point clé de la performance des systèmes. Or plusieurs études ont montré qu'avec un nombre de processeurs croissant, l'usage d'un cache L2 partagé n'est pas envisageable. Ainsi, les architectes ont proposé des nouvelles hiérarchies mémoire afin d'améliorer la latence des accès.

Les architectures NUCA sont les premières propositions dans lesquelles les mémoires caches de second niveau sont divisées en petits blocs appelés bancs [KBK02]. Le temps d'accès à ces blocs diffère de la perspective d'un processeur à un autre. Le défi de la conception d'une telle architecture est l'emplacement des données. En fait, nous souhaitons que les données fréquemment utilisées soient les plus proches du demandeur. La réplication, la migration, la coopération entre caches, etc, sont des mécanismes proposés pour répondre à ce défi et ils seront détaillés dans la suite.

NUCA peut être configurée de deux manières différentes : l'une statique nommée S-NUCA et l'autre dynamique appelé D-NUCA. S-NUCA est une conception conservative qui réduit l'utilisation du réseau et l'énergie au détriment de la performance, tandis que D-NUCA est un design agressif qui négocie l'utilisation du réseau et de la consommation d'énergie pour des performances élevées.

S-NUCA (*Static-NUCA*) utilise un placement et une méthode de recherche de données statique, où les bancs contiennent un ensemble d'adresses contiguës. L'aspect statique garantit une efficacité en énergie et une faible utilisation du réseau. Mais cela n'implique pas que le système soit performant car les données pourraient être situées dans des bancs distants du demandeur, de plus un même banc pourrait être utilisé concouramment par plusieurs processeurs. C'est l'architecture la plus utilisée pour les architectures CMP commerciales actuelles [KBM+10].

D-NUCA (*Dynamic-NUCA*) vise les bonnes performances. Ainsi, un cache L2 est un ensemble de bancs partagés par tous les processeurs avec un mappage dynamique de données [CPV03]. Le concept de ce mappage est mis en œuvre à travers les techniques de placement, migration et de réplication [BMW06, BW04, CPV05, ZA05a, MPG10].

Ces approches sont associées à plusieurs techniques de recherche de données. Une première est basée sur une comparaison partielle des étiquettes, ces derniers sont stockés dans un tableau au niveau du contrôleur des caches [KBK02]. La difficulté principale est due à l'augmentation de la complexité de la structure du cache. La solution à ce problème était l'utilisation de filtres permettant de stocker des informations compressés et ainsi de libérer de l'espace. Dans ce contexte, Ricci et al [RBGB06] s'appuient dans leurs travaux sur les « filtres de Bloom » pour vérifier si le bloc existe dans un ensemble des bancs du cache. L'autre orientation des travaux est la réduction des latences d'accès. Bischewski et al.[BPG05] ont exploré l'effet de prédicteurs de bancs et ils ont conclu qu'une prédiction précise est nécessaire pour avoir de bonnes performances.

3.2.1 Réplication

Pour la réplication, les données fréquemment utilisées sont générées dans plusieurs emplacements, ce qui permet d'économiser la latence au détriment d'une capacité réduite. Cette approche est basée sur des bancs de cache privés.

Plusieurs études ont évalué les avantages et les limites de la réplication dans le

contexte des multiprocesseurs. Huh et al. [HKS+05] ont étudié l'effet de variation du degré de partage des caches pour divers applications et ils en ont déduit qu'il y a un avantage à faire quelques réplications au sein des bancs de cache. Ensuite, le principe d'optimisation de la réplication a également été étudié dans les travaux de Chisti et al.[CPV05] pour réduire les réplicas inutiles. Ils ont essayé de moduler la capacité des différents caches en fonction des besoins des nœuds. Dans le même contexte, Zhang et al.[ZA05a] présentent une nouvelle politique qui consiste à répliquer les victimes des caches L1 au niveau du banc de cache L2 local. Dans leurs travaux, Chisti et Zhang adoptent chacun une stratégie de réplication : le premier réplique la donnée au moment de l'accès alors que le deuxième la réplique au moment de l'évincement. Le mécanisme de réplication dans les travaux précédents est statique. Ainsi, la dégradation des performances pour certaines applications est justifiable. Dans le but de pallier cette dégradation, des travaux ont été menées ajustant dynamiquement la réplication au comportement de l'application, on peut citer en particulier ASR, R-NUCA, PSA-NUCA.

ASR (*Adaptive Selective Replication*) est une technique qui contrôle le taux de duplication des données partagées. En effet, les auteurs estiment dynamiquement le coût (en terme de défaut de cache) et l'avantage (en terme de latence d'accès) d'une réplication d'un bloc en analysant le comportement de l'application. Cette technique est basée sur une probabilité et un mécanisme chargé de l'ajustement de cette probabilité.

R-NUCA (*Reactive-NUCA*) est un autre modèle proposé par Hardavellas et al. [HFFA09]. Dans leur étude ils ont montré que les accès au cache L2 générés par l'exécution d'un ensemble de benchmarks (selon le type : serveur web, programme scientifiques...) peuvent être classés en plusieurs catégories et chacune sera le sujet d'une stratégie de placement des blocs mémoires. L'architecture matérielle utilisée pour leur étude est une architecture multi-tuiles de 16 cœurs, et le simulateur est Flexus [WWF+06]. Suite à l'analyse des accès au cache L2 pour les applications de type serveur web, les auteurs ont trouvé que les accès aux instructions et aux données partagées dominent la majorité des accès au cache L2. Pour les instructions, la stratégie de réplication est souhaitable. Pour les données partagées, les stratégies de réplication ou de migration sont inutiles. Ainsi, ils ont proposé une architecture nommé R-NUCA qui regroupe logiquement les tuiles afin d'y appliquer les différentes stratégies. Les instructions sont répliqués par groupe de tuiles et ils sont entrelacés d'une manière rotationnelle à l'intérieur d'un seul groupe. Alors que les données partagées sont entrelacés dans l'ensemble des tuiles. Un système d'exploitation est chargé de la localisation des blocs mémoires, du classement des accès et du regroupement des tuiles.

PSA-NUCA est une technique qui repose sur l'aspect de la distribution asymétrique des accès mémoires pour les applications [HGG+12]. Elle englobe les techniques de remplacement, de mappage et de réplication en s'appuyant sur une métrique qui représente le nombre d'accès à chaque ensemble (set) du cache : « l'information de pression ».

Le principe de la réplication adaptative a montré son efficacité dans les systèmes multiprocesseurs. En revanche, une limitation majeure de l'approche de réplication est le maintien de la cohérence des copies. En substance, les formes de cohérence sont diverses, mais aucune n'est à la fois simple et performante.

3.2.2 Migration

La technique de placement et migration consiste à migrer le bloc vers un banc plus proche du demandeur. L'étude de Beckmann et al.[BW04] montre que la migration dynamique des blocs est performante mais elle exige un algorithme de recherche et elle provoque la contention de quelques bancs de cache. En effet, dans les systèmes multiprocesseurs les blocs sont partagés par plusieurs processeurs et la demande de ces blocs peut induire des migrations excessives. Le bloc en migration est instable, ce qui augmente le taux de défauts de cache. Donc, un mécanisme nommé « migration paresseuse » pour gérer ce problème a été proposé. Dans [ZA05b] une stratégie de migration similaire à la stratégie "*First-Touch*" soumise à des conditions pour le déplacement des données au demandeur a été présentée.

En partant du constat que la migration des données permet de résoudre le problème du placement de données partagées, un algorithme de migration pour les applications parallèles a été proposé dans [KLIS08]. Ce dernier vise à optimiser la localisation des différents blocs durant l'exécution de l'application. L'algorithme, basé sur un modèle de placement, exploite la variance des latences pour les différents emplacements des blocs.

Une autre recherche est basée sur le comportement des applications. Hammoud et al. [HCM09] ont développé un système de migration entre les caches exclusifs L2. Il s'agit d'un mécanisme matériel, ACM (*Adaptive Controlled Migration*) qui s'appuie sur une prédiction. Le système recueille les informations concernant les blocs accédés par les différents processeurs ainsi, en supposant que ces processeurs auront à nouveau accès à ces blocs, le bloc sera migré à un banc assurant la latence minimale.

Récemment, [HBH12] présente une stratégie dynamique de gestion des caches qui vise à réduire les contentions entre les partitions. Une adaptation efficace des techniques de migration et d'insertion a été illustrée afin de satisfaire les exigences des différentes applications. Il consiste à déterminer lors de l'exécution s'il faut ou non migrer le bloc d'un cache distant au cache local. La décision est basée sur les rapports de succès dans les caches locaux et distants.

Une autre méthode qui vise d'étendre D-NUCA aux CMPs basée sur les tuiles [DK15]. L'idée est d'autoriser la migration des blocs au niveau d'un ensemble de bancs et d'ajuster la technique de remplacement sans affecter les performances. Il s'agit d'utiliser une technique mixte de recherche : sa première étape est de consulter le banc le plus proche du demandeur. À l'échec, la requête sera diffusé aux restes des bancs du même ensemble.

3.2.3 Cache Coopératif

Le principe du cache coopératif consiste à créer un espace de mémorisation logiquement partagé à travers le regroupement de plusieurs caches privés. Les cœurs d'une même zone de coopération ont un accès étendu aux caches des autres cœurs de la même zone. Les données sont accessibles à travers un espace d'adressage commun. Le mécanisme de cache coopératif pour les systèmes multicœurs sur puce a été proposé dans les travaux de Chang et Sohi [CS07]. Il consiste à adapter l'utilisation des ressources par les cœurs d'une manière dynamique en fonction de leur besoin en mémoire.

Le mécanisme de coopération est mis en œuvre à l'aide d'unités matérielles spécialisées dans le contrôle de l'allocation des données dans les caches. Les données stockées dans la zone coopérative sont gérées par l'intermédiaire d'une unité de contrôle de

cohérence de caches. Chang et Sohi [CS06] ont proposé dans leur modèle de cache coopératif que la gestion de la cohérence soit centralisée. Elle est assurée par un répertoire commun CCE (*Centralized Cooperative Engine*) comportant la duplication de toutes les informations de cohérence des caches L1 et L2 (figure 3.1). Lors d'un défaut d'accès au cache local, la requête est redirigée par le CCE vers le cœur en possession de cette donnée ainsi cette dernière est transférée au demandeur.

Lorsqu'un bloc du cache L2 est évincé, le CCE transfère ce bloc à un autre cache de la même zone partagée. Pour une meilleur exploitation de l'espace de mémorisation, le contrôleur de cohérence utilise une politique de remplacement dite *N-chance forwarding* [HGC10b], qui permet d'éviter la circulation infinie d'un bloc sur le réseau. Cette technique consiste à associer un compteur à chaque bloc de données. Le bloc est autorisé à migrer N fois avant son éjection hors puce. L'initialisation du compteur est faite à chaque recharge de la donnée sur la puce.

Néanmoins, les limitations de l'approche de cache coopératif à répertoire centralisé sont multiples. Une première concerne le passage à l'échelle du répertoire. La deuxième est liée à la consommation d'énergie au niveau de l'unité de contrôle centralisée dont la charge augmente avec le nombre de cœurs. Pour répondre à ces limitations une nouvelle approche a été proposée : une stratégie de gestion de cohérence de données répartie appelée DCC (*Distributed Cooperative Caching*) [HGC08].

Le mécanisme du DCC consiste à diviser l'unité de contrôle CCE en plusieurs unités réparties DCC, et chacune s'occupe d'une plage d'adresses (figure 3.1). Les DCC comportent des informations sur la distribution des blocs au niveau des cœurs. Ce mécanisme a été conçu pour résoudre le problème de passage à l'échelle de la configuration centralisée.

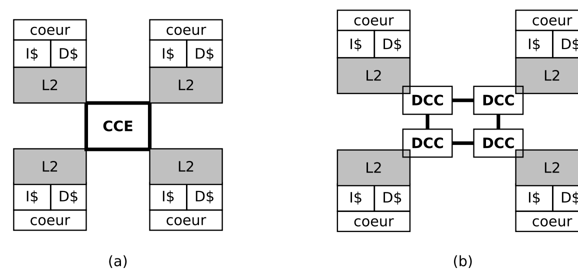


FIGURE 3.1 – Structure de cache coopératif à contrôle (a) : centralisé, (b) : distribué

Une autre approche apparait dans le contexte des grappes de calcul multicœurs : CCM (*Cluster Cache Monitor*)[LTL+13] . Cette approche qui tire profit de la propriété de données voisines et donc de l'aspect cluster, est basée sur les propriétés des programmes parallèles. Similaire au mécanisme du DCC, la gestion de cohérence est assurée via un répertoire mais sa taille est beaucoup plus réduite (*cluster-based sharer information*) en la comparant à celle du DCC.

Pour un meilleur passage à l'échelle de la mémoire sur puce pour les architectures manycœurs, une nouvelle technique de coopération des caches a été proposée par Herrero et al.[HGC10a], le cache élastique. Cette technique s'appuie sur une division logique des caches LLCs en deux zones privée et partagée afin d'optimiser leurs utilisation. La

zone privée stocke les données éjectées du niveau plus haut. Alors que la zone partagée accueille les données provenant du voisinage. Ce partitionnement est adapté dynamiquement aux comportements des applications. Ce mécanisme vise équilibrer l'utilisation de l'espace de mémorisation, malgré la différence en besoin mémoire des différents cœurs. Une autre étude similaire à la précédente nommé CloudCache [LCC11] implémente une extension de cache privé qui coopère avec les bancs de cache L2. Dans cette étude l'espace d'adressage est partitionné en plusieurs sous espaces d'adressage adaptés à des types de données spécifiques.

En s'inscrivant dans le débat de l'interférence des threads pour les accès aux caches et dans le contexte du partitionnement spatial des caches, Dybdahl and Stenstrom[DS07] proposent un algorithme dynamique d'organisation de ces derniers en deux parties : privée et partagée. Cette technique nécessite un dispositif partagé qui partitionne l'espace de cache et contrôle l'éviction des blocs dans tous les bancs partagés. La décision est prise par cet engin après un certain nombre de défauts de cache. De plus, sa scalabilité est limité par l'augmentation de la taille des caches ou l'augmentation du nombre de cœurs. Pour pallier à ce problème, Merino et .al [MPPG08] proposent un mécanisme nommé SP-NUCA (Shared/Private-NUCA) capable d'ajuster le partitionnement des bancs de cache selon le comportement de l'application. Il s'agit d'une division dynamique des ensembles (sets) du banc de cache L2 en des ensembles privés ou partagés. Grâce à des simples modifications au niveau de l'algorithme de remplacement, les données privées sont placés proche de leur demandeur. Cette approche réduit la latence vu que les interférences induites par les processeurs pour les accès aux données privées ont diminué. Cependant, ce mécanisme hérite les limitations des caches privés et partagés. Une concerne le déséquilibre dans l'utilisation des ensembles (sets) privés et la deuxième est reliée à la latence d'accès aux blocs partagés et qui réside dans des bancs distants du demandeur. Dans le but de répondre à ces limites, une extension du précédent a fait apparaître un mécanisme nommé "ESP-NUCA" [MPG10]. Ce dernier supporte des réplicas (une copie des blocs partagés est stockée dans la partition locale du cache) et des victimes (les blocs privés dans des bancs distants sont stockés au niveau de la partition de cache partagée). Le majeur inconvénient de ce mécanisme et la gestion de la cohérence des réplicas (hardware overhead).

3.2.4 Discussion : limitations de la gestion du contenu des caches

Les techniques discutées précédemment s'intéressent à la gestion du contenu des mémoires caches. Ils ont démontré des performances variables en fonction de la configuration du système et du comportement des applications. La réplication favorise les accès locaux et la répartition des charges mais elle impacte l'espace de mémorisation. Son aspect adaptatif nécessite l'intervention d'un OS, une solution qui n'est pas purement matérielle. La migration permet de résoudre le problème de placement des données mais elle est coûteuse et elle peut être préjudiciable pour les données partagées. La coopération combine l'avantage de réduire la latence d'accès aux données grâce à l'organisation privée et l'avantage de diminuer le nombre de défauts de cache en autorisant l'échange des données. Ses limites sont reliées au surcoût de la gestion de cohérence.

En partant du constat que la gestion des caches est probablement plus efficace si ces techniques exploitent les caractéristiques du réseau sur puce, la section qui suit présente les différentes techniques proposées dans ce contexte.

3.3 Autour des NoCs

La prise en charge de la localité des accès mémoire est fortement liée à la performance du réseau sur puce autour duquel s'articule les ressources (cœurs, caches, mémoire). Plusieurs approches ont été proposées tirant profit des spécificités du réseau. L'une des premières approches vise à réduire les délais de transmission des données entre les différents nœuds. Une des solutions consiste à utiliser le mécanisme de commutation de circuits [MTCM05]. Le chemin de communication entre une paire de nœuds est établi avant que les transmissions puissent avoir lieu. L'avantage de ce mécanisme est que la latence de transmission est connue à l'avance et il est bien adapté aux transferts de données importantes. Or, une fois la connexion établie dans ce mécanisme, les éléments du réseau sont alloués et ne peuvent pas être utilisés par d'autres unités. De plus, ce mécanisme qui nécessite des contrôleurs centraux est coûteux en terme de surface et il n'est plus flexible. Par conséquent, la recherche a été focalisée sur une alternative : le mécanisme de commutation de paquets. Les données dans ce cas sont encapsulées dans des paquets provenant de la source avant leur envoi à la destination. Son avantage est la performance maximale du réseau vu que la commutation est locale, il est entre deux routeurs et non pas entre deux unités de traitement.

Des travaux ont combiné l'utilisation des deux modes de commutation. Pour cela il existe ceux qui définissent deux réseaux séparés : un pour la commutation de circuit et le deuxième pour la commutation de paquet [PPCR12]. D'autres incluent les deux types en utilisant un même réseau [JPL08, AJM12]. Des techniques qui allouent des ressources à l'avance permettant une transmission de données en un temps réduit [LJ13].

La suite des travaux se concentre sur l'architecture du routeur. Une première approche implique les routeurs spéculatifs qui utilisent des chemins sans réservation préalable. Elle maximise l'efficacité des ressources de mémorisation dans les routeurs. La seule condition est l'absence de contention. L'architecture de ces derniers est complexe et peut induire des pénalisations en énergie et en performance [MWM06].

Dans le mode de commutation de paquets, les routeurs ont besoin d'éléments de mémorisation pour stocker les paquets. La stratégie la plus répandue est celle basée sur des canaux virtuels : des files d'attente associées à chaque canal physique permettant de diviser la bande passante du canal. Ainsi, dans [KPKJ07], Kumar et al. introduisent la notion des canaux virtuels « express ». Ils organisent l'ensemble de ces canaux en deux groupes, un normal et un deuxième formé des canaux expresses (EVCs). Le paquet circulant dans ce type de canal, suit un chemin prédéfini entre source et destination, et il traverse des nœuds court-circuités caractérisés par un délai de transmission minimal.

La technique précédente a été étendue par Krishna et al. [KKC+08]. Ces derniers ont constaté que la limitation de EVC est le temps mis pour l'allocation des files d'attente lors de la traversée du chemin express. Ils ont donc proposé l'utilisation d'un niveau de contrôle pour des configurations à l'avance. Tous les travaux qui ont été cités s'appuient sur des canaux prioritaires.

ReNoc [SS08] est le fruit d'une étude qui vise la reconfigurabilité de la topologie du réseau selon le besoin de l'application. L'idée est d'utiliser un multiplexeur entre les sorties du routeur et les lignes d'entrées afin de court-circuiter le routeur. SMART [CPK+13] est une technique qui définit un canal de court circuit autour des files d'attentes en entrée. Le crossbar dans ce routeur reçoit la donnée de la part de ce canal ou de la file d'attente en entrée. LOCO [KKP14] est une approche qui se sert du routeur

SMART pour une diffusion efficace des messages. Un réseau hiérarchique où chaque routeur est attaché à une grappe de composants (cluster) tels que le cœur, un banc mémoire... À la recherche d'une donnée, le cœur consulte son nœud *home* en traversant le canal de court-circuit, ainsi que les autres par une diffusion via le réseau virtuel.

À la différence des travaux précédents qui demandent une pré-configuration et qui utilisent des multiplexeurs au niveau des ports d'entrées ou dans le routeur, FP-NUCA [AHS+15] repose sur un routeur nommé "Freeze" et un concept de chemins prédéfinis caractérisés par leur rapidité. Ce routeur dispose d'un circuit de validation d'horloge qui gèle le fonctionnement du routeur (les registres du pipeline) temporairement afin d'arrêter la circulation des messages. Au cours de cette phase, les éléments de mémorisation continuent à stocker les messages. Au niveau des ports de sorties, des multiplexeurs sont utilisés pour choisir entre les messages en provenance du chemin rapide ou du chemin classique (étages du pipeline).

Les travaux illustrés ci-dessus tendent à optimiser l'utilisation du réseau sur puce au profit des mémoires caches. Toutes les améliorations ont été effectuées au niveau de l'architecture du réseau ainsi que de ses mécanismes pour réduire les délais de transmission et donc la latence de communication. Néanmoins, cette communication est affectée par l'organisation des caches et la modélisation isolée de ces derniers par rapport au réseau ne permet pas d'aboutir à de bonnes performances. De ce fait, des études de co-conception ont vu le jour et la section suivante les présente.

3.3.1 Co-conception mémoire et NoC

Nous présentons dans cette section les techniques qui ont été proposées pour que l'organisation des mémoires caches soit en concordance avec les réseaux sur puce.

Ortin et al. [OSV+14] se reposent sur le trafic induit par les mémoires locales lors de la conception des NoC pour améliorer l'efficacité du système. En effet, cette technique est inspirée de l'observation du modèle du trafic : la fréquence des motifs requête-réponse. Elle propose ainsi de construire dynamiquement le chemin de réponse en réservant les ressources à l'avance (lors de l'envoi des requêtes), une sorte d'anticipation afin de réduire la latence du réseau. Dans cette approche, tous les messages partagent les circuits. Une autre étude qui s'inscrit dans le même contexte [JKY07] a été élaborée pour adapter l'utilisation des ressources du réseau sur puce aux besoins des mémoires caches. Elle décrit, en fait, un modèle de routeur dédié qui supporte le mécanisme de "multicast" dans les caches. Ensuite, les auteurs ont proposé un algorithme de routage et une nouvelle topologie hétérogène nommé "halo" qui minimise le nombre de fils du réseau. Cette étude n'est pas limitée au réseau, une partie a été consacrée aux opérations liées aux caches. Il s'agit d'un algorithme "Fast-LRU" qui tend à réduire les latences induites par la politique de remplacement. D'autres travaux préconisent l'analyse des liens de communication ainsi que la topologie du réseau autour de laquelle s'articule les mémoires caches. Muralimanohar et Balasubramonian [MB07] introduisent une hétérogénéité dans les interconnexions ainsi que dans la topologie, qui utilise des connexions point-à-point avec un bus. Ensuite, avec Joupi [MBJ07] les auteurs précédents ont modifié l'outil Cacti pour qu'il supporte les caches NUCA et leur intervention a ciblé les fils et leurs types. Une autre étude a été menée par Foglia et al. [FMP05] dans laquelle l'architecture D-NUCA s'étend à TD-NUCA. Les bancs de cache sont organisés sous une forme triangulaire et le but est de réduire l'énergie au niveau des gros caches des processeurs embarqués.

Les travaux précédents ciblent des caches LLCs dont la taille est de l'ordre des megabytes, et donc les bancs de cache sont assez gros. L'approche L-NUCA de Suarez [SMV+09] vise les bancs de taille réduites, et consiste à intercaler des caches victimes entre les caches L1 et les LLCs afin de bénéficier la localité temporelle. Ils contribuent à réduire l'écart en latence entre les caches L1 et les LLCs et ils sont considérés virtuellement comme extension de la capacité des L1s. À la différence des NUCA, cette approche utilise des connexions spécifiques qui minimisent l'activité du réseau, ce qui réduit la consommation d'énergie. Cette approche a été étendue dans [SDM+12] afin de minimiser la consommation d'énergie.

Pour les architectures 3D, Chou et al. [CCW+09] proposent un réseau d'interconnexion avec une topologie en anneau qui relie les caches L1. La connexion est assurée par l'intermédiaire d'un anneau avec des connexions bidirectionnelles et un arbitre global pour gérer les accès simultanés. Cette approche permet de partager efficacement les caches L1 aux dépens de l'augmentation de la latence et de la complexité des accès distants dans les manycœurs.

3.3.2 Cohérence de caches et Réseau

Dans le même contexte et plus précisément pour gérer le trafic induit par le protocole de cohérence de caches qui peut impacter la performance du réseau, des supports spécifiques ont été implémentés au sein du réseau. Citons l'exemple du protocol hammer qui repose sur le mécanisme de diffusion (*broadcast*), un support est dédié à ce type de trafic. Dans [LFA12] Lodde et al. s'attaquent aux problèmes des acquittements (suite aux broadcasts) capables de ralentir les accès. Ils ont ajouté un circuit de contrôle qui collecte ce type de messages et les délivre en un temps fixe.

Eisley et al. [EPS06] implémentent un protocole de cohérence de caches au niveau du réseau d'interconnexion. Il s'agit d'intégrer un répertoire au niveau de chaque routeur qui gère et oriente les requêtes vers la donnée à proximité. Cette technique assure une optimisation dans les délais des accès mémoire ainsi qu'un meilleur passage à l'échelle. Dans [BGC+07] le réseau associe une priorité à chaque type de message. On distingue deux types : messages de contrôles et les messages de données. Également, Walter et al. [WCK08] explorent l'intérêt d'intégrer un bus personnalisé dans l'architecture du réseau sur puce. Ce bus est utilisé pour le broadcast, les messages de contrôles et pour l'échange des données de petite taille.

Volos et al. [VSG+12] présentent un design spécial du réseau qui répond aux besoins des applications grâce à une paire de réseaux asymétriques accordés aux différents types de trafic. La différence entre les réseaux réside dans la largeur du chemin de données, l'architecture du routeur, la stratégie du contrôle de flux et les délais. Cette architecture du réseau permet de réaliser des gains significatifs en énergie et des performances similaires à celles du réseau conventionnel.

3.3.3 Discussion

De nombreux travaux ont été effectués au niveau des réseaux sur puce à cause de la contribution importante de ces derniers sur le temps d'exécution. La majorité de ces travaux porte sur une conception conjointe des modules mémoires et du réseau qui les relie. Ces solutions touchent les liens de communication, la topologie du réseau, des supports

matériels dédiés à des trafics spécifiques, ... Ces solutions purement matérielles sont généralement coûteuses et spécifiques (non génériques). D'autres études d'optimisation réseau en faveur des protocoles de cohérence ont été menées. Ces études, bien qu'elles montrent leur efficacité, sont très coûteuse en terme de surface.

Afin de définir un meilleur compromis performance/coût, la solution doit être optimisée de façon à réduire les coûts matériels. La solution est d'exploiter le réseau physique avec efficacité. Ceci nous a poussé à réfléchir aux approches de virtualisation.

3.4 Virtualisation

3.4.1 Généralité

La virtualisation en informatique est l'ensemble des techniques matérielles et /ou logicielles permettant le fonctionnement de plusieurs systèmes d'exploitations et/ou plusieurs applications, séparés les uns des autres, sur une seule machine. Elle fait référence à l'abstraction physique des ressources.

Au départ la virtualisation a ciblé le marché des serveurs [PG74] et elle a cherché notamment la consolidation des serveurs grâce à sa capacité à partager le matériel. En effet, le serveur virtualisé est capable d'offrir plusieurs services qui fonctionnent de façon concurrente. Ensuite, la virtualisation est apparue dans le marché des ordinateurs de bureau, en particulier parce qu'elle est capable d'exécuter en même temps et de manière totalement transparente sur une même machine différents systèmes d'exploitation. Et récemment, la virtualisation envahit le marché de l'embarqué afin de répondre à ses problématiques. À titre d'exemple, la virtualisation peut permettre à un système de continuer de fonctionner, potentiellement en mode dégradé, si l'un de ses composants est défaillant.

Plusieurs architectes et concepteurs se tournent vers les solutions de virtualisation en raison des atouts supposés de ces technologies, tels que :

- une meilleure utilisation des ressources matérielles qui peut induire une économie d'énergie ;
- une meilleure disponibilité des systèmes (cas de panne matérielle) ;
- une meilleur répartition de charge, vu que la solution de virtualisation répartit la charge entre les différentes machines ;
- de potentiels gains d'encombrement.

3.4.2 Domaines de virtualisation

La virtualisation offre de nombreuses opportunités, de ce fait, il existe de nombreux types de virtualisation : d'application, de serveur, de stockage et de réseau.

La virtualisation d'application. C'est l'exécution d'un logiciel sans l'installer physiquement sur le système auquel on se connecte. Il s'agit d'une transformation de l'application en une donnée stockée quelque part dans l'attente d'être transportée, vers un système d'exploitation (serveur ou poste) où elle sera utilisée.

La virtualisation des Serveurs. Son principe est simple, le serveur est considéré comme un ensemble de ressources (CPU, RAM...) allouées de manière statique ou

dynamique aux serveurs virtuels. Son objectif est de mutualiser les capacités de chacun, afin de réduire les investissements en infrastructures physiques et de réaliser des économies. En effet, les serveurs sont souvent rassemblés dans des *datacenters*, or la majorité de ces salles doivent maintenir à titre d'exemple une température constante et un éclairage permanent [BBE+13].

La virtualisation du stockage. C'est une abstraction du stockage physique assurée par une décomposition de celui-ci en volumes logiques [MH07]. Cette technique est basée sur une couche logicielle qui intercepte les E/S.

La virtualisation du réseau. C'est le domaine le plus ambigu des approches de le virtualisation. Il combine les ressources réseau matérielles et logicielles dans une seule unité. Son objectif est de fournir un partage efficace et sécurisé des ressources. Nous nous focalisons dans la suite de ce document sur la virtualisation du réseau dans le contexte de l'embarqué : les réseaux sur puce.

3.4.3 Virtualisation du réseau sur puce

Un réseau virtualisé est un réseau qui peut se répartir en différentes régions, chacune servant une application et un flux de trafic. Ce réseau est caractérisé par une capacité à se reconfigurer sans aucune manipulation des objets physiques tels que les liaisons et les routeurs.

Ainsi, les propriétés fondamentales d'un réseau virtuel se résument en :
- la flexibilité et l'hétérogénéité permettant de construire des topologies arbitraires et hétérogènes constituées de nœuds et de liens virtuels paramétrables.
- l'isolation qui garantit une séparation stricte entre chaque réseau virtuel (VN) qui vit au sein d'une même infrastructure physique.
- la stabilité qui assure que des erreurs dans un réseau virtuel ne peuvent pas affecter un autre réseau virtuel.

Les enjeux à l'égard du réseau virtualisé consistent en une meilleure exploitation des ressources ; une utilisation des domaines de cohérence ; une augmentation du rendement de la puce ; et une réduction de la consommation d'énergie [FRD+08b].

3.4.3.1 Utilisation des ressources

Les applications qui ne fournissent pas suffisamment de parallélisme ne peuvent pas exploiter efficacement les architectures multicœurs. En fait, le déséquilibre dans l'utilisation des ressources implique leur gaspillage. Dans une telle situation, un mécanisme de partitionnement est nécessaire dans le but de gérer les ressources et de les affecter aux applications d'une manière efficace. Par conséquent, différentes applications peuvent s'exécuter indépendamment sur une même plateforme, avec l'amélioration de leur performance.

Ce mécanisme de répartition devrait répondre aux défis liés à la maximisation d'utilisation de ressources et à la prévention des conflits entre les messages appartenant aux différentes applications. Le partitionnement d'une puce multicœurs n'est autre qu'un regroupement de ressources, ce qui rend le problème similaire à celui de l'allocation de processeurs. La stratégie d'allocation contiguë attribue l'ensemble des ressources

adjacentes à une application. Pour les réseaux ayant comme topologie une grille 2D ou un tore, la stratégie traditionnelle permettant d'allouer des sous-grilles 2D ou des sous cubes, respectivement, renforce l'apparition de niveaux de fragmentation élevés. Cette fragmentation est due à un échec d'allocation en raison de l'exigence qu'un ensemble de ressources doivent pouvoir former une sous-grille.

Les algorithmes d'allocation contiguë ont une qualité attrayante : les ressources allouées à une tâche sont sélectionnées en fonction de l'algorithme de routage sous-jacent, de sorte que les liens ne sont pas partagés entre les messages. Ce type de routage confiné est d'un grand intérêt. En effet, malgré la concurrence des tâches, l'algorithme est efficace. Par exemple, la congestion au niveau d'une partie du réseau (liens, routeurs...) n'affecte pas l'exécution sur le reste du réseau.

Donc, le couplage étroit entre les stratégies d'allocation et les algorithmes de routage permet le développement de nouvelles approches de routage dans le contexte de la virtualisation. Ces approches impliquent l'assignation des régions ayant des formes irrégulières et l'utilisation d'un algorithme de routage indépendant de la topologie. Une autre propriété attractive est que ces approches sont tolérantes aux défauts dans une topologie régulière.

3.4.3.2 Rendement

La solution de virtualisation peut permettre d'accroître le rendement des systèmes sur puce. Rappelons que si la puce intègre des composants défectueux, le NoC doit être en mesure de fonctionner en prenant en compte ces défauts. Dans le cas où un bloc de la puce a été désactivé, le réseau virtualisé rétablit la situation par un regroupement des cœurs en régions. Dans le cas où un lien a été désactivé, deux solutions peuvent se présenter. Une première consiste à exclure ce composant par un partitionnement intelligent cloisonnant le NoC en différentes régions. La seconde vise l'ajout des mécanismes de tolérance aux pannes à des commutateurs pour contourner la défaillance dans une région.

3.4.3.3 Consommation d'énergie

La consommation d'énergie dans les systèmes multiprocesseurs est une préoccupation majeure et sa minimisation pendant la communication est étroitement liée aux protocoles développés pour la couche réseau (protocole de routage). Ainsi, les optimisations du trafic de l'application permettent d'assurer une courte durée d'exécution et donc réduisent cette consommation.

La virtualisation offre des solutions à cette issue de la même manière qu'elle se charge de l'augmentation de rendement lors d'une défaillance. Par ailleurs, les composants inactifs peuvent être mis hors tension grâce à la virtualisation. Or, dans ce contexte la technique de virtualisation doit être dynamique, contrairement à celle qui concerne le rendement qui pourrait être statique.

3.4.3.4 Domaine de cohérence

Pour un ensemble de cœurs qui exécutent des tâches, ceux qui partagent des données peuvent être regroupés en une région, le domaine cohérent. Une fois que les ressources nécessaires à ce mécanisme sont mises en place au niveau du réseau, le trafic induit par

le protocole est limité à la zone cohérente et il empêche l'interférence avec le trafic dans l'autre partie de la puce.

Dans les architectures multicœurs, la probabilité que les ressources soient partagées par différents domaines n'est plus négligeable. La virtualisation au niveau du réseau sur puce est en mesure de fournir des solutions pour limiter le trafic dans chaque domaine. On pourrait par example implanter dans un domaine cohérent des mécanismes assurant la diffusion.

Pour résumer, la virtualisation offre donc de nombreuses opportunités tels qu'une souplesse architecturale, une meilleure bande passante, une capacité à hiérarchiser le trafic pour atteindre les performances souhaités, et ce à moindre coût, en utilisant moins de ressources à un instant donnée, réduisant de ce fait la consommation d'énergie. Les techniques de virtualisation au niveau du réseau sur puce (NoC) peuvent aider à traiter aussi bien les problèmes de calcul que de communication.

3.4.4 Stratégies de virtualisation

Les deux principales techniques de virtualisation du réseau sont la virtualisation basée sur les canaux virtuels et la technique de partitionnement des ressources.

3.4.4.1 Canaux virtuels

Généralement les canaux virtuels servent à répartir les paquets en fonction de leur destination ou de leur priorité. C'est un ensemble de files d'attente qui gère le contrôle de flux. Bien que ces derniers impliquent une augmentation de la surface et de la latence, due à la mémorisation et au contrôle nécessaire, ces canaux, en plus d'éviter les interblocages et d'optimiser l'utilisation des liens, fournissent des services séparés à travers une séparation des trafics.

L'objectif est de virtualiser le trafic des applications en veillant à ce que les différents messages d'une application ne soient pas mélangés dans le réseau. Cette virtualisation est ainsi obtenue en garantissant que le trafic de différentes applications se sert des différents VCs [GKM+06]. En dehors de son potentiel d'assigner des priorités aux trafics suite à leur isolation, les caractéristiques des réseaux virtuels restent les mêmes.

L'inconvénient de cette approche est son besoin en ressources. Pour la virtualisation, des VCs différents sont nécessaires pour chaque application. Avec l'augmentation du nombre de cœurs, les architectures exigeront des dizaines de VCs, ce qui impliquera un impact en surface et en consommation d'énergie. Donc, cette approche influe sur le passage à l'échelle de l'architecture.

3.4.4.2 Partitionnement des ressources

Une deuxième technique de virtualisation est le partitionnement du réseau, ou chaque partition nécessite une configuration afin d'assurer l'isolation du trafic de différentes applications. Ce mécanisme permet une meilleure gestion de ressources et une assignation efficace des applications. Dans [TSAF11], une évaluation des performances est mise en œuvre lors de l'utilisation d'un réseau virtuel capable d'isoler le trafic des applications sous des scénarios statiques. En outre, un partitionnement intelligent fournit au réseau la capacité d'exclure les composants défaillants du système. Éventuellement,

les cœurs et les composants du réseau à l'état "idle", peuvent être mis en veille afin de réduire la consommation d'énergie. Pour supporter ce modèle de virtualisation, une modification architecturale est nécessaire : l'algorithme de routage.

Algorithmes de routages

Dans les architectures multicœurs, l'algorithme de routage le plus simple et le plus couramment utilisé dans le NoC est l'algorithme de routage "XY". Il s'agit d'un algorithme ordonné par les dimensions (*dimension ordered* ou *DOR*), qui, partant de la source prend la direction "X" jusqu'à la verticale du nœud destination, puis ensuite prend la direction "Y". Un algorithme statique, déterministe et sans interblocage. Cependant, ce type est limité à des topologies de réseau régulières spécifiques (ex., grille 2D, tore 2D, hyper-cube...). Or, dans le but de maximiser l'utilisation des ressources, comme évoqué dans la section 3.4.3.1, des topologies irrégulières sont plus appropriées pour des réseaux spécifiques à une application. En conséquence, des algorithmes de routage spécifiques associés à ce type de topologies s'avèrent nécessaires. Dans [SHZG05], une méthode basée sur l'algorithme de routage pair-impair (Odd-Even) a été proposé pour ce type de topologie. Pour l'irrégularité, des restrictions ont été définies telles que les tours Est-Nord ou Est-Sud pour les nœuds des colonnes paires et les tours Nord-Ouest ou Sud-Ouest pour ceux des colonnes impaires.

Les techniques de routage traditionnelles utilisées dans les topologies irrégulières sont consistuées principalement du routage à la source et du routage basé sur les tables [FRD08a]. Dans le routage à la source, l'itinéraire du trajet depuis la source vers la destination est pré-calculé et fourni dans l'entête du paquet. Or la taille importante de l'entête du paquet consomme de la bande passante. Pour le routage basé sur une table, cette dernière est stockée au niveau de chaque routeur et contient le port de sortie associé à chaque destination. Son avantage est qu'il est indépendant de la topologie. Mais, avec l'augmentation de la taille du NoC la taille de cette table augmente et ceci affecte la quantité et la performance des ressources[MPF+09]. Dans [PKH06] un mécanisme qui compresse les tables de routage a été illustré. Bien qu'il aide à améliorer le passage à l'échelle de ces tables, il vise un algorithme de routage particulier qui s'appuie sur la communication entre les nœuds en assurant l'absence d'interblocage. De plus, ce mécanisme est restreint à un chemin de routage minimal, par conséquent, il ne permet pas le routage dans un réseau avec des liens défaillants.

Pour un meilleur passage à l'échelle et pour une architecture tolérante aux fautes, [MPF+09] propose une technique de routage basée sur les régions. Partant d'une topologie et d'un algorithme de routage, cette technique stocke, au niveau de chaque routeur, les destinations dans chaque région. Ceci permet d'éliminer les informations redondantes dans les tables de routage. Avec l'apparition de défauts sur les liens, cette technique associée à un algorithme de routage basé sur les segments [MFD+07] assure le fonctionnement sans dégradation des performances.

D'autres solutions plus récentes ont été proposées permettant le routage indépendant de la topologie dont la forme est semi-régulière comme illustré sur la figure 3.2.

LBDR [FRD08a], une technique flexible, est conçue pour tenir compte des fortes contraintes liées aux systèmes multicœurs relatifs à la latence, la consommation d'énergie et à la surface. Toutefois, ce mécanisme exige 12 registres par routeurs (3 par ports : un pour la connectivité et les deux qui restent pour le routage), et quelques portes logiques

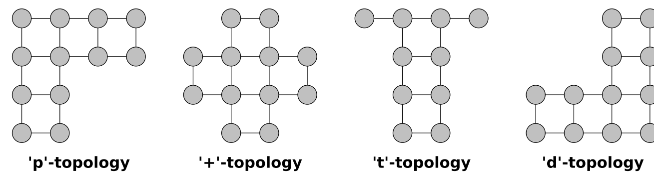


FIGURE 3.2 – Exemple de topologies semi-régulière (irrégulière) qui ne supporte pas un routage *DOR*

supplémentaires. En plus, avec la variation de la forme et la taille de la topologie, LBDR régénère à chaque fois les bits de routage ce qui rend le routage plus complexe. Afin de surmonter cette complexité un algorithme nommé FDOR (Flexible DOR) a été proposé [SSJR+09]. Il est adapté aux topologies de grille (2-D et 3-D) irrégulières aussi. Cette topologie est décomposée en au moins trois grilles 2D ou 3D : un cœur, un flanc X- et un flanc X+. Tous les nœuds du flanc X+ (resp. X-) ont des abscisses x supérieures (resp. inférieures) à celles du nœud du cœur (voir figure 3.3). Le coût de l'implémentation de cet algorithme est faible (un registre par routeur) donc il est de moindre complexité que le LBDR. Sa simplicité rend son utilisation intéressante dans le contexte des réseaux virtualisés. Un autre algorithme, CBDOR, destiné aux topologies irrégulières et qui ont la particularité d'être convexe, a été présenté dans [SZL+10]. En fait, une topologie est convexe si aucune cavité n'existe en son sein. Cet algorithme est très simple et efficace, il nécessite deux registres par routeur contenant l'information de connectivité. Alors, l'acheminement des paquets est basé sur ces registres et il ne demande aucune opération supplémentaire. Le mécanisme de routage CBDOR est plus simple que celui du LBDR ce qui le rend plus pratique et passant mieux à l'échelle. Malgré les avantages qui ont été cités, ce mécanisme souffre du fait que les ressources ne peuvent pas être partagées entre les applications puisque chaque région est assignée à une application spécifique.

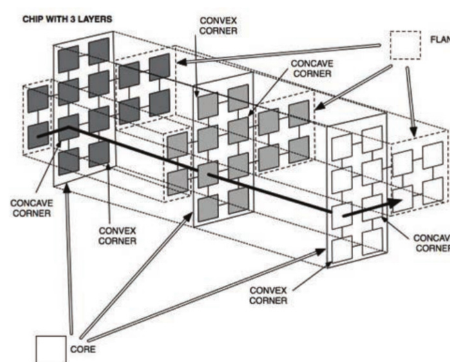


FIGURE 3.3 – 2x4x3 core de topologie grille avec un routage ZYX, et 2x1x2 flanc X-, et 2x2x3 flanc X+. La flèche décrit le chemin du paquet du nœud situé dans le flanc X- jusqu'au nœud dans le flanc X+ en traversant le core

L'algorithme de routage doit être doté d'une flexibilité pour la formation des régions de formes irrégulières. Néanmoins, les contraintes liées aux multicœurs limitent le choix

des stratégies de routage et d'allocation de ressources. Les tables de routages nécessitent un espace de stockage important (des ressources au niveau du routeur). Le routage à la source, le routage basé sur la région et les mécanismes permettant un routage indépendant de la topologie assurent suffisamment de flexibilité mais ils sont dotés d'un certain niveau de complexité.

Pour certains NoCs et sous certaines conditions, ces stratégies peuvent être inacceptables. En effet, le partitionnement de la puce en régions peut ne pas être efficace. Sa performance dépend en partie des ressources mémoires et de leur degré de partage. Elle est dictée par la communication et le protocole de cohérence des caches. Dans leur étude [LF13] Lodde et Flich ont cherché des solutions pour gérer efficacement cette communication critique lors de la virtualisation des ressources. Leur approche vise le partitionnement des ressources de la puce avec une utilisation intelligente de ses mémoires caches. C'est une combinaison de l'algorithme de routage LBDR avec une stratégie de mappage dynamique RHM [LFA13] pour, respectivement, un partitionnement niveau réseau et un autre niveau mémoires caches. Le mappage associe les blocs à un banc LLC, qui est proche du demandeur, au moment de l'exécution. Deux techniques orthogonales qui coopèrent pour offrir un support efficace à la virtualization.

3.5 Positionnement

Cette étude porte en priorité sur la virtualisation du réseau à cause de sa contribution importante à la performance des architectures multi-cœurs. La majorité des travaux s'appuie sur ce concept afin d'assurer l'isolation entre les applications ou entre les trafics d'une seule application. Or la concurrence entre tâches et le partage de ressources complexifie ce type de virtualisation.

Dans cette section, nous identifions certaines limites de la virtualisation qui l'empêchent actuellement d'être implantée dans les réseaux sur puces. Elles sont notamment liées à un manque de garanties de performance à l'intérieur du réseau virtuel partageant une infrastructure physique. Ces limites se résument comme suit :

- la plupart des travaux de virtualisation partitionnent le réseau en des régions indépendantes. Néanmoins, vu que les ressources peuvent avoir des degrés de partage élevés, le chevauchement des régions est une solution pour certaines situations ;
- la configuration des réseaux virtuels est limitée à un ensemble de ressources adjacentes. Cela est dû aux techniques d'allocation utilisées pour attribuer les ressources à une application. Pourtant, pour certaines applications, un réseau virtuel qui assure la communication entre des ressources non contiguës de la puce est nécessaire pour fournir des garanties de performance à ces applications ;
- l'aspect virtualisation des réseaux sur puce repose sur un seul type de topologie (sous-grille, sous cubes...). Cependant, pour une qualité de service des réseaux virtuels, plusieurs topologies peuvent cohabiter. Chacune est associée à un réseau virtuel pour répondre aux besoins. En outre, chaque réseau peut adopter son propre algorithme de routage.

Ces limites nous ont motivés à *i) virtualiser les ressources de la puce en tolérant le chevauchement des réseaux en cas de besoins*, *ii) permettre la configuration des réseaux reliant des ressources distantes*, et *iii) paramétrer la fonction de routage ainsi que la topologie des*

réseaux virtuels en fonction des besoins de l'application en ressources.

En particulier, nous proposons une abstraction de l'algorithme de routage qui assure la virtualisation des réseaux : des réseaux se superposent pour une meilleure utilisation des nœuds partagés. Cette abstraction permet de configurer la topologie de chaque réseau.

Dans ce manuscrit, nous proposons une virtualisation des ressources associées à des nœuds distants en tolérant la superposition de certaines réseaux. Les réseaux virtuels peuvent avoir des topologies différentes et des algorithmes de routage dédiés. Ceci permet de créer des réseaux applicatifs sur une plate-forme d'usage général.

3.6 Conclusion

La virtualisation est une technique attrayante qui est largement utilisée aujourd'hui dans des domaines variés. Elle a fait l'objet de nombreux travaux et ses contributions rendent son utilisation dans les architectures multiprocesseurs intéressante.

Traditionnellement, les réseaux ont été virtualisés pour offrir une connectivité indépendante à différentes applications. La virtualisation touche les ressources et leur fonctionnalités. Cela a engagé des recherches principalement sur la configuration des canaux virtuels et la façon de partitionner les ressources, pour s'adapter aux exigences des applications. Or, la virtualisation n'est pas toujours performante. En effet, le degré de partage des ressources (processeurs et mémoires) pour certaines applications est élevée. Pour fournir à chaque réseau virtuel des garanties de fonctionnalité et de performance complètes, il est crucial de déterminer les éléments affectant ceux-ci tels que la topologie et l'algorithme de routage.

La virtualisation du réseau a été également exploitée dans le contexte de la hiérarchie mémoire. Une organisation de la hiérarchie qui s'appuie sur les techniques de virtualisation, telles le regroupement et la gestion des mémoires caches par niveau, sont basés sur des réseaux virtuels ayant des topologies spécifiques. Il est probable que ces réseaux virtuels qui se superposent assurent une meilleure utilisation des ressources.

Nous présentons dans les chapitres suivants l'approche de virtualisation du réseau qui vise à combler les limites des techniques de virtualisation discutées dans ce chapitre. Diverses applications peuvent tirer profit de notre approche. Nous nous sommes intéressés à la hiérarchie mémoire et à l'exploitation de ce concept pour sa gestion.

Chapitre 4: Protocole de cohérence pour une nouvelle organisation hiérarchique des mémoires caches

Sommaire

3.1 Hiérarchie des mémoires caches 16
3.1.1 Généralités 16
3.2 Architecture NUCA 19
3.2.1 Réplication 19
3.2.2 Migration 21
3.2.3 Cache Coopératif 21
3.2.4 Discussion : limitations de la gestion du contenu des caches . . 23
3.3 Autour des NoCs 24
3.3.1 Co-conception mémoire et NoC 25
3.3.2 Cohérence de caches et Réseau 26
3.3.3 Discussion 26
3.4 Virtualisation 27
3.4.1 Généralité 27
3.4.2 Domaines de virtualisation 27
3.4.3 Virtualisation du réseau sur puce 28
3.4.4 Stratégies de virtualisation 30
3.5 Positionnement 33
3.6 Conclusion 34

Le chapitre précédant a mis en lumière les contraintes d'accès aux données et l'intérêt d'exploiter la hiérarchie des mémoires locales. Pour tirer parti des hautes performances fournies par les nœuds de calcul, les principes de localités doivent être respectés. Dans un programme à base de tâches parallèles, le degré de partage des données est important. Ainsi, le placement des données est déterminant. Il convient de porter un intérêt particulier au placement dans les bancs de cache pour réduire les effets non désirés des accès non uniformes (NUCA). Cependant, la capacité limitée des bancs de cache et le phénomène de congestion qui peut avoir lieu nous a mené à remettre en question l'organisation de la hiérarchie des caches, et notamment l'organisation des données dans ces mémoires caches.

Vis-à-vis d'un nœud de calcul, le placement optimal d'une donnée est celui qui permet de minimiser le nombre moyen de cycles d'horloge par instruction (CPI). Cette

métrique dépend de la pénalité d'un échec de cache. Elle varie également à cause des interactions et d'autres effets de bords (dépendances des données, congestion sur le réseau, etc...). C'est sur cet axe que la structuration de la hiérarchie a été élaborée.

En outre, les données étant manipulées par les divers cœurs, leurs modifications doivent être tracées et répercutées pour maintenir un état cohérent. Donc, un mécanisme de cohérence des caches doit être mis en œuvre. Il s'agit d'un protocole adapté aux spécifications de la hiérarchie.

Ce chapitre présente une nouvelle structure de la hiérarchie mémoire permettant d'exploiter efficacement l'espace de mémorisation embarqué sur la puce. Ce dernier est relativement limité, ce qui pousse à réfléchir à sa bonne exploitation. Cette structuration vise une optimisation d'accès aux données. Néanmoins, l'aspect partagé des données exige le maintien de la cohérence. Il découle de ce fait le besoin d'un protocole de cohérence de caches.

4.1 Structure de la hiérarchie des mémoires caches

Dans les mémoires caches de type NUCA, le placement des données est induit par les sous blocs des mémoires physiquement contigus. La hiérarchie est à la base d'un espace d'adressage contigu, divisé entre les caches L2 et pour laquelle chaque cache L1 peut accéder au cache L2 approprié selon les bits de poids faible d'une adresse (L2 cible).

Néanmoins, ce type de distribution de données peut induire de la contention au niveau des bancs ayant des blocs caractérisés par des degrés de partage élevés. La contention au niveau d'un banc de cache se produit lorsque le contenu de ce banc est un ensemble de blocs de mémoire contigus et que la plupart des données consultées sont placées dans le même banc. De ce fait, la latence pourrait croître de façon exponentielle.

Le principe de notre architecture consiste à avoir des caches contenant des adresses non nécessairement contiguës. En fait, le remplissage a lieu au fur et à mesure et selon le besoin d'un processeur. Ainsi donc, la notion de cible est définie selon le critère de localité spatiale du banc de cache d'un niveau par rapport au cache du niveau supérieur. En effet, un banc de cache L2 peut cacher n'importe quelle adresse demandée par un ou plusieurs caches L1. Chacun de ces derniers ne peut accéder qu'à un seul cache L2, sa cible. Ainsi chaque L1 dispose d'un L2 cible et un L2 forme avec un ou plusieurs L1 un *cluster* (figure 4.1). Idem pour les caches L3 : ils contiennent aussi des données venant d'adresses dispersées de la mémoire et chaque cache L2 se voit associé à un cache L3 cible. Ainsi, un L3 avec un ou deux caches L2 forment un cluster.

L'intérêt indéniable d'une telle organisation est que la distribution des données permet de réduire le nombre d'accès distants, d'équilibrer l'utilisation des différents bancs de cache du même niveau et d'assurer une meilleure exploitation de l'espace de mémorisation dédié à un niveau dans la hiérarchie. Le choix du cache cible est fixé au moment de la configuration en prenant en compte le nombre et l'emplacement des bancs de cache dans un niveau par rapport aux caches dans un autre niveau. Sans perte de généralité mais pour fixer les idées, nous faisons l'hypothèse que l'architecture matérielle sous-jacente dispose de 16 cœurs, 4 bancs de cache L2, 2 bancs de cache L3 et d'une mémoire principale. La figure 4.2 représente la configuration des clusters, les numéros des feuilles de l'arbre correspondant au numéro du cœur dans la grille 2D tels

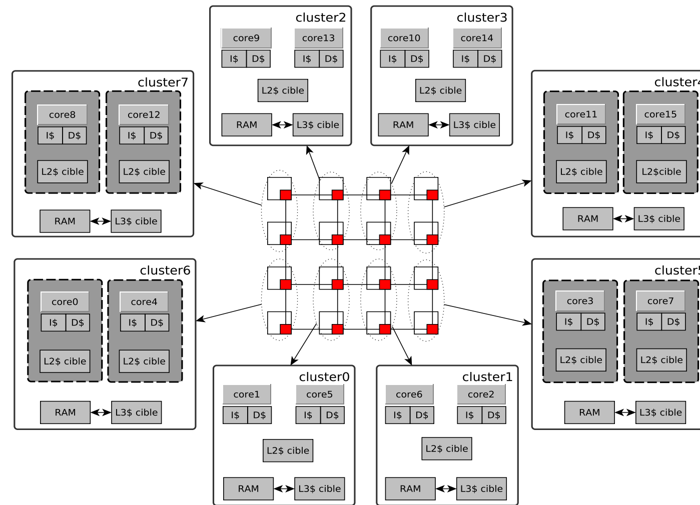


FIGURE 4.1 – Structure de la hiérarchie des mémoires locales

qu'indiqué figure 4.1.

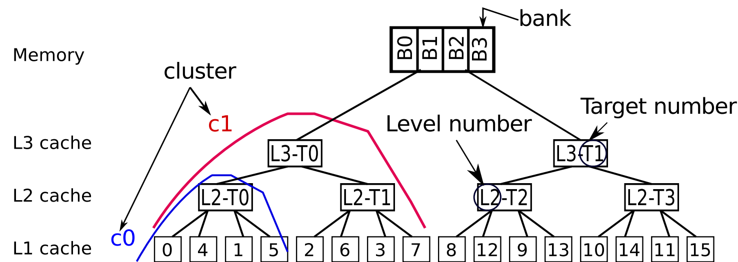


FIGURE 4.2 – Configuration logique des clusters

Étant donné que la cible peut être associée à un ensemble de cœurs, un banc peut être partagé par plusieurs cœurs. Alors, dans le cas où un ensemble de cœurs disposent d'un degré élevé de données partagées, il est préférable d'assigner la même cible à ces cœurs. En conséquence, une mauvaise association impliquera inévitablement des accès distants. Cette configuration est figée et elle n'est pas modifiable par l'OS. Nous faisons l'impasse du choix topologique des cibles dans cette étude.

4.2 Métriques d'évaluation des performances

Vis-à-vis de la hiérarchie mémoire dans un système multiprocesseur, l'évaluation de la performance est fortement liée à la manière dont elle est gérée, et elle se traduit par la valeur de différentes métriques. La plus importante est le *temps moyen d'accès à la mémoire* (AMAT) définie par l'équation 4.1.

$$AMAT = (1 - MissRate_L1) \times HitTime_L1 + MissRate_L1 \times AMT_L1 \tag{4.1}$$

Comme les caches L1 disposent d'un temps d'accès fixe, l'amélioration du *temps d'échec moyen* (AMT_L1), défini par l'équation 4.2, est la clé de la performance du système.

$$AMT_L1 = AAL_L2 + MissRate_L2 \times MissPenalty_L2 \tag{4.2}$$

Le *temps d'échec moyen* est le temps pendant lequel le processeur aura été bloqué à cause d'un échec d'accès aux données dans les caches de premier niveau. Il dépend principalement de deux facteurs :

– le *temps d'accès moyen* au cache L2 noté AAL_L2.
– le *taux d'échec* d'accès à un banc de cache L2($MissRate_L2$).

Notons que le facteur de la *pénalité d'échec* ($MissPenalty_L2$) dans l'équation 4.2 est déterminé par l'architecture matérielle : si la hiérarchie des mémoires caches inclut un troisième niveau, ce facteur est l'approximation de $l'AAL_L3$ et du taux d'échec d'accès aux caches de niveau 3. Donc, la généralisation de l'équation de l'AMAT à une architecture ayant 3 niveaux de cache a la forme suivante :

$$AMAT = (1 - MissRate_L1) \times HitTime_L1 + MissRate_L1 \\ \times [AAL_L2 + MissRate_L2 \times (AAL_L3 + MissRateL_L3 \times MissPenalty_L3)] \quad (4.3)$$

L'ajout de niveaux de cache dans la hiérarchie n'a d'influence que sur la pénalité d'échec qu'il convient de réduire au minimum. En fait, ces ajouts permettent de masquer au mieux la latence d'accès à la mémoire principale, qui est assez importante. Cette latence n'est autre que le facteur de la pénalité d'échec aux caches de niveau 3 ($MissPenalty_L3$) dans l'équation 4.3.

Pour mieux assimiler les choses, nous nous appuyons sur l'équation 4.2 dans la suite de cette partie et nous analysons ses facteurs de performance. L'objectif principal de notre travail est donc de réduire le taux d'échec du cache L2 ($MissRate_L2$). En effet, comparé aux organisations classiques, la notre permet de souligner que nous visons à augmenter la probabilité de satisfaire le besoin du demandeur en ne stockant qu'une seule copie par niveau. Cela est possible grâce à la particularité de notre architecture à répartition partagée où chaque niveau contenant un ensemble de bancs semble en pratique ne former qu'un seul cache important, sans aucune restriction pour accéder aux données.

Cependant, une telle réduction ne correspond pas nécessairement à une amélioration de la performance globale du système. Cela peut entraîner une amélioration réelle si la configuration de la mémoire cache réussit à compenser la latence d'accès moyenne (AAL_L2) par la latence économisée grâce à la réduction du taux d'échec ($MissRate_L2$). Il convient donc de prendre en compte le nombre d'accès entre le cache L1 (demandeur) i et le banc de cache L2 j, et la latence moyenne L_{ij} en nombre de cycles des accès sur le lien (i, j), particulièrement si les accès sont distants. En fait, la latence d'accès moyenne au cache L2 (AAL_L2) est formée de deux composantes. La première correspond à la latence d'accès au banc L2 cible et la deuxième est la latence d'accès au banc différent de la cible que nous notons "$\overline{cible}$".

$$AAL_L2 = AAL_L2cible + AAL_L2\overline{cible} \quad (4.4)$$

Ainsi $AAL_L2\overline{cible}$ est le facteur qui doit être compensé. Il est approximativement constant tant qu'il n'existe pas de congestion au niveau des bancs de cache L2 ou sur les liens de communications. Or cette hypothèse est loin d'être réelle. Lors de la recherche de la donnée dans des bancs distants, la congestion au niveau des liens de communication aura lieu. L'aspect communication doit donc être étudié convenablement.

4.3 Gestion de mémoires caches

4.3.1 Placement des données

Les performances de la hiérarchie des mémoires locales sont conditionnées par la répartition des données que ce soit à la phase initiale ou bien au cours de l'exécution. Le placement doit être attentivement étudié puisqu'il peut entraîner de sévères dégradations. En se basant sur la structure de la hiérarchie des caches et la manière dont les requêtes ont été servies, nous mettons l'accent sur la façon d'attribuer les données aux différents bancs de cache. L'attribution est effectuée au moment d'un échec et les deux cas de figure qui se présentent sont les suivantes :

- Échec dans tous les bancs d'un même niveau : Si la donnée n'existe dans aucun banc d'un niveau donné (ex. niveau 2), nous supposons que le cœur qui sollicite la donnée est celui qui en a vraiment besoin. Ainsi, la donnée est placée dans le banc cible associé à ce cœur lors de la configuration.
- Échec dans le banc cible associé au demandeur : Si la donnée réside dans un banc de cache distant (qui n'est pas la cible du demandeur), nous considérons que le dernier cœur émetteur de la requête a moins besoin de la donnée que les cœurs associés à ce banc. Par conséquent, la donnée est renvoyée au demandeur sans être migrée sur la cible du demandeur.

4.3.2 Mécanisme de recherche de la donnée

Lors d'un échec au niveau du banc cible, un mécanisme de recherche dans les bancs du même niveau se déclenche. C'est une sorte de diffusion des requêtes ayant la mission d'interroger les autres bancs. À la réception d'une requête, l'étiquette des lignes de l'ensemble est comparé à celui de l'adresse accédée. En cas de succès, la donnée est renvoyée au demandeur. En cas d'échec un accusé de réception (Ack) est envoyé. Si la cible reçoit des Acks de la part de tous les bancs de son niveau, cela signifie que la donnée n'existe plus dans ce niveau de la hiérarchie et la requête est transférée à la cible dans le niveau suivant.

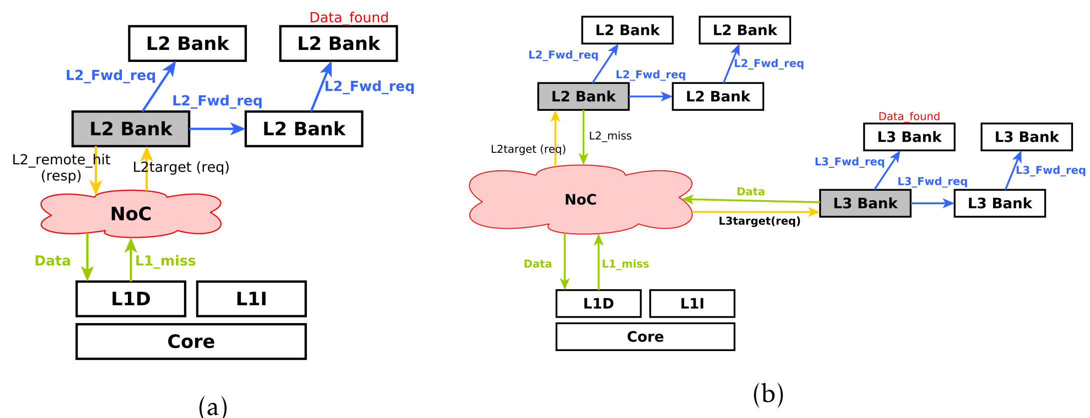


Figure 4.3 – Aperçu du mécanisme de recherche de donnée haut niveau : (a) donnée dans un banc L2 différent de la cible (b) donnée dans un banc L3 différent de la cible

La figure 4.3 décrit le processus de la recherche d'une donnée absente du banc L2 cible. La diffusion de la requête à tous les bancs du même niveau est déclenchée. Dans

la figure 4.3a la donnée est trouvée dans un banc du même niveau. Elle sera redirigée à la cible qui la transfère au cache L1. Ce transfert aura lieu après la réception des acquittements des autres bancs qui n'ont pas été modélisés dans le schéma. Pour la figure 4.3b la donnée n'existe pas dans le niveau 2, ainsi, suite à la réception de tous les accusés de réception, le banc L2 cible transmet la requête à sa cible L3. N'ayant pas la donnée cette dernière déclenche la diffusion. La donnée, une fois trouvée, est renvoyée à la cible L3, qui l'envoie à la cible L2 puis cette dernière la transmet au cache L1.

4.3.3 Problème lié à la structure de la hiérarchie

Dans notre système nous n'avons aucune garantie sur les temps d'accès aux bancs de cache et à la ligne sollicitée. Par conséquent, un problème peut avoir lieu lorsque deux cœurs ou plus (n'ayant pas la même cible) veulent accéder simultanément à une même donnée et que cette dernière n'existe dans aucune des cibles. En effet, chaque cœur consulte son banc L2 cible et chaque L2 déclenche la diffusion mais la ligne est mise dans un état d'attente d'Ack. Ainsi, si une requête demande la ligne qui est en attente, elle ne sera pas servie. Dans ce cas, plusieurs bancs s'attendent les uns des autres, ce qui entraîne un phénomène d'interblocage (en anglais *deadlock*).

Pour éviter ce phénomène, l'idée principale est que les requêtes doivent être sérialisés. Nous proposons une solution qui n'est pas optimale mais qui permet de résoudre le problème. Dans la situation illustrée précédemment les deux requêtes seront transmises aux bancs du niveau suivant. D'ailleurs, lorsque la diffusion est déclenchée un drapeau (en anglais *flag*) associé au bloc (au niveau de la cible) est activé. À la réception d'une requête de diffusion de la part d'une autre cible, le drapeau est testé. S'il est actif l'Ack est envoyé même si la ligne est dans l'état d'attente. Par conséquent, les cibles peuvent relayer leurs requêtes au niveau suivant. Si les requêtes consultent le même banc dans le niveau suivant, ce dernier représentera le point de sérialisation au niveau duquel la première requête sera servie et la seconde sera redirigée à la cible L2 (dont la requête a été servie). La requête redirigée sera bloquée jusqu'à ce que la cible reçoive la réponse du niveau suivant. En revanche, si chaque requête est envoyée à un banc de cache L3, le même processus du second niveau aura lieu et le point de sérialisation sera alors le niveau qui suit.

4.4 Protocole de cohérence des caches

4.4.1 Principe

La gestion efficace de la mémorisation des données au niveau de la hiérarchie pour une meilleure utilisation de l'espace mémoire disponible sur la puce nécessite l'adaptation du mécanisme de cohérence. Ce mécanisme est indispensable dès lors que l'on souhaite supporter des applications parallèles multi-tâches classiques.

Afin de maintenir la cohérence des données dans nos mémoires cache, l'approche proposée est inspirée des protocoles de "snoop" présentés dans le chapitre précédent. Elle est à mi-chemin entre l'approche "source snoop" qui diffuse la requête à tous les caches du système et l'approche "home snoop" qui se base sur un répertoire associé au banc mémoire contenant la ligne de cache recherchée et indiquant l'état de partage de la ligne et le banc contenant la donnée. En effet, notre approche consiste à faire la diffusion

si le cache cible consulté ne contient pas la donnée. La cible dans notre cas est un banc de cache L2, et non pas un banc de mémoire. La différence réside dans le fait que le banc mémoire contient sûrement la donnée recherchée alors que le banc de cache L2 peut ne pas contenir la donnée à l'instant de l'envoi de la requête. Nous avons donc adopté le mécanisme de diffusion dans le reste des bancs de cache L2. Ainsi, l'absence de la donnée au niveau de la cible induit la réplication de la requête et l'envoi de ces répliques aux autres bancs de cache L2. Une telle diffusion est basée sur un algorithme particulier détaillé ultérieurement.

De ce fait, le mécanisme de recherche de la donnée fait partie intégrante du protocole de cohérence de caches. Ce protocole est une adaptation du protocole MESI existant qui repose sur le fait que les bancs L2 semblent former un seul cache L2 et que l'accès à ces derniers n'est pas restreint. Cela signifie qu'une donnée peut exister non pas dans le banc cible de l'initiateur de la requête, mais dans un autre banc du même niveau.

Par ailleurs, la donnée qui réside dans un banc différent de la cible associée à l'initiateur de la requête n'a pas la possibilité de migrer en dehors du banc sur lequel elle a été placée, et son évincement du cache privé nécessite de la mettre à jour (son état de partage, la ligne elle-même...) dans le banc L2.

Dans le protocole, les requêtes de lecture d'une donnée (GETS) ou d'une instruction (GET_INST), et d'écriture (GETX) envoyées du premier niveau(L1) vers le second(L2), sont les déclencheurs de la diffusion si la ligne n'est pas hébergée par la cible. Les modifications majeures sont reliées aux caches de niveau 2 ou 3 s'il existe. Nous aurons donc besoin de plus d'états transitoires pour gérer les scénarios susceptibles de se produire.

4.4.2 Implantation du protocole

4.4.2.1 États du premier niveau du protocole

Ce niveau contient deux mémoires caches, un cache de données et un autre d'instructions mais les deux partagent le même contrôleur. Ce dernier est connecté directement aux processeurs. Or la connexion entre les contrôleurs L1 est assurée via les requêtes issues du contrôleur L2.

États stables

Un des quatre états modifié (**M**), exclusif (**E**), partagé (**S**), invalidé (**I**) est attribué à chaque ligne contenue dans les caches L1. Le premier (**M**) correspond à une ligne qui a été écrite localement. Le second (**E**) traduit l'unicité de la ligne, elle existe dans un seul cache. Dans ces deux états l'écriture et la lecture sont gérées localement. L'état (**S**) est attribué aux lignes cohérentes dans plusieurs bancs de cache à la fois. Les lectures sont traitées localement, tandis qu'une écriture génère une demande d'exclusivité pour invalider les autres copies et récupérer la copie à jour si c'est nécessaire. Dans l'état (**I**) la copie est périmée, elle doit être réactualisé avant que le cœur de calcul puisse en exploiter les données. Généralement, un état supplémentaire (**NP**) est l'état initial, mais une fois la simulation commencée, cet état n'est plus atteint, d'où la décision d'utiliser uniquement l'état (**I**).

États transitoires

La nécessité de ce type d'état est liée au fait que la plupart des opérations qui se produisent ne sont pas atomiques et que les accès concurrents doivent être résolus. La programmation des applications multi-tâches exige un mécanisme de blocage, mais il n'existe pas une manière explicite de bloquer le contrôleur. De plus des contraintes d'ordre ne sont plus imposées et souvent les messages reçus ne sont plus traités dans l'ordre de leurs émission/réception. La solution pour la création du mécanisme de blocage est de définir des états transitoires. Certes, ces états transitoires ne sont pas tous bloquants et pour certains états, le contrôleur peut continuer l'envoi des réponses.

À l'issue d'une requête d'écriture (resp. lecture) d'une donnée, l'état attribué est **IM** (resp. **IS**) jusqu'à la réception de la réponse. Dans le cas où la ligne est dans l'état partagé (**S**) et qu'une requête d'écriture est initiée, la ligne transite à **SM** ce qui assure l'invalidation des copies et l'obtention de la permission d'exclusivité. Un autre état **IS_I** est atteint à la réception d'une invalidation de la part d'un banc L2. En fait, pendant que le cœur considéré essaie de lire cette ligne (état **IS**), des accès concurrents dus à des écritures au niveau de la même ligne se produisent. Également, lors de l'évincement d'une ligne modifié (**M**) qui nécessite une action de mise à jour au niveau du cache L2 ("write-back"), la ligne passe dans l'état **M_I** indiquant qu'elle attend l'acquittement. Or, lors de cette attente la ligne peut recevoir une intervention (une requête transmise d'autres cœurs) et la situation de concurrence aura lieu. Pour le résoudre et afin d'éviter le problème des write-backs abandonnées la ligne passe dans l'état **SINK_WB_ACK**.

4.4.2.2 États de second niveau du protocole

États stables

À ce niveau les états stables cherchent à traduire les propriétés suivantes : si le bloc dans le cache est partagé (*clean*) ou modifié (*dirty*), et si ce banc est le seul qui contient le bloc. Ainsi, les états basiques sont :

NP : La ligne n'est pas en cache.

SS : La ligne réside dans plusieurs caches privés (i.e. est dans l'état *S* dans les caches privés). Un vecteur associé à la ligne identifie ces derniers. Cette ligne est en lecture seule.

M : La ligne dans l'état modifié existe uniquement dans le cache L2 ayant la permission d'exclusivité. Les requêtes de lecture/écriture sont satisfaites par le cache L2.

MT : La ligne n'est pas à jour dans le cache L2 mais elle existe dans un seul cache L1. L'identité de ce dernier est sauvegardée dans un champ associé à la ligne. Le cache L2 ne peut pas servir les requêtes de lecture/écriture issues des autres caches L1. Ces requêtes seront transférées au cache L1 propriétaire de la ligne.

États liés à l'évincement des lignes

L'évincement d'un bloc peut être induit par le contrôleur L1, ou par une requête émise par un autre nœud. Dans le cas où le bloc n'est pas partagé, le contrôleur transite dans l'état ***NP***. Inversement, on attend jusqu'à ce que tous les niveaux soient mis à jour.

M_I : Cet état est atteint lors d'un évincement d'une ligne dans l'état ***M***. Cette ligne reste ainsi dans l'attente d'un acquittement suite à une mise à jour "PUTX" au niveau du répertoire associé.

MT_I : L'évincement d'une ligne dans l'état ***MT*** nécessite l'invalidation de la ligne dans le cache L1 correspondant (*owner*). Ce dernier se charge d'envoyer la ligne à jour à ce cache avant de l'invalider et durant ce temps cette ligne est dans l'état ***MT_I***.

MCT_I : Similaire à l'état précèdent, sauf que la ligne est à jour "clean" dans ce banc.

I_I : Lors de l'évincement d'une ligne dans l'état ***SS***, les invalidations sont envoyées aux caches L1 partageant la ligne et la ligne passe dans l'état ***I_I***, attendant la réception des acquittements de la part des caches L1.

S_I : Similaire à l'état ***I_I*** sauf que la donnée ici est "dirty" ce qui nécessite sa mise à jour au niveau de la mémoire.

États liés aux requêtes mémoire

ISS : À la réception d'une requête de lecture d'un bloc qui n'existe pas dans ce banc, cette requête est transférée aux autres bancs du même niveau. L'état est obtenu uniquement si la requête demande une donnée et non pas une instruction. De plus, une ligne dans cet état indique qu'un seul cache L1 a demandé la ligne et cette dernière sera transférée au cache L1 avec l'état exclusif.

ISS_B : Cet état est obtenu lorsque la ligne reçoit un acquittement d'un banc du même niveau, indiquant que la requête doit être transférée à la mémoire. C'est le dernier acquittement reçu (i.e. la donnée n'existe plus dans aucun banc de cache de ce niveau).

IS : Similaire à l'état ***ISS***, sauf que cette fois la requête correspond à une instruction ou bien plus qu'un processeur demande la donnée en lecture à l'instant de l'attente d'une réponse. Une fois cette réponse reçue, le bloc est envoyé au(x) demandeur(s).

IS_B : Identique à l'état ***ISS_B***, cet état est atteint lors de la réception de tous les acquittements de la part des bancs du même niveau. Ainsi la requête est transmise à la mémoire. Une autre situation où la ligne peut avoir cet état est due à la réception d'un acquittement particulier. Ce type d'acquittement est envoyé en interrogeant un banc, cible sollicitée par un autre processeur ayant la ligne dans l'état ***ISS***.

IM : Une requête d'écriture est reçue demandant une ligne qui n'existe pas dans ce cache. La ligne transite vers cet état et la requête est transférée aux autres bancs du même niveau pour demander la donnée si elle existe.

IM_B : L'état est atteint lors de la réception d'un dernier acquittement des bancs de cache consultés traduisant que la donnée n'existe dans aucun ou qu'elle existe mais dans un état ***IM***. La différence par rapport à l'état précédant ***IM*** est que dans cet état la requête est transférée à la mémoire et la ligne est bloquée.

États bloquants

Le contrôleur du cache L2 transite vers l'un de ces états lors de la réception d'une requête de l'un des caches L1 et que la ligne requise a des copies dans d'autres caches L1. Dans ce cas, ce contrôleur agit comme transitaire, et envoie la donnée avec le nombre de L1s partageant la ligne au demandeur (L1 cache) de cette ligne. Ce dernier doit ainsi

débloquer le contrôleur L2 une fois qu'il a reçu les réponses des caches L1 qui partagent la ligne. Le contrôleur à ce niveau est dans l'attente d'une réponse des caches L1 privés et jusqu'à la réception d'une telle réponse, aucune autre demande ne sera traitée.

Si un cache L1 veut lire une ligne et non pas y écrire, le cache L2 peut envoyer la réponse immédiatement à condition que la donnée n'existe pas dans un autre cache L1 dans l'état exclusif. Autrement, dans telle situation, le L2 demande au propriétaire d'envoyer la donnée au demandeur (cache L1), et probablement si la donnée est dirty elle sera mise à jour au niveau du L2. En outre, lorsque l'un des caches L1 change de l'état ***E*** ou ***M*** à ***S***, un message write-back devra être envoyé car un cache L1 dans l'état ***S*** peut évincer la ligne et les modifications seraient sinon perdues. Nous énumérons dans la suite les états bloquants attribués à une ligne :

MT_MB : Le banc de cache L2 envoie la ligne avec une permission d'exclusivité au cache L1 demandeur (directement ou en passant par la cible), mais il n'a pas encore reçu l'action "unblock" de la part de ce dernier qui accuse la réception de la permission exclusive.

MT_MBB : Lors de la réception d'une requête d'écriture d'un banc de cache du même niveau et que la ligne demandée est dans l'état ***MT*** cette ligne passe dans l'état ***MT_MBB*** qui indique que la ligne est bloquée jusqu'à ce que le demandeur soit servi. Dans cet état tous les types de requêtes sont mis en attente jusqu'au le déblocage de la ligne.

SS_MB : Cet état est atteint chaque fois qu'un cache L1 demande un bloc avec une permission exclusive (GETX ou Upgrade) et l'état de cohérence de ce bloc est ***SS***. Cela signifie que le bloc sollicité à des copies dans plusieurs caches L1 privés. L'invalidation de toutes ces copies est nécessaire. Ainsi l'état ***SS_MB*** indique que les invalidations ont été envoyées et le demandeur est informé du nombre d'acquittements qu'il doit recevoir. À la réception de tous les acquittements, le cache L1 demandeur envoie une action "unblock" afin de débloquer la ligne et reprendre le traitement des requêtes demandant auparavant la ligne.

MT_IIB : Lors de la réception d'une requête de lecture pour une ligne résidente dans un cache L1 ayant l'état exclusif ***E***, le répertoire associé au cache L2 transfère la requête au cache L1 propriétaire et la ligne passe dans l'état ***MT_IIB***. Deux évènements doivent avoir lieu pour le déblocage de cette ligne : Le propriétaire (*owner*) envoie la ligne au cache L2 pour sa mise à jour avec la dernière valeur et le cache L1 demandeur initial envoie une action de déblocage "unblock" précisant qu'il a reçu la ligne

MT_IB : Il est atteint une fois que la ligne dans le cache L2 dans l'état ***MT_IIB*** reçoit l'action de déblocage "unblock" du L1 demandeur alors qu'il n'a pas encore reçu l'action de mise à jour (*write-back*) de la part du précédant propriétaire (*owner*).

MT_SB : Lorsque la ligne est dans l'état ***MT_IIB*** et elle reçoit l'action de mise à jour (*write-back*) avant l'action de déblocage, on atteint cet état.

MT_IBB : Cet état est similaire dans l'état ***MT_MBB*** sauf que celui-ci concerne les requêtes d'instructions ou les requêtes de lecture. Donc, une fois que la ligne dans l'état ***MT*** reçoit une requête d'un banc de cache de même niveau, elle aura l'état ***MT_IBB***

Ayant été conçu comme une évolution du protocole MESI [**?**], nous nous concentrons dans ce que suit sur la partie du protocole modifiée. En effet, la différence réside au niveau 2 de la hiérarchie des mémoires locales : lors d'un échec au niveau du banc de

cache cible, au lieu de transmettre la requête au niveau suivant (le cas pour MESI), nous consultons les autres bancs de cache du même niveau. De ce fait, les modifications apportées concernent les bancs de cache de niveau 2 et plus précisément les états d'une ligne de ce banc. Les états stables sont intacts alors qu'une modification des actions générées et un ajout d'états transitoires ont été nécessaire. Pour les caches de niveau 1 les mêmes états ont été conservés mais ils produisent, et donc subissent également, des actions supplémentaires.

À noter que tous les états présentés pour une ligne dans un banc de cache L2 restent valables pour les bancs de cache L3 et dans la suite tous les mécanismes concernant les caches L2 seront les mêmes pour les caches L3.

4.4.2.3 Requête L2_Fwd_GET(X/S/_INST)

Pour répondre à une requête d'écriture (GETX), de lecture (GETS) d'une donnée ou de lecture d'une ligne d'instruction (GET_INST), lors de la consultation de la cible (banc L2) et que la ligne n'y existe pas, un mécanisme de recherche dans les bancs de cache du même niveau est déclenché. La cible envoie une requête « L2_Fwd_GET(X/S/_INST) » aux différents bancs pour leur demander la donnée en affectant à la ligne l'état IM/IS/ISS. C'est une requête de multicast aux bancs du même niveau.
Lorsque l'un des bancs reçoit une requête d'écriture "**GETX**", si la ligne n'existe pas un accusé de réception est envoyé à la cible. Dans le cas inverse, c'est l'état de la ligne (SS/M/MT) qui dicte le processus (figure 4.4).

➤ À l'état **MT**, une requête est transférée au cache L1 propriétaire de la ligne et transite dans l'état MT_MBB indiquant que la ligne est bloquée au moment du broadcast. Ensuite, cette ligne attend la réception de la donnée pour passer dans l'état MT_MB : un deuxième état bloquant au cours duquel la donnée est relayée par le L2 cible au demandeur initial. Une fois la donnée du L1 reçue, le L2 transfère un accusé avec la donnée au L2 cible, qui ne la sauvegarde pas. Cette donnée est transmise à l'initiateur de la requête. Finalement, ce dernier envoie un message au banc de cache L2 contenant la donnée pour débloquer la ligne, qui passe de l'état MT_MB à l'état MT.

➤ À l'état **M**, la requête est servie par ce banc de cache L2 et la ligne passe dans l'état MT_MB. Il envoie l'accusé avec la donnée à la cible qui a initié la requête, à son tour cette cible transmet la donnée au L1 demandeur en indiquant l'identité du banc L2 qui contient la donnée. Le cache L1 envoie à ce dernier le message pour débloquer la ligne et son état passe à MT.

➤ Dans l'état **SS**, deux types d'actions se déclenchent : le premier consiste à envoyer des invalidations à tous les caches L1 ayant une copie de la ligne et le second type est un accusé envoyé à la cible accompagné de la donnée et du nombre de caches partageant cette ligne. L'état de cette ligne dans ce banc passe à SS_MB. Ensuite, la cible transmet la réponse et les informations reçues à l'initiateur de la requête. Ce dernier n'envoie le message de déblocage au banc contenant la donnée que lorsqu'il reçoit les accusés de réception de tous les messages d'invalidation. Suite à ceci la ligne passe de SS_MB à MT.
Lors de la réception d'une requête de lecture d'une donnée "**GETS**" ou d'une instruction "**GET_INST**" et que le banc cible sollicité ne contient pas cette donnée, les requêtes L2_Fwd_GETS ou L2_Fwd_GET_INST se déclenchent (figure 4.5).

➤ Dans l'état **MT** (figure 4.5 (a)), les deux types de requêtes seront servis de la même manière : La requête est transférée au cache L1 propriétaire et la ligne transite dans l'état

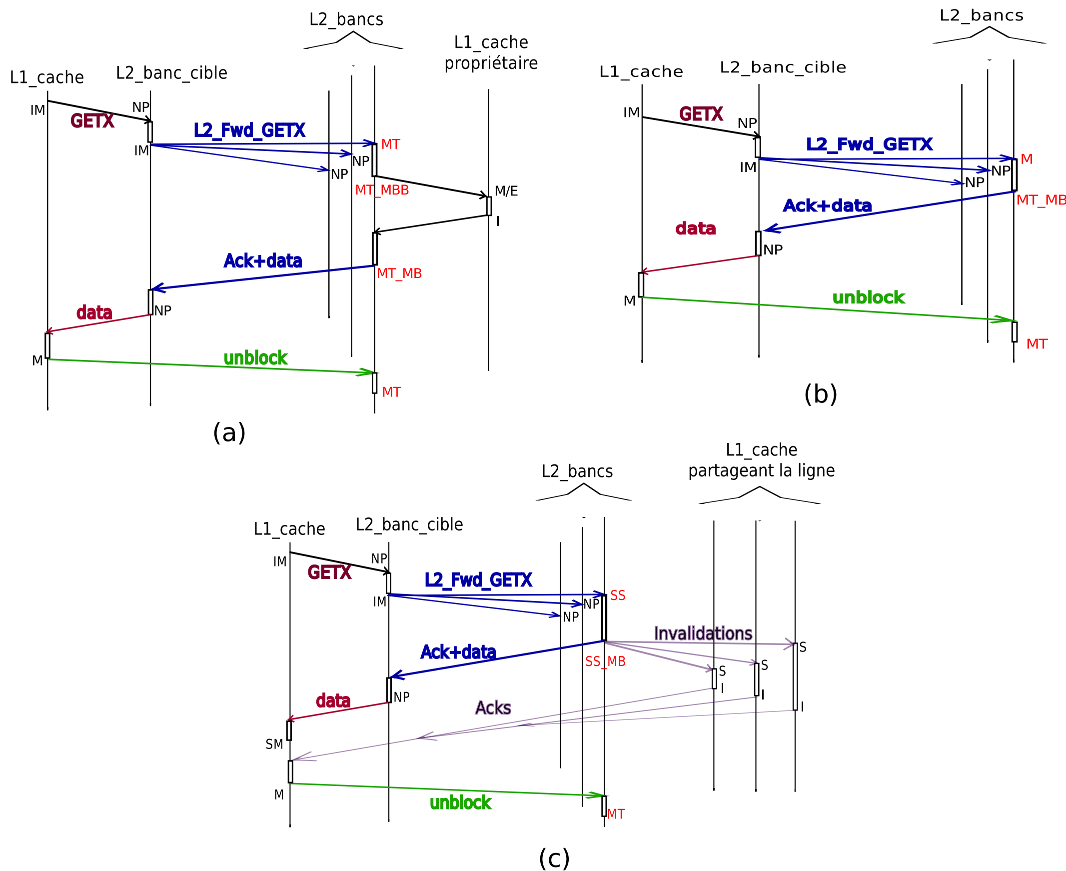


Figure 4.4 – Scénarios de l'écriture d'une donnée résidente dans un banc de cache L2 différent de la cible (a) État de la ligne MT (b) État de la ligne M (c) État de la ligne SS

MT_IBB. Ensuite, lors de la réception de la donnée, la ligne passe dans l'état MT_SB, l'état bloquant au cours duquel la donnée est relayée au demandeur initial. Une fois la donnée reçue, le contrôleur transfère un accusé avec la donnée à la cible, sans l'y sauvegarder, et la donnée est transmise à l'initiateur de la requête. Ce dernier envoie un message au banc de cache contenant la donnée pour débloquer la ligne qui passe de l'état MT_SB à l'état S.

➤ À l'état **M** et à la réception de la requête L2_Fwd_GETS (figure 4.5 (b)), la ligne passe dans l'état MT_MB après l'envoi de l'accusé de réception accompagné de la donnée à la cible L2. Cette dernière transmet la donnée au demandeur qui à son tour envoie le message de déblocage exclusif au banc L2 dans lequel réside la donnée. Ainsi l'état passe de MT_MB à MT. Pour la requête L2_Fwd_GET_INST (figure 4.5 (d)), la ligne consultée dans l'état M transite directement dans l'état SS et elle sera relayée à la cible qui se charge de la transférer aux demandeurs L1s.

➤ À l'état **SS**, le processus lors de la réception de la requête L2_Fwd_GETS (figure 4.5 (c)) diffère de celle de L2_Fwd_GET_INST (figure 4.5 (d)) : Suite à la première, la ligne dans le banc sollicité passe dans l'état SS_MB. Cet état bloquant est nécessaire pour la cohérence de la ligne qui peut recevoir une requête d'écriture à n'importe quel instant. Donc, l'état empêche l'accès à la donnée jusqu'à son déblocage de la part de son demandeur. Ensuite, dès qu'elle passe dans l'état SS, les requêtes en attente seront

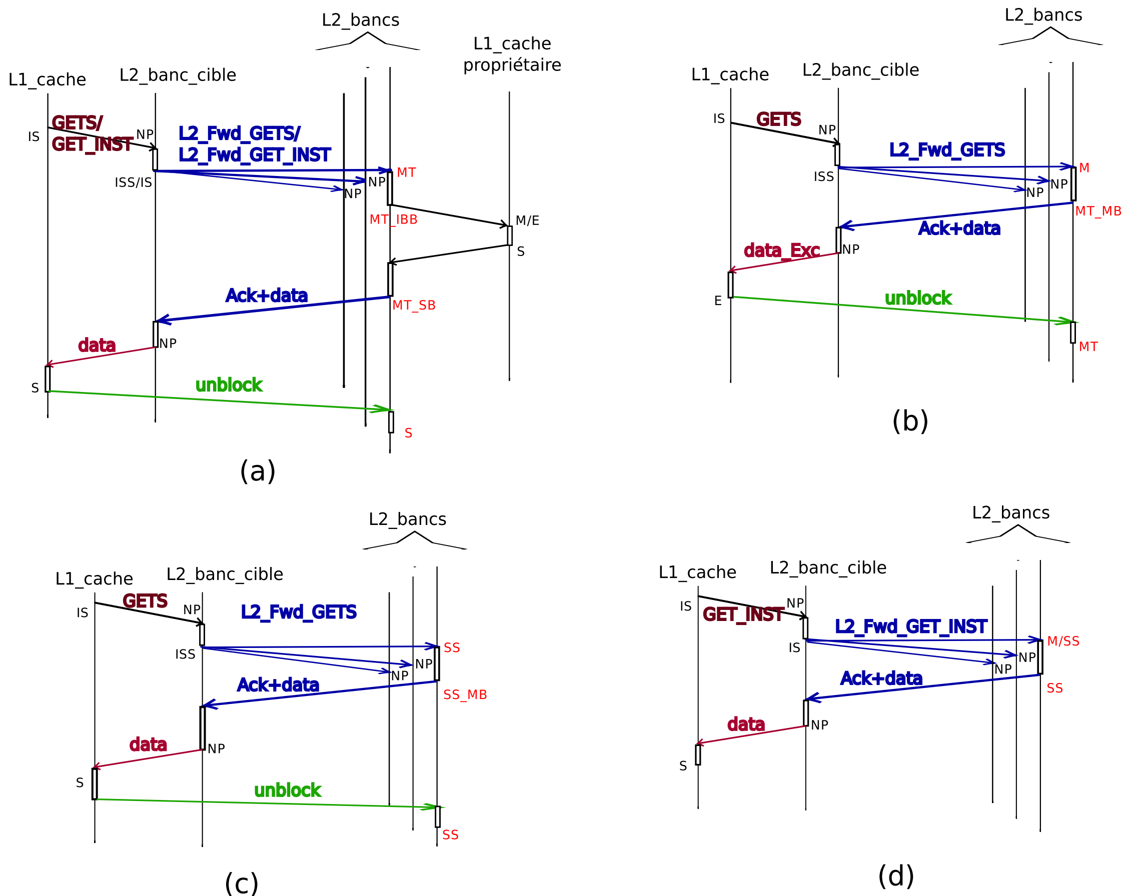


FIGURE 4.5 – Scénarios de la lecture d'une donnée et/ou d'une instruction résidente dans un banc de cache L2 différent de la cible (a) Lecture d'une donnée ou d'une instruction dans l'état MT (b) Lecture d'une donnée dans l'état M (c) Lecture d'une donnée dans l'état SS (d) Lecture d'une instruction dans l'état M ou SS

servies. Par contre, lors de la réception du deuxième type de requête la ligne reste dans l'état SS et l'instruction est transmise à la cible puis au demandeur.

Cache L2 cible

La stratégie du protocole consiste à consulter le banc de cache cible en premier lieu pour répondre à la demande du cache L1. Si le banc ne peut pas servir la demande, une ligne est allouée en lui affectant un état transitoire : IM (cas d'une écriture), ISS (cas d'une lecture d'une donnée pour la première fois) ou IS (cas d'une lecture d'une instruction ou d'une donnée déjà demandée en lecture). La ligne dans ce banc est dans l'un de ces états jusqu'à la réception des accusés de réception et/ou de la donnée de la part des bancs du même niveau. Pour la clarté des figures nous n'avons pas représenté les accusés de réception suite à la consultation d'un banc de cache qui ne contient pas la donnée. Nous nous contentons de tracer uniquement l'accusé accompagné de la donnée du banc contenant la donnée. Une fois reçue, la ligne est déchargée et elle passe de l'état IM/IS/ISS à l'état NP.

Il est important de s'appuyer sur les requêtes de lecture d'une ligne ou d'une instruction. En effet, dans le protocole MESI standard, ces deux types de requêtes sont servis même si la ligne dans le banc cible est dans l'état transitoire (ISS/IS). Un exemple simple

est que la ligne dans l'état ISS passe dans l'état IS lors de la réception d'une requête de lecture de la donnée. L'identité de l'initiateur est ajoutée à la liste des caches L1 partageant la ligne, au niveau de cette cible. Alors, dès que la donnée atteint la cible, elle sera envoyée à tous les caches L1 de la liste. Cependant, pour notre protocole le scénario critique qui peut avoir lieu est le suivant : Suite à la demande de lecture d'une ligne qui n'existe pas dans le banc cible, la requête répliquée est envoyée à tous les bancs du même niveau. Ensuite une deuxième requête est initiée, elle demande la même ligne de la cible. Cette ligne dans l'état ISS passe dans l'état IS et l'identité du cache L1 (deuxième demandeur) est sauvegardé. Dès que la cible reçoit la donnée, les caches L1 seront servis. Mais, étant donné que la cible n'est pas le banc dans lequel réside cette donnée, la liste des caches L1 ayant une copie de la ligne doit être dans le banc contenant la ligne. Cette liste est ainsi transférée au banc possédant la ligne comme le montre la figure 4.6.

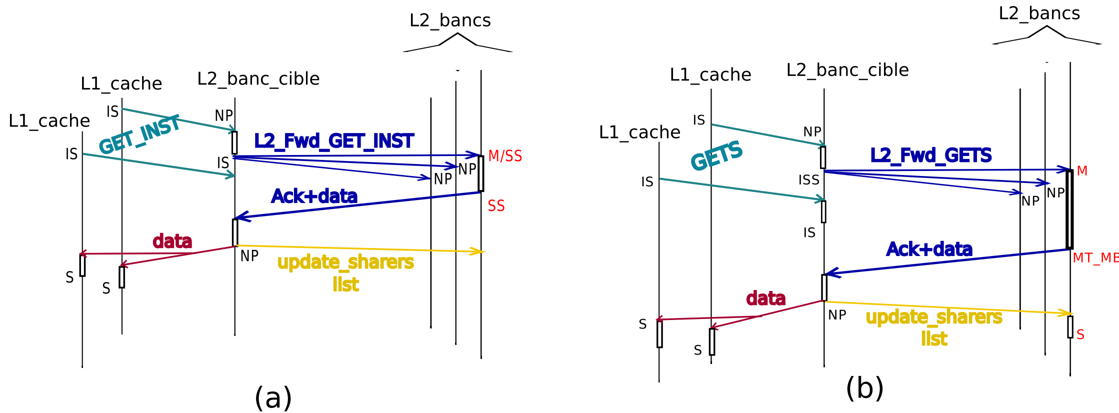


FIGURE 4.6 – Mise à jour de la liste des caches au niveau d'un banc L2 différent de la cible lors de plusieurs lectures d'un même bloc (a) d'instructions (b) de données

Conséquences sur les caches L1

Nous analysons dans cette section les conséquences sur le cache L1 lorsque la donnée est localisée dans un banc de cache différent de la cible. Nous commençons par les caches L1 initiateurs de la requête.

- **L1 initiateur de la requête**
  Lors d'une requête d'écriture "Store" la ligne passe dans l'état IM et une requête L1_GETX demandant la donnée du niveau inférieur (L2s) est initiée. Si la donnée sollicitée existe dans l'un des bancs dans l'état MT ou M, dès sa réception la ligne transite vers l'état M. Par contre, si la ligne était dans l'état SS dans le banc L2, la donnée est envoyée au demandeur L1 avec le nombre d'accusés de réception à recevoir suite aux invalidations. L'état de la ligne passe à SM indiquant que la donnée est bien reçue. Puis lors de la réception de tous les accusés la ligne passe dans l'état M.
  Lors d'une requête de lecture "Load" la ligne passe dans l'état IS et une requête est envoyée sollicitant cette ligne du niveau inférieur (L2s). Dans le cas où la ligne existe dans un banc L2 différent de la cible dans l'état MT ou SS, dès que l'initiateur reçoit la ligne, l'état de cette dernière passe à S. En revanche, elle passe dans l'état E si la ligne trouvée dans le banc L2 est dans l'état M. En ce qui concerne

la lecture d'une instruction « Ifetch » quel que soit l'état de la ligne dans le banc L2 (MT/M/SS), la ligne dans le cache L1 passe de IS à S.

– **Autres caches L1**
Dans le protocole, l'état MT est utilisé pour les lignes dans un banc de cache L2 pour indiquer que la ligne est à jour uniquement dans un cache L1 dont l'identité est sauvegardé. Ainsi, à la réception d'une requête d'écriture "L2_Fwd_GETX", une requête est relayée au cache L1 propriétaire pour avoir la donnée. La ligne dans ce cache initialement dans l'état E ou M passe dans l'état I dès qu'il reçoit cette requête.
Pour la requête de lecture de données ou d'instructions "L2_Fwd_GETS/ L2_Fwd_GET_INST", après le transfert de la requête au propriétaire L1 pour lui demander la ligne, cette dernière qui peut exister dans l'état M ou E transite vers l'état S. En effet, la ligne sera partagée par plusieurs caches L1 (initiateur de la requête, propriétaire initiale).
Si la ligne est dans l'état SS dans le banc de cache L2, lors de sa demande en permission exclusive (L2_Fwd_GETX), une invalidation des copies dans les différents caches L1 est primordiale. Ainsi, les copies de la ligne dans les caches L1 passent de l'état S à l'état I.

4.4.2.4 Requête L2_Fwd_Upgrade

Lorsqu'un processeur initie une requête d'écriture d'une donnée qui est partagée par plusieurs caches L1, la nécessité d'avoir la permission d'exclusivité nécessite d'envoyer une requête "Upgrade" au banc de cache L2 cible. Ceci a été relevé dans le protocole MESI de base. Néanmoins, dans notre architecture la donnée peut soit résider dans la cible soit ailleurs (i.e. dans un autre banc du même niveau). La première situation est gérée comme dans le protocole MESI standard alors que la seconde exige le déclenchement de la diffusion. Pour cela, des requêtes "L2_Fwd_Upgrade" sont envoyées par la cible à tous les bancs du même niveau.

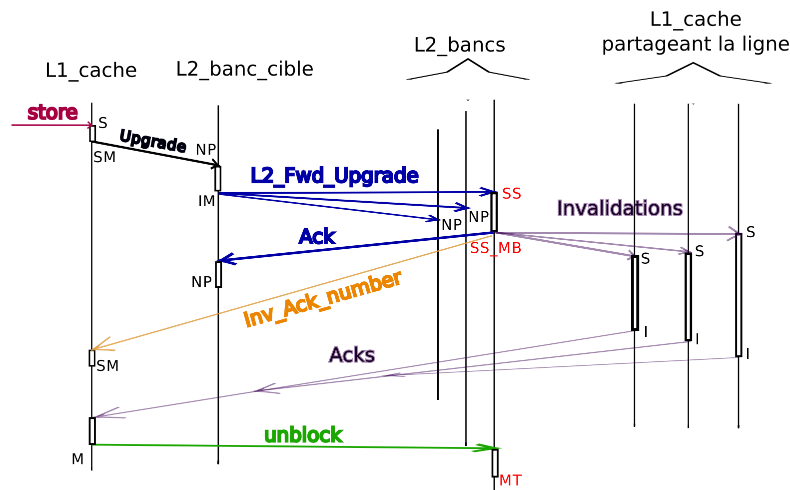


FIGURE 4.7 – Scénario d'écriture dans une ligne partagée (Upgrade)

La figure 4.7 illustre le scénario dans la deuxième situation. La requête "Store"

implique le passage de la ligne de l'état S dans l'état SM désignant l'attente d'une permission d'exclusivité. Ensuite, la requête "Upgrade" est envoyée à la cible. N'ayant pas la donnée cette dernière transmet les requêtes "L2_Fwd_Upgrade" aux bancs de cache L2. Une fois la donnée trouvée, ce banc agit comme suit :

- Un accusé de réception indiquant l'existence de la donnée est envoyé à la cible.
- La donnée est dans l'état SS dans ce banc, ce qui implique l'envoi de commandes d'invalidation à tous les caches L1 ayant une copie à part le demandeur actuel (celui qui a déclenché l'"Upgrade").
- Le nombre d'accusés des invalidations est transféré à l'initiateur de la requête "Store".

Donc, à la réception de tous les accusés, le cache L1 passe de l'état SM à l'état M et il envoie un message de déblocage "Unblock" au banc L2 dans lequel réside la donnée accompagnée de son identité. La ligne dans ce banc passe dans l'état MT.

4.4.2.5 Mécanisme d'évincement dans le cache L1

Lors de l'évincement d'une ligne du cache L1 (pour faire de la place), si cette ligne a été partagée (S) elle sera évincée automatiquement et son état transite à I. Mais, si la ligne a été modifié (M) elle doit être réécrite dans le cache L2 (le banc dans lequel réside cette donnée). Dans l'architecture à base du protocole MESI, si la donnée existe elle ne peut résider que dans le banc de cache L2 cible. Par contre, dans notre architecture, elle peut exister soit dans le banc cible soit dans un autre banc L2. Le premier cas est traité comme dans le protocole MESI de base. Le second impose la vérification entière du cache L2 (i.e. tous les bancs formant le cache L2) avant l'évincement de la ligne. De cela découle le besoin d'ajouter un état bloquant (NP_B) à une ligne qui n'existe pas dans la cible. L'affectation de l'état (NP_B) à la ligne après en avoir alloué l'espace permet d'assurer la consultation du reste des bancs. La figure 4.8 illustre ce scénario. Après l'envoi des requêtes "L2_Fwd_PutX" aux différents bancs, celui qui possède la ligne (mise à jour ou non) répond à la cible par un accusé de réception "WB_Ack" pour débloquer la ligne qui passe de NP_B à NP et cette cible se charge de transférer l'accusé indiquant que la ligne a été bien mise à jour dans le cache L2 (l'un des bancs) avant qu'elle soit évincée. La ligne dans le cache L1, étant dans l'état M_I, passe à I dès qu'il reçoit cet accusé. Il est à noter que pour la lisibilité du schéma, nous n'avons pas dessiné les accusés de réception des requêtes ayant sollicitées les autres bancs.

4.4.2.6 Mécanisme d'évincement dans le cache L2

L'évincement d'une ligne de cache L2 oblige la réécriture de la ligne dans la mémoire principale (ou dans le niveau de cache suivant) lorsque la ligne est modifiée. De plus, afin de garantir l'inclusivité des caches L1 dans le cache L2, des invalidations doivent être envoyées à tous les caches L1 ayant une copie (voir fig.4.9).

Notre protocole utilise les mêmes mécanismes permettant l'évincement que le protocole MESI standard. En effet, si la ligne évincée est dans un état partagée SS le cache L2 envoie des invalidations à tous les caches L1 ayant une copie et si elle est dans l'état modifié M la ligne est réecrite dans la mémoire principale. À l'état MT la situation nécessite à la fois l'invalidation de la copie et la mise à jour de la mémoire principale. Ces deux actions ne peuvent pas être envoyées parallèlement pour une simple raison :

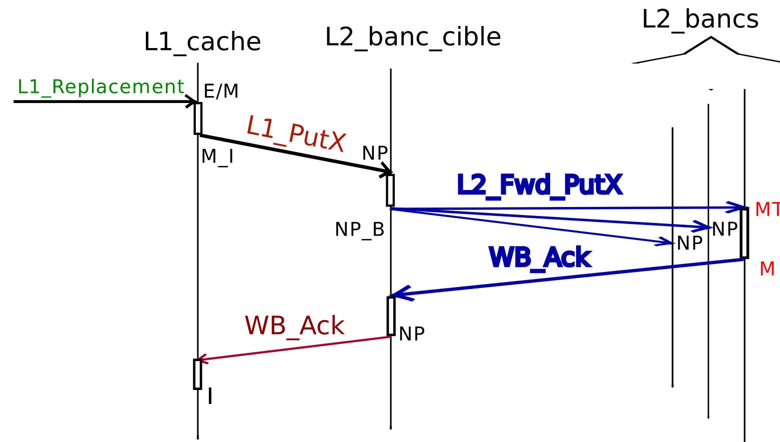


FIGURE 4.8 – Évincement d'une ligne résidente dans le cache L1

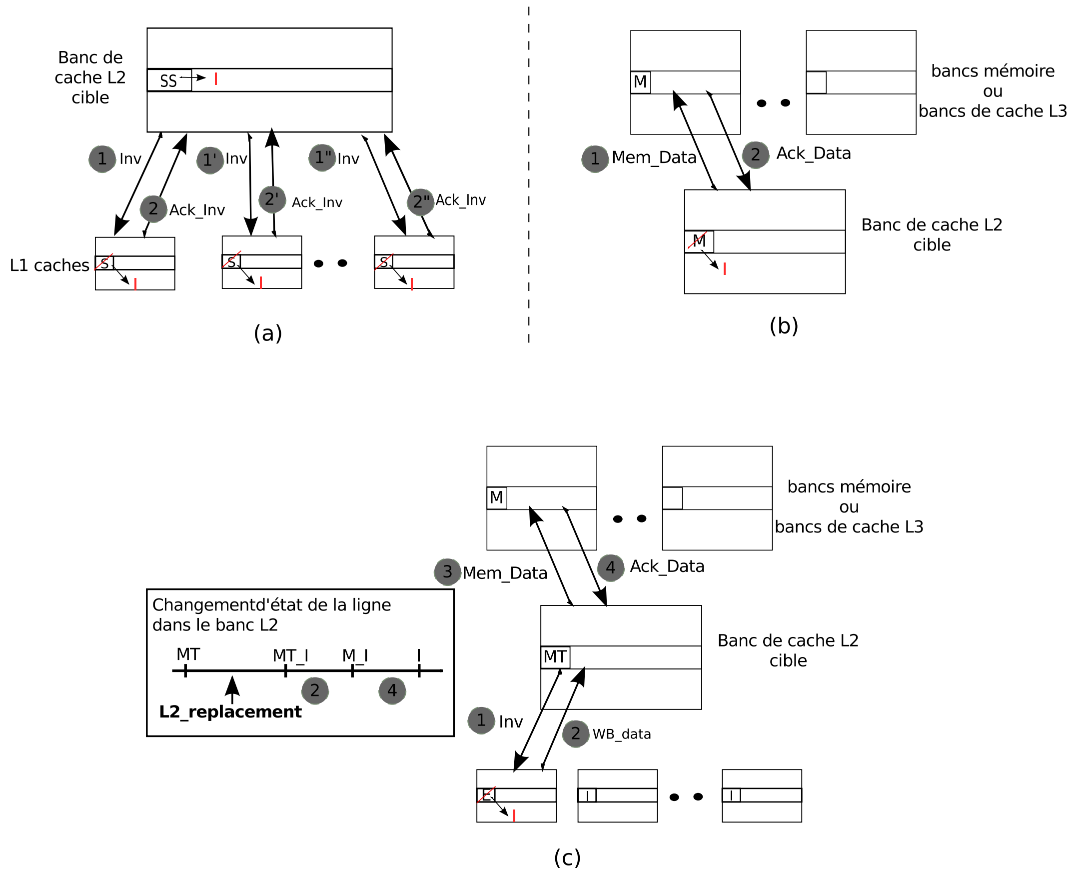


FIGURE 4.9 – Évincement d'une ligne du banc L2 dans différentes états (a) partagé, (b) modifié et aucun cache L1 ne possède la ligne, (c) modifié et la copie à jour est dans un autre cache L1

la ligne peut ne pas être à jour dans le cache L2. Débuter par l'envoi de l'invalidation à l'unique copie dans un cache L1 est une condition nécessaire. À la réception de la donnée à jour, la transaction d'écriture vers la mémoire se déclenche.

4.4.3 Interblocage

Dans cette partie nous décrivons la solution que nous avons adoptée pour résoudre le problème d'interblocage illustré dans la section 4.3.3. Un tel incident se produit lorsqu'une ligne est sollicitée par deux caches L1 n'ayant pas la même cible et qu'aucune de ces derniers ne la possède. Les deux requêtes initiées en même temps, avant qu'elles ne soient dupliquées et envoyées aux autres bancs, allouent chacune de l'espace pour accueillir la donnée demandée. L'état des lignes réservées est un état transitoire qui traduit que cette ligne ne pourra être ni lue ni écrite. Ces états sont IM, IS, ISS détaillés dans la section 4.4.2.2.

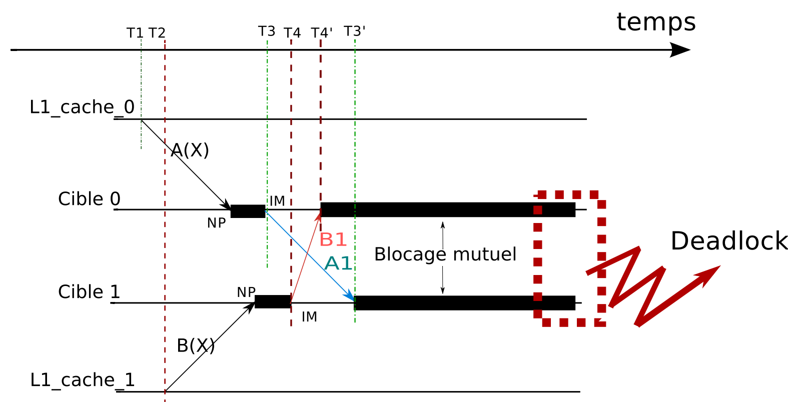


FIGURE 4.10 – Phénomène d'interblocage

La figure 4.10 schématise le phénomène d'interblocage. À l'instant T1 un cache L1 envoie une requête A à sa cible (0) en lui demandant une ligne X. Après un laps de temps un autre cache L1 déclenche une requête B sollicitant la même ligne X de sa propre cible (1). N'ayant pas la ligne, les cibles envoient les requêtes dupliquées aux autres bancs. Afin de ne pas encombrer le schéma, nous avons présenté uniquement la requête $A1$ (resp. $B1$) destinée à la cible de B, 1 (resp. A, 0). À l'instant T3' de la réception de la requête $A1$ la ligne est dans un état transitoire (IM). Elle est dans l'attente d'une réponse à la requête $B1$ initiée à T4. De même pour la requête $B1$ reçue par la cible 0 à T4', la ligne est dans l'attente d'une réponse. Ainsi, les deux sont bloquées mutuellement : chacune est dans l'attente de la réponse de l'autre, ce qui génère un interblocage.

Afin de résoudre ce phénomène, la solution proposée repose sur l'ajout d'un drapeau, le changement de la propriété des états transitoires existants et l'ajout des états transitoires bloquants. En effet, à la réception d'une requête d'écriture ou de lecture d'une ligne et que cette dernière est dans l'état IM, IS ou ISS, deux cas de figure peuvent avoir lieu :

- La ligne a reçue l'acquittement du banc qui initie la requête avant qu'elle n'ait été sollicitée (dans l'état NP) mais elle est encore dans l'attente du reste des accusés de réception.
- La ligne reçoit la requête quand elle est dans l'attente d'une réponse de l'initiateur de cette requête.

Face à ces situations la requête sera traitée et une réponse particulière est envoyée : c'est un acquittement spécifique indiquant que la ligne n'existe pas à l'instant mais qu'elle a déjà été demandée (drapeau activé). Ce type d'acquittement permet de débloquer les

requêtes et il fournit à la cible une information concernant le nombre de bancs dans l'état transitoire. Ce nombre sert à gérer les requêtes dans les niveaux inférieurs (cache L3 ou mémoire), ce qui sera décrit ultérieurement.

Il est important de noter que l'ordre de réception de ce type d'acquittement n'a aucune influence sur le traitement de la requête. En effet, la ligne dans l'état IM, IS ou ISS ne change pas jusqu'à la réception de tous les accusés de réception. Le dernier accusé, que ce soit un accusé simple signalant que la ligne n'existe pas ou un accusé particulier, transforme la ligne dans l'état IM_B, IS_B ou ISS_B selon le type de la requête. Ces états décrits dans la section 4.4.2.2 traduisent que la ligne ne peut être ni lue ni écrite, elle est dans un état bloquant et elle ne sera débloquée que par une réponse d'un banc du niveau inférieur (cache L3 ou banc mémoire). Dans le cas d'un banc mémoire, notre stratégie requiert des bits de méta-données comme dans d'autres stratégies.

Ayant résolu le problème du blocage mutuel, la question de savoir comment servir les requêtes relayées au niveau inférieur se pose. En effet, si les deux requêtes sont servies la réponse sera envoyée aux deux cibles et deux copies de la ligne existeront dans un même niveau de cache. Or, la particularité de l'organisation des mémoires locales est qu'une seule copie peut exister par niveau (sauf le L1 bien sûr).

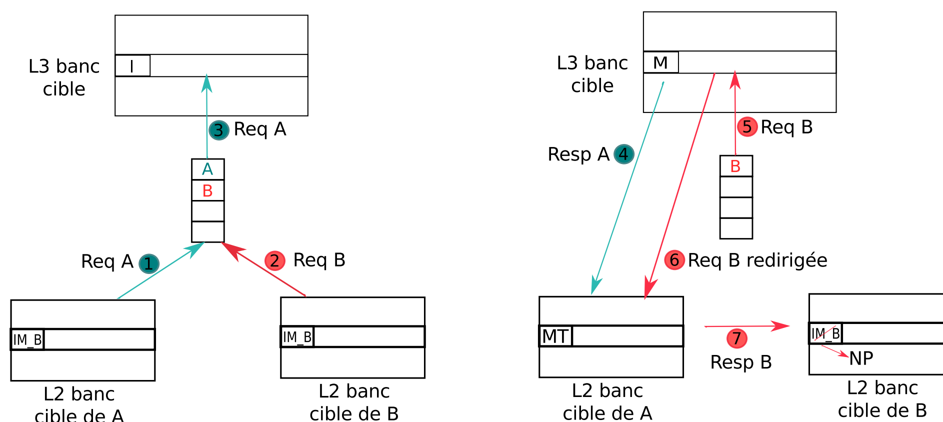


FIGURE 4.11 – Scénario de redirection

Pour répondre à cette question, nous supposons que les deux requêtes consultent la même cible dans le niveau qui suit (banc L3 cible ou banc de la mémoire) et que la ligne demandée existe dans cette cible. Les deux requêtes seront sérialisées à travers une file d'attente de type FIFO à l'entrée de la cible. La première requête reçue sera traitée et la réponse est envoyée à sa cible dans le niveau supérieur (banc L2 cible). La deuxième est redirigée vers le banc L2 cible dont la requête a été servie (cible intermédiaire) comme illustré dans la figure 4.11 . La redirection de la requête est réalisée à la réunion des conditions suivantes :

– Le nombre de bancs de cache L2 ayant la ligne dans un état transitoire est supérieur ou égale à 1. L'égalité à 1 n'implique pas qu'un seul banc ait la ligne dans l'état transitoire. En réalité, il y en a au moins deux, ce paramètre n'a pas été mis à jour car lorsque le banc a été sollicité, la ligne était invalide et l'accusé a été envoyé.
– La ligne dans la cible L3 ou dans le banc mémoire n'est pas dans l'état invalide (I).

La requête redirigée telle que présente sur la figure 4.11 est servie par la cible intermédiaire. Selon l'état de la ligne, la réponse est transmise à la cible réelle de la

requête.

4.5 Virtualisation de la hiérarchie

Les mécanismes de gestion et de cohérence de la hiérarchie des mémoires locales telle qu'elle a été introduite sont bâtis autour d'un support de communication. Notre hiérarchie n'est performante que si les accès y sont localisés, c.-à-d. si les données sont souvent réutilisées. Dans la situation inverse, les performances de l'unité de mémorisation sont déterminées par les transferts de données entre les niveaux de la hiérarchie.

En nous appuyant sur notre modèle architectural, nous définissons différents types de transferts entre les niveaux de mémoire supportant des temps d'accès variés. Un premier type donne la possibilité de transférer des données entre les différents niveaux hiérarchiques. Il reproduit le fonctionnement normal des caches classiques où les données passent d'un niveau à l'autre sur la base de la fréquence à laquelle elles sont référencées. Le second type offre une plus grande flexibilité en permettant les transferts de données au sein d'un niveau de mémoire cache (L2 ou L3). Ce type se justifie dans la mesure où il sera opportun pour certaines données non trouvées dans le cache L2 cible (resp. L3 cible) de s'affranchir de la traversée de toute la structure hiérarchique du cache pour atteindre la donnée qui existe déjà dans le niveau 2 (resp. niveau 3), mais dans un cache différent de la cible.

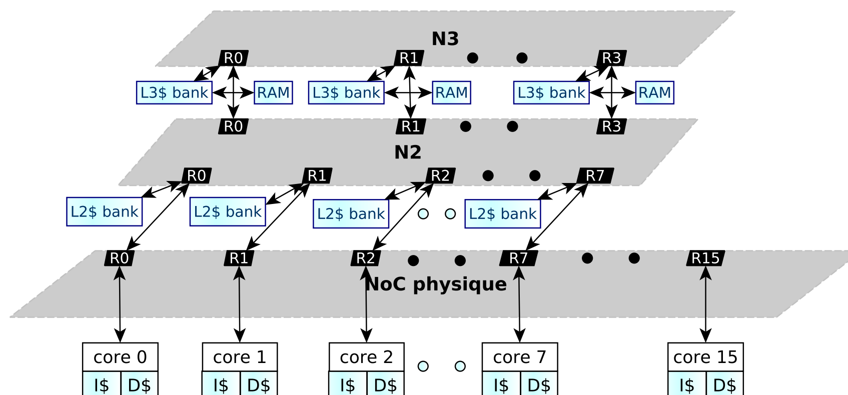


Figure 4.12 – Architecture logique

L'architecture logique présentée dans la figure 4.12 est la solution qui a permis de supporter les différents types de transfert au niveau de la hiérarchie mémoire. C'est une sorte de virtualisation favorisant un comportement adéquat avec les mécanismes adoptés. La figure 4.12 donne l'illusion d'avoir un système multi-couches dans lequel les différents niveaux de cache sont empilés sur les cœurs. Dans cette représentation purement intellectuelle chaque niveau de cache est une couche en soi. Ainsi, les caches L2 sont regroupés et interconnectés à travers un réseau N2, les caches L3 interconnectés à travers un deuxième réseau N3 localisé dans une deuxième couche, et ainsi de suite. Cette architecture permet de se débarrasser de la contention d'un réseau d'interconnexion. En contrepartie, elle révèle des variations des temps d'accès aux différents niveaux qui peuvent affecter sa performance. Il est donc intéressant de s'appuyer sur le réseau d'interconnexion et ses particularités.

4.6 Conclusion

Pour résumer, nous avons dans ce chapitre présenté une nouvelle structure de la hiérarchie de mémoires caches. Cette organisation permet d'exploiter efficacement l'espace de mémorisation disponible sur la puce en offrant plus de flexibilité au niveau de l'organisation des données manipulées. Dans ce cadre, la nouvelle politique de gestion de cache a été détaillée permettant d'éclairer l'attribution et la distribution des données au niveau de la hiérarchie.

Néanmoins, comme dans toute organisation le maintien d'un état cohérent des données est nécessaire. C'est un problème substantiel lorsque les processeurs de l'architecture doivent travailler de concert. Ainsi, nous avons défini et présenté de manière précise un protocole de cohérence adapté au mode en réecriture tardive (write-back). Ce protocole se rapproche du protocole MESI, qu'il étend pour assurer la cohérence intra-niveau de la hiérarchie. L'annexe donne les détails de ce protocole. Au final, les transferts inter et intra niveaux justifient le besoin d'un support de communication adéquat, la solution étant de définir une architecture logique dans laquelle la hiérarchie est virtualisée. Le système donne l'illusion d'un système multi-couches au sein duquel les niveaux de cache sont manipulés séparément.

Chapitre 5: Un réseau virtualisé comme support de communication

Sommaire

4.1 Structure de la hiérarchie des mémoires caches 36
4.2 Métriques d'évaluation des performances 37
4.3 Gestion de mémoires caches 38
4.3.1 Placement des données 38
4.3.2 Mécanisme de recherche de la donnée 39
4.3.3 Problème lié à la structure de la hiérarchie 40
4.4 Protocole de cohérence des caches 40
4.4.1 Principe 40
4.4.2 Implantation du protocole 41
4.4.3 Interblocage 51
4.5 Virtualisation de la hiérarchie 53
4.6 Conclusion 54

Dans le chapitre précédent, nous avons évoqué le besoin d'un support de communication ad-hoc : un réseau d'interconnexion qui assure un transfert au niveau de la hiérarchie mémoire à moindre coût (en terme de latence). Dans ce chapitre nous allons présenter la solution que nous proposons pour répondre au besoin.

La virtualisation du réseau est un concept assez répandu car il offre de nombreuses opportunités. Sa capacité de reconfiguration est l'une de ses caractéristiques qui le rend adaptable au besoin des applications. Le réseau virtualisé est un ensemble de nœuds, non nécessairement voisins, interconnectés via des liens logiques. Ce réseau nécessite un algorithme de routage adéquat pour l'acheminement des paquets. Cependant, comme toute stratégie de routage, l'enjeu principal est de garantir l'absence d'interblocage. Nous visons à éviter ce phénomène en exploitant les ressources existantes. Le concept de réseau virtuel tel que nous l'avons défini est utile dans plusieurs domaines d'application : sécurité, tolérance aux fautes, cohérence de caches, ...

La suite de ce chapitre est organisée comme suit : la section 5.1 présente le concept de virtualisation d'un réseau, la section 5.2 décrit le mécanisme de routage proposé pour le réseau virtualisé, la section 5.3 détaille la micro architecture des ports du routeur supportant ce routage. La section 5.4 illustre le problème d'interblocage associé à l'algorithme de routage et la solution à mettre en place afin de l'éviter. La solution proposée peut être exploitée dans plusieurs situations et quelques exemples d'utilisation sont évoqués dans la section 5.5. La section 5.6 conclut le chapitre.

5.1 Concept du réseau virtuel

Le mot « virtuel » est une alternative courante à « logique », ce terme est utilisé pour désigner un objet artificiel, par exemple la mémoire virtuelle par opposition à la mémoire physique d'un ordinateur. Le deuxième sens du mot est l'émulation d'une fonction ou d'un objet qui n'existe pas réellement.

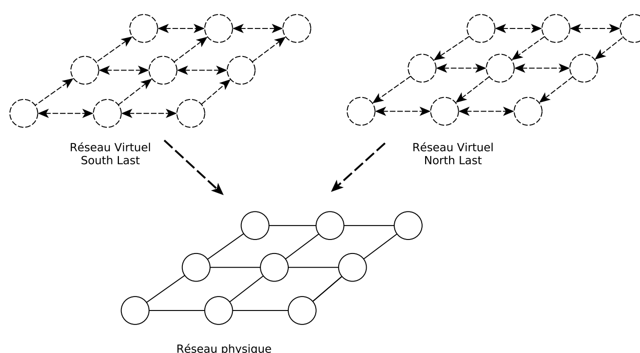


Figure 5.1 – Exemple de réseau virtuel

La notion de réseau virtuel(VN) a été introduite en 1995 [CA95]. Il s'agit d'un réseau logique projeté sur un réseau physique en utilisant les canaux virtuels (VCs). La figure 5.1 schématise deux VNs projetés sur le réseau physique qui possède 4 canaux virtuels. À chaque VN est attribué deux canaux. Dans cet exemple, à chaque VN est associé un algorithme de routage. Les paquets destinés aux nœuds dont la position est au sud de la source traversent le VN "North Last", et les paquets dont la destination est au nord de la source circulent dans le VN "SouthLast".

Le réseau virtuel n'est qu'un ensemble de nœuds interconnectés par des liens virtuels formant une topologie de réseau émulé. Ainsi, des nœuds physiquement distants peuvent être logiquement voisins en formant un réseau virtuel (cf. figure 5.2). L'avantage de ce type de réseau est qu'il permet de mettre deux nœuds distants en communication à très faible coût. Ce réseau se contente d'utiliser les ressources physiques existantes. De plus, chaque VN peut avoir son propre algorithme de routage et sa propre topologie. C'est autour de ce type de réseau que nous avons élaboré notre approche.

5.2 Routage Logique

Ce routage définit le chemin emprunté par un paquet entre la source et la destination dans un réseau logique. Le mécanisme de routage proposé pour ce type de réseau est complètement distribué, la décision de routage étant mise en œuvre dans chaque routeur en fonction de l'adresse cible et d'autres critères éventuels tel que la topologie.

Dans un réseau typique, un algorithme de routage distribué est utilisé. Selon l'algorithme, l'adresse courante et l'adresse de destination, l'un des nœuds voisins est désigné comme destinataire auquel le paquet reçu sera transféré. Le même mécanisme se produit au niveau des routeurs associés aux nœuds voisins jusqu'à ce que le paquet atteigne la destination finale. Par analogie, le même processus de routage aura lieu dans

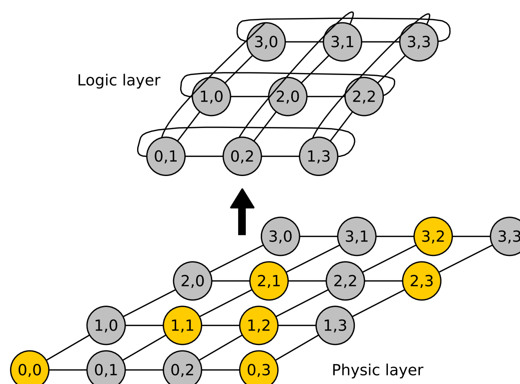


FIGURE 5.2 – Exemple de réseau logique ayant comme topologie un tore inclus dans un réseau physique de topologie maille

le réseau logique. Conformément à l'algorithme de routage, l'adresse courante et celle de destination, un nœud voisin parmi les voisins du routeur courant dans ce réseau sera choisi comme destination (intermédiaire ou finale) suivante.

Dans la suite, nous nous concentrons sur l'exemple d'un réseau physique ayant la topologie d'une maille 2D, sur lequel est projeté un réseau logique ayant la topologie d'un tore (cf. figure 5.2). Nous expliquons à travers cet exemple les principes du routage pouvant être adapté à n'importe quel réseau logique.

Dans l'architecture d'un réseau de type grille 2D, les adresses sont à la base des composantes (x,y). La projection d'un réseau logique sur ce dernier impose la distinction de deux types d'adresses pour l'acheminement des paquets : une physique et une logique. La première, et comme son nom l'indique, représente l'adresse réelle du nœud au niveau du réseau physique. La deuxième est une adresse représentative qui indique la position de chaque nœud dans le réseau logique. Donc, un paquet envoyé sur le réseau logique utilise des adresses logiques, mais réellement le transfert entre deux nœuds logiques est basé sur la position de ces deux nœuds dans le réseau physique.

Chaque routeur connaît la position physique de ses voisins logiques enregistré durant la phase de configuration. Afin d'envoyer un paquet à une destination intermédiaire, un nouvel en-tête est généré avec l'adresse physique du voisin logique (nouvelle destination physique). Atteignant ce voisin, l'en-tête est analysé pour distinguer s'il s'agit d'un en-tête ajouté ou de l'en-tête l'original. Dans le premier cas, cet en-tête est supprimé afin d'analyser l'en-tête original. Si le routeur courant ne représente pas la destination finale, le même scénario se répète : i.e. le masquage de l'en-tête original, un nouvel en-tête de l'adresse physique du voisin logique suivant est généré. Le nouvel en-tête contient un bit indiquant que ce dernier n'est pas l'original.

Ce mécanisme, que nous qualifions de méta-routage, est utilisé avec n'importe quelle topologie de réseau et elle peut exploiter n'importe quel algorithme de routage pour un réseau logique. Ainsi, les stratégies de routage et/ou les algorithmes au niveau des réseaux logiques peuvent se transformer en une série de routages dans le réseau physique. Comme le chemin physique reliant la source originale à la destination n'est pas minimal, la projection des réseaux logiques doit être optimisé.

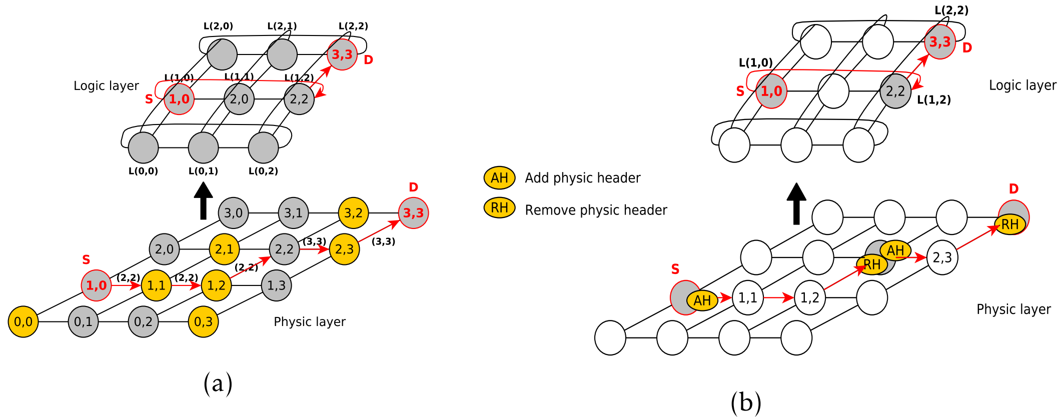


Figure 5.3 – (a) Exemple de méta-routage logique au niveau d'un réseau logique de topologie tore inclus dans un réseau physique de topologie maille (b) Ajout et suppression de l'en-tête temporaire au niveau des routeurs du réseau logiques

5.2.1 Exemple de méta-routage

La figure 5.3a illustre un exemple de ce méta-routage. Le nœud L(1,0) du réseau logique ayant la topologie d'un tore veut envoyer un paquet au nœud L(2,2) de ce même réseau. Les deux nœuds logiques correspondent aux positions physiques (1,0) et (3,3) respectivement. L'envoi du paquet de L(1,0) à L(2,2) suit le chemin logique suivant : L(1,1)→L(1,2)→L(2,2). Mais le paquet traverse réellement le réseau physique donc le chemin est comme suit p(1,0)→p(1,1)→p(1,2)→p(2,2)→p(2,3)⟶p(3,3).

Au niveau du nœud source le paquet est construit, il contient l'adresse du destinataire logique. Afin d'atteindre le nœud logique voisin, un recours à son adresse physique est indispensable. Alors, un en-tête temporaire est ajouté au paquet, contenant (2,2) comme destination. (2,2) est l'adresse physique du nœud logique L(1,2) qui est le voisin logique west du L(1,0). Le paquet est acheminé à ce nœud en utilisant l'algorithme de routage X-first. À son arrivée au nœud (2,2) qui est un nœud logique intermédiaire, l'en-tête ajouté est remplacé par un deuxième en-tête temporaire contenant (3,3) comme destination dans l'étape suivante du trajet. Cette destination est finale, donc à l'arrivée du paquet l'en-tête temporaire est supprimé et le paquet est consommé par le nœud. L'ajout et la suppression de l'en-tête sont représentés dans la figure 5.3b.

Pour généraliser, un paquet issu d'une source S dont la destination est D et en l'absence d'une liaison directe entre S et D dans le réseau physique, la source ajoute un en-tête temporaire : il s'agit d'un flit (*FLow control unIT*), l'unité de contrôle de flot du réseau, qui est ajouté de telle sorte qu'il cache l'en-tête original (destination logique). Cet en-tête temporaire contient l'information suivante : les coordonnées physiques du nœud logique à consulter et un drapeau A_H qui indique que le présent en-tête n'est pas l'original.

Chaque nœud logique doit contenir l'adresse physique de ses voisins logiques. Ayant 4 voisins le routeur associé au nœud concerné sauvegarde l'adresse physique de ces derniers. En se basant sur l'exemple précédent et choisissant comme source le nœud logique L(1,0), ses voisins dans le réseau logique sont L(1,1), L(1,2), L(0,0) et L(2,0) qui ont comme adresses physiques p(2,0), p(2,2), p(0,1) et p(3,0) respectivement. Ces

informations seront sauvegardées au niveau de chaque routeur.

5.2.2 Routage au niveau du port d'entrée local

Le routage au niveau de l'entrée locale est présenté dans la figure 5.4a par un organigramme. Lorsque le routeur reçoit l'en-tête, une vérification du drapeau A_H a lieu pour identifier le réseau sur lequel le paquet doit circuler : son activation désigne que le réseau à traverser est un réseau logique.

Si le paquet n'appartient à aucun réseau logique, l'algorithme de routage physique est utilisé et le paquet est injecté dans le réseau physique. Dans le cas contraire, c.-à-d. que le paquet fait partie d'un réseau logique, le module de routage indique le voisin logique auquel le paquet sera délivré. L'adresse physique qui lui correspond est lue à partir d'un registre et l'en-tête temporaire est ajouté au paquet. Ensuite, ce dernier sera transmis au port de sortie correspondant.

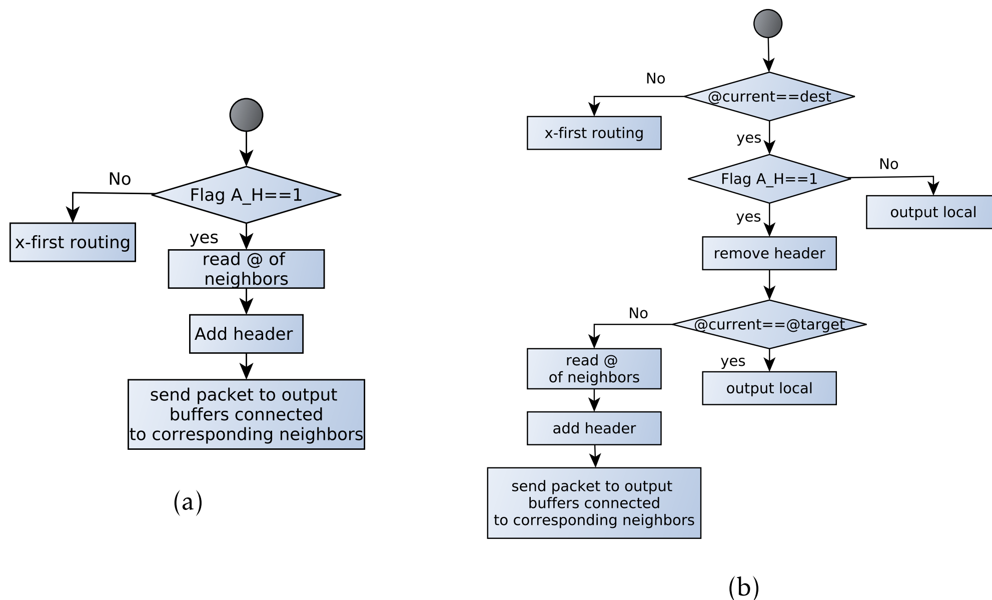


Figure 5.4 – L'algorithme de routage au niveau (a) du port d'entrée local et (b) des ports d'entrée (E,W,N,S)

5.2.3 Routage au niveau des ports d'entrées (E,W,N,S)

De la même manière, on décrit le comportement des ports d'entrée (E,W,N,S) via un organigramme dans la figure 5.4b. En fait, la décision du routage aura lieu au niveau du port d'entrée afin de déterminer le port de sortie approprié pour l'envoi du paquet. Ces ports ont la charge de la suppression des en-têtes temporaires, en plus de l'ajout d'en-tête qui est similaire à celui du port d'entrée local.

Ainsi, à la réception d'un paquet, le port vérifie d'abord si le nœud courant correspond à la destination. Dans l'affirmative et si l'en-tête est permanent, cela signifie que ce nœud est la destination physique finale et par conséquent le paquet sera consommé par le port de sortie local. Si l'en-tête est temporaire, cela signifie que ce nœud est la cible logique, éventuellement une destination intermédiaire et non la destination finale. Dans

ce cas, le port d'entrée supprime l'en-tête temporaire et regarde s'il doit transmettre le paquet au port de sortie local ou bien s'il lui faut reconstruire un nouvel en-tête et le transmettre au nœud logique suivant.

5.3 Micro architecture des ports d'un routeur

D'un point de vue microarchitecture, la suppression de l'en-tête temporaire est réalisée au niveau des ports d'entrées 2D (E,W,N,S) alors que l'ajout peut avoir lieu en plus au niveau du port d'entrée local. Lorsque le tampon du routeur courant n'est pas vide (*ROK*) et celui à l'entrée du routeur en aval n'est pas rempli (*WOK*), la machine d'états associée à chaque port peut générer un signal d'ajout ou de suppression selon le besoin du paquet.
L'architecture des différents ports d'entrée englobe le module de routage, un tampon et une machine d'états qui décide du destin du paquet. Nous commençons par la présentation du port local ainsi que la machine d'états qui lui est associée.

Dès que le port d'entrée local reçoit un paquet, le type de réseau (logique/physique) à traverser est déjà connu. S'il s'agit d'un réseau logique, le paquet subit une modification : un flit est ajouté à l'en-tête contenant l'adresse physique du nœud logique voisin auquel le paquet sera envoyé. Dans le cas contraire, le paquet est transmis à sa destination transcrite dans son en-tête original. La figure 5.5 représente la machine d'états du port local. Comme nous le verrons, la vérification du réseau auquel appartient le paquet est réalisée dans l'état initial "init". Pour transiter sur le réseau logique (L=1), le paquet passe à l'état "AH", au cours duquel un en-tête temporaire est ajouté au paquet. Si le paquet appartient au réseau physique, il passe directement dans l'état de transmission ordinaire "OT".

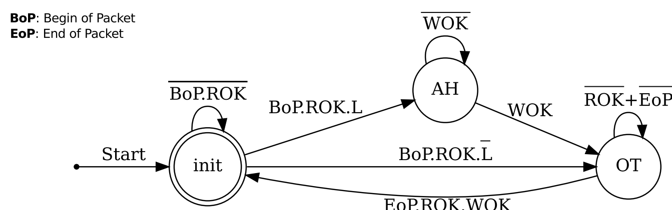


Figure 5.5 – Machine d'états du port local du routeur

En outre, la machine d'états du port d'entrée local commande le signal Sel_AH pour manipuler les signaux Read et Write lors de l'ajout d'un nouvel en-tête. Lors de son activation, le tampon associé au port d'entrée ne reçoit pas le signal Read, alors l'en-tête temporaire du paquet ne sera pas retiré. Simultanément, le signal Sel_AH active le signal write du routeur suivant, donc l'en-tête temporaire sera écrit à la place de l'en-tête original (figure 5.6). Lorsque le port d'entrée suivant reçoit avec succès la donnée, le signal Sel_AH est désactivé. Au cycle d'horloge suivant l'en-tête original et le reste des flits du paquet seront transférés au port de sortie correspondant.

En ce qui concerne les autres ports d'entrée (N-S-E-W) leur machine d'états est plus compliquée, vu qu'ils doivent supporter les opérations d'ajout et de suppression d'en-tête. La figure 5.7 schématise cette machine d'états. Si le drapeau A_H de l'en-tête du paquet est activé, cela signifie que l'en-tête est temporaire. De plus, si le signal associé au

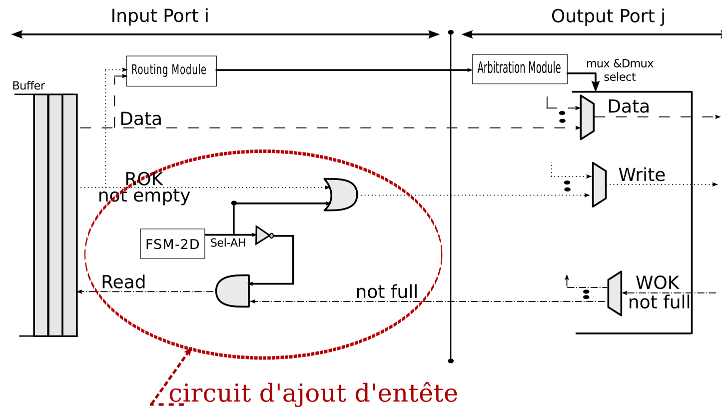


FIGURE 5.6 – Architecture du port local du routeur

port de sortie physique R_p est actif, l'en-tête doit être supprimé. Le paquet transite vers l'état "RH". L'activation du signal de commande généré par la machine d'états "Sel_RH" implique l'activation du Read (pour la lecture du flit) et la désactivation du Write (pour interdire son écriture). La figure 5.8a illustre le circuit de suppression du flit. Cette opération consomme un cycle d'horloge.

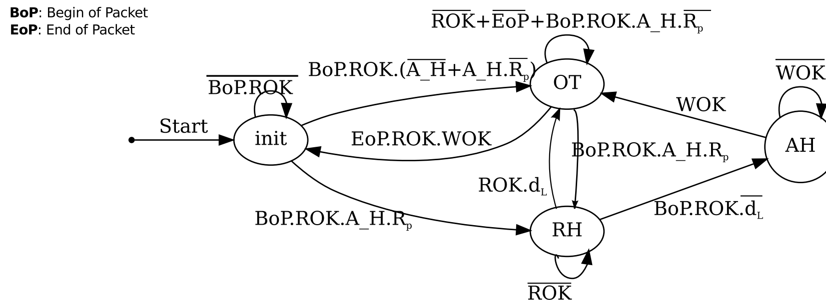


FIGURE 5.7 – Machine d'états pour les ports d'entrées 2D

Une fois désactivée (cycle suivant), la destination dans l'en-tête original (second flit du paquet) est comparée aux coordonnées du routeur courant. Une égalité (d_L=1) implique le passage du paquet à l'état de transmission ordinaire "OT". Sinon le paquet transite vers l'état "AH" au cours duquel un nouvel en-tête temporaire est ajouté contenant l'adresse physique du voisin logique suivant à consulter.

Fondamentalement, la micro-architecture des ports d'entrée (N-S-E-W) représentée par la figure 5.8b inclut les circuits d'ajout et de surpression de l'en-tête. Notons que chaque mécanisme requiert quelques portes logiques afin d'être mis en œuvre.

5.4 Interblocage

L'interblocage est le processus qui fait qu'un paquet doit attendre indéfiniment avant d'atteindre sa destination du fait qu'un ensemble de paquets sont bloqués dans le réseau. La situation d'interblocage se produit lorsqu'un ensemble de paquets demandent en

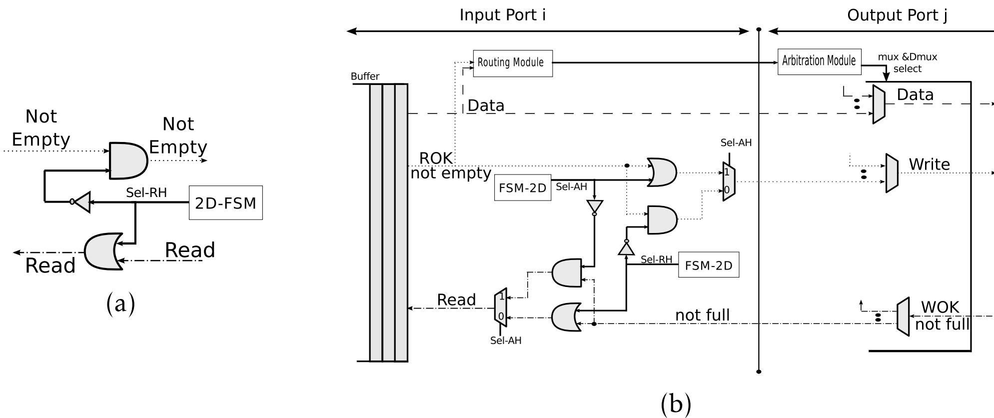


FIGURE 5.8 – (a) Circuit de suppression de l'en-tête temporaire (b) Architecture du port d'entrée 2D

même temps des ressources que certains détiennent alors que d'autres sont en attente dessus, et ce de manière croisée.

Pour éviter ce phénomène, il faut choisir un algorithme de routage qui y est insensible, tel que le routage X-first sur une grille 2D, ou qui puisse éviter la dépendance circulaire entre les paquets. À ceci s'ajoute le recours aux canaux virtuels. En effet, un paquet temporairement bloqué peut être doublé par un autre paquet et stocké sur un autre canal virtuel.

5.4.1 Situation

Dans la figure 5.9a, la stratégie de routage adaptée ne garantit pas l'absence d'interblocage. Notre algorithme de routage s'appuie sur le routage ordonné par dimension X-first. Le chemin traversé par le paquet dans le réseau physique du nœud (1,0) au nœud (3,3) ne respecte pas les règles du routage ordonné par dimension. En effet, au niveau du nœud (2,2), une violation du routage X-first est visible.

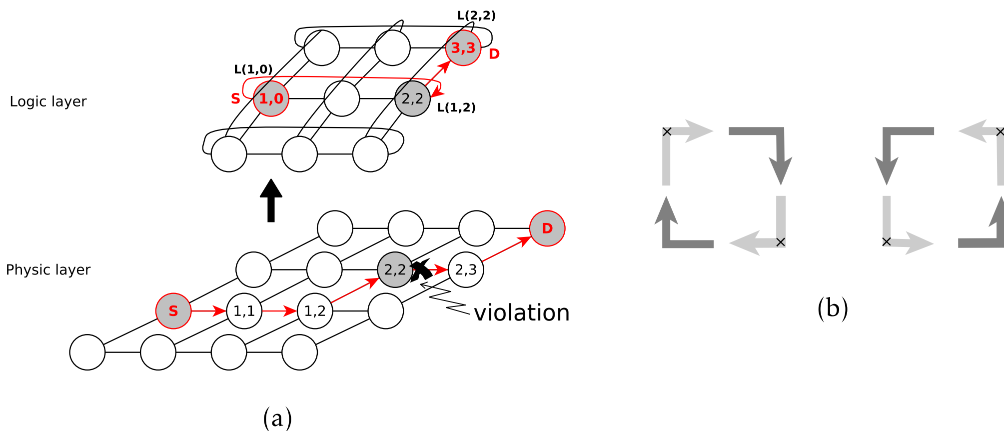


FIGURE 5.9 – (a) Violation dans le réseau physique (b) Restriction dans le modèle tour pour l'algorithme X-first dans une topologie maillé NoC

Afin de comprendre la notion de violation, nous nous référons aux algorithmes de routage à base du modèle de tours (*turn-model*)[GN92]. Dans ce modèle, certains tours sont restreints pour la communication en fonction des règles utilisées. En topologie grille et pour l'algorithme de routage X-first, sur huit tours possibles la moitié est autorisée (figure 5.9b). Selon cet algorithme, le paquet doit être acheminé le long de l'axe X jusqu'à atteindre la même colonne que celle de la destination. Ensuite, il est routé le long de l'axe Y en direction du destinataire. La violation consiste à ne pas respecter une de ces deux règles. C'est le cas dans l'exemple précédent lorsque le paquet traverse un chemin non autorisé en allant du nœud (2,2) vers le nœud (3,3). Le paquet en provenance du port Sud (dimension Y) est dirigé vers le port Est (dimension X) et ceci rend un blocage possible.

5.4.2 Recouvrement d'interblocage

Pour éviter la situation d'interblocage, les auteurs dans [SMBM10] ont proposé un algorithme applicable à toutes les topologies de réseau sur puce et adaptable à toutes les stratégies de routage. Il s'agit d'utiliser un certain nombre de canaux virtuels. Inspiré par cette solution, nous avons défini une règle simple : Lors de la violation des règles d'un routage sans interblocage, le paquet doit changer de canal virtuel. Ceci permet de s'abstenir des communications circulaires dans le graphe de dépendance du canal.

Dans l'exemple illustré précédemment, un second canal virtuel est nécessaire pour transférer le paquet du nœud (2,2) au nœud (3,3). Notre algorithme de routage est notamment déterministe, donc nous sommes en mesure de détecter toutes les violations du routage physique (i.e. au niveau du réseau physique) dues aux chemins empruntés dans les réseaux logiques.

Le nombre de canaux virtuels (VCs) nécessaire pour garantir l'absence de blocage peut donc être déterminé à l'avance (voir équation 5.1). Ce nombre est égal au nombre maximal de violations d'un chemin i, parmi tous les chemins logiques possibles (P) entre les sources et les destinations, plus un.

$$VCs = 1 + \max_{\forall i \in \{P\}} V_i \tag{5.1}$$

avec V_i : nombre de violations sur le chemin i

P : Tous les chemins entre source et destination

Les canaux virtuels sont des tampons physiques et en se basant sur les ressources matérielles disponibles, les topologies logiques supportées qui garantissent l'absence de blocage peuvent être déterminées. Autrement, pour un réseau physique à n canaux virtuels, les réseaux logiques supportés ne doivent pas dépasser les $n-1$ violations. Par voie de conséquence, la question de l'optimisation de la construction des réseaux logiques se pose et devra prendre en compte les contraintes de ressources.

5.5 Exemples d'utilisation du concept de réseau logique

5.5.1 Applications qui requièrent différentes topologies

Le choix de la topologie pour une architecture NoC est fortement lié aux contraintes de l'application et au trafic du réseau. Ce choix est complexe, notamment face au com-

promis entre la performance intrinsèque de la topologie et son coût d'implémentation. De ce fait, le concept de réseau logique avec sa topologie associée pourrait constituer une alternative qui assure un meilleur support de la qualité de service des applications.

Comme illustré auparavant, la construction d'un réseau logique n'est contrainte ni en nombre de nœuds, ni en topologie. Donc, chaque application peut avoir son propre réseau logique. La seule contrainte est le nombre de canaux virtuels disponibles qui devra être prise en compte lors de la projection des VNs sur le réseau physique. À l'inverse des solutions de partitionnement existantes, les VNs peuvent partager les ressources matérielles et un nœud d'un réseau peut appartenir plusieurs VNs.

5.5.2 Sécurité vis-à-vis des applications malveillantes

Avoir plusieurs applications sur une seule plate-forme rend le système vulnérable si l'espace de données des applications est accessible depuis l'extérieur de l'application. Afin de garantir un certain niveau de sécurité, les applications indépendantes ne doivent pas avoir accès aux données et à l'espace d'adressages les unes des autres. Même si les approches classiques telles que les machines virtuelles qui s'appuient sur des solutions logicielles ont fait leur preuves, des garanties supplémentaires peuvent être demandées dans certains contextes applicatifs. Assurer par exemple que l'exécution de plusieurs systèmes d'exploitation sur une même plateforme est totalement indépendante, et que donc ces OSs n'accèdent pas l'espace d'adressage l'un de l'autre, nécessite un support matériel spécifique.

Ainsi, le concept de VNs indépendants qui introduit une logique de séparation de flux peut être une solution qui répond aux besoins. Le fait que les paquets soient incapables de traverser des réseaux logiques (le réseau dans lequel ils sont injectés), et que chaque système logique possède son propre espace d'adressage, empêche tout type d'infiltration de l'extérieur de l'application ou du système d'exploitation.

5.5.3 Routage tolérant aux fautes

Le problème à résoudre lors de la défaillance d'un nœud (lien ou routeur) est celui de la perte de connectivité et de régularité dans le réseau. Plusieurs travaux ont été menés au niveau du routage consistant à reconfigurer les routeurs voisins de celui fautif pour créer des contournements de zone [ZGT08, ZGB10]. La régularité de ces zones est une condition nécessaire et ceci induit de sacrifier des routeurs sains.

Le concept de VN permet de développer un mécanisme de routage innovant, tolérant aux fautes et en tant que tel il permet de s'affranchir de ces problèmes de connectivité et de régularité. La méthode de routage est basée sur les principes fondamentaux du méta-routage expliqué dans la section précédente, avec des modifications mineures.

Afin de détailler la manière d'exploiter le concept de VN pour tolérer les fautes, nous nous appuyons sur la figure 5.10a, dans laquelle les nœuds défectueux sont représentés par X et les nœuds sains par O. Dans le but de contourner la zone défectueuse, nous formons un VN avec une topologie en anneau dirigée, ce qui signifie que chaque nœud logique a un seul voisin logique, qui est le nœud suivant dans l'anneau. Notons que les voisins dans cet anneau logique doivent également être des voisins dans le réseau physique. Lorsqu'un paquet atteint un nœud X et qu'il se rend compte qu'il ne peut pas suivre son chemin vers la destination, il retourne vers l'anneau VN afin de contourner la

région X. En traversant le réseau logique (chemin logique), une fois que le paquet atteint un nœud dont le voisin selon le chemin d'origine (X-first) est sain, il quitte l'anneau et reprend son acheminement sur le réseau d'origine vers sa destination. La ligne bleu (en pointillée) de la figure 5.10a est un exemple d'un tel chemin.

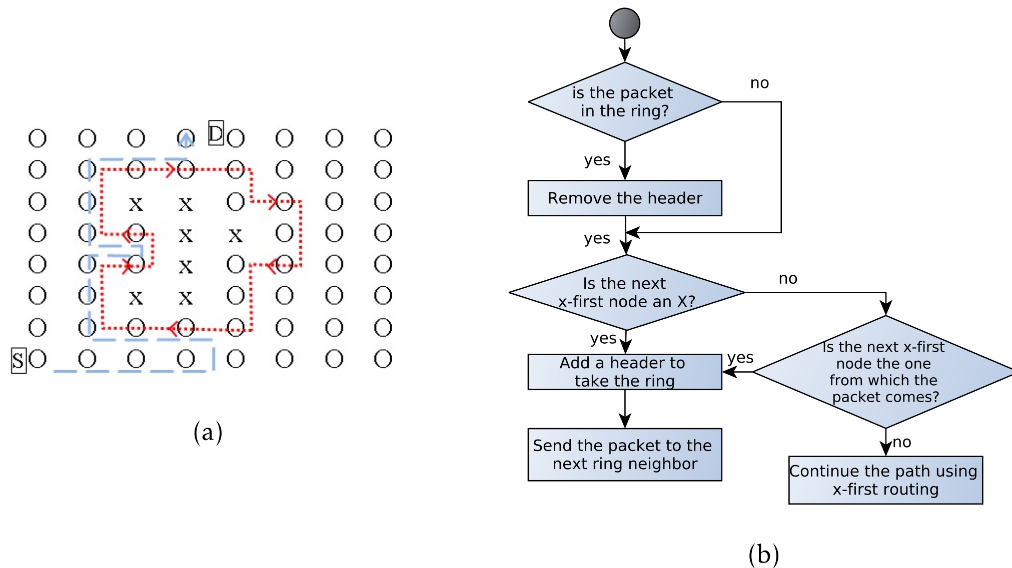


Figure 5.10 – (a) Un VN de topologie anneau contourne la région défectueuse pour diriger la paquet vers un chemin sain (b) Algorithme de routage tolérant aux fautes

Contrairement à l'idée originale du VN, nous constatons que dans cette situation le paquet change de réseau. En effet, lorsque le paquet doit prendre l'anneau VN, il sera encapsulé dans un nouveau paquet avec un nouvel en-tête pour être routé dans l'anneau. Au niveau de chaque nœud logique le paquet sera décapsulé pour vérifier si le nœud suivant sur le chemin d'origine est un X ou un O. L'organigramme présenté par la figure 5.10b traduit l'algorithme de routage tolérant aux fautes.

5.5.4 Protocole de cohérence de cache

Pour assurer la cohérence mémoire au sein d'un système à mémoire partagée, l'utilisation d'un réseau sur puce nécessite dans la majorité des situations l'envoi d'un message à tous les nœuds (diffusion ou *broadcast*), ou à un sous-ensemble de nœuds (multidiffusion ou *multicast*). La multidiffusion est en pratique complexe à implanter sur un réseau, ce qui en limite l'intérêt par rapport à la diffusion.

Les cibles des messages étant les caches d'un même niveau, nous proposons donc de connecter logiquement tous les caches d'un même niveau pour former des réseaux logiques virtuels correspondants à chacun des niveaux de cache, e.g VN1 pour les L1s, VN2 pour les L2s, etc ... Ultérieurement, nous pourrons utiliser un algorithme de routage de diffusion efficace dans chaque VN. Cette partie sera détaillée dans le chapitre suivant.

5.6 Conclusion

Nous avons proposé dans ce chapitre une adaptation du concept de virtualisation d'un réseau sur puce. Le principe de la virtualisation repose sur l'utilisation de ressources matérielles disponibles et vise à répondre à la problématique de gestion des communications dans une architecture distribuée. Ce concept présente de nombreux avantages. Un premier avantage est que des nœuds distants physiquement peuvent être interconnectés logiquement et ceci n'engendre aucun surcoût matériel. Le deuxième avantage toujours lié à l'architecture est sa flexibilité vis-à-vis de la topologie du réseau, notamment avec la possibilité de définir plusieurs topologies pour chaque réseau virtuel projeté sur le réseau physique. Le troisième avantage est que cette virtualisation permet de fournir des services de plus haut niveau à peu de frais, par exemple aider à garantir la sécurité à l'égard d'applications malveillantes, ou encore être en mesure d'assurer le fonctionnement lors d'une défaillance au niveau d'un routeur ou d'un lien. Au final, c'est aussi une solution intéressante à évaluer dans le cadre d'un protocole de cohérence associé à une hiérarchie de mémoires caches.

Le concept de virtualisation de réseau en tant que tel requiert une stratégie de routage particulière. Or, une condition nécessaire de cette stratégie est qu'elle doit garantir l'absence d'interblocage. Cette garantie peut être donnée, et elle s'appuie uniquement sur les ressources disponibles de l'architecture.

Chapitre 6: Stratégie de multidiffusion utilisant le concept du réseau virtualisé

Sommaire

5.1 Concept du réseau virtuel 58
5.2 Routage Logique 58
5.2.1 Exemple de méta-routage 60
5.2.2 Routage au niveau du port d'entrée local 61
5.2.3 Routage au niveau des ports d'entrées (E,W,N,S) 61
5.3 Micro architecture des ports d'un routeur 62
5.4 Interblocage 64
5.4.1 Situation 64
5.4.2 Recouvrement d'interblocage 65
5.5 Exemples d'utilisation du concept de réseau logique 66
5.5.1 Applications qui requièrent différentes topologies 66
5.5.2 Sécurité vis-à-vis des applications malveillantes 66
5.5.3 Routage tolérant aux fautes 66
5.5.4 Protocole de cohérence de cache 67
5.6 Conclusion 68

Un réseau virtuel supportant un mécanisme de routage adéquat est un besoin de la hiérarchie des mémoires caches que nous proposons. Du fait la manière dont les données résident et sont gérées par la hiérarchie, le raffinement du support de communication est une brique principale de notre approche.

Au travers du chapitre précédent, nous avons décrit le principe général du réseau virtuel et du routage en son sein. Dans ce chapitre, nous rendons ce principe concret pour l'organisation de notre hiérarchie mémoire qui nécessite des réseaux virtuels reliant les bancs d'un même niveau de cache. Ces réseaux servent à la circulation des paquets de diffusion, mécanisme requis lors d'un échec au niveau de la cible.

Ce chapitre est organisé comme suit : la section 6.1 décrit le modèle de diffusion à mettre en place et une abstraction de l'algorithme. La micro architecture du routeur requiert son adaptation, d'une part pour la formation des réseaux virtuels par niveau de cache et d'autre part pour supporter la multidiffusion. Ceci fera l'objet de la section 6.2. Finalement, une synthèse englobant les différentes parties de la solution est illustrée dans la section 6.3 à travers un exemple de recherche de données dans la hiérarchie.

6.1 Modèle de diffusion

6.1.1 Calcul des différents voisins

Comme nous l'avons mentionné dans le chapitre 3, lors d'un échec au niveau de la cible, les bancs du même niveau doivent être consultés. Il s'agit d'une forme de diffusion et la manière dont la diffusion se produit est la suivante :
Nous définissons la fonction *sign*(x).

$$sign(x) = \begin{cases} 1 & \text{si } x > 0 \\ -1 & \text{si } x < 0 \end{cases}$$

En faisant l'hypothèse que le réseau logique est une grille ou un tore de taille $n \times n$, à partir d'un nœud cible qui est l'origine de la diffusion (x_0,y_0) le message est envoyé aux nœuds voisins (x, y) comme suit :

1er cas : $x = x_0$ et $y = y_0$
Le message est envoyé aux 4 nœuds voisins $(x, y+1)$, $(x, y-1)$, $(x+1, y)$, $(x-1, y)$.

2ème cas : $x = x_0$ ou exclusif $y = y_0$.

– $x = x_0$ et $y \neq y_0$ (tout au long de l'axe des y)

$sign(y-y_0) \times (\mid y-y_0 \mid \mod n)$	négatif	positif
impair	$(x, y-1)$ et $(x+1, y)$	$(x, y+1)$ et $(x-1, y)$
pair	$(x, y-1)$ et $(x-1, y)$	$(x, y+1)$ et $(x+1, y)$

– $x \neq x_0$ et $y = y_0$ (tout au long de l'axe des x)

$sign(x-x_0) \times (\mid x-x_0 \mid \mod n)$	négatif	positif
impair	$(x-1, y)$ et $(x, y-1)$	$(x+1, y)$ et $(x, y+1)$
pair	$(x-1, y)$ et $(x, y+1)$	$(x+1, y)$ et $(x, y-1)$

3eme cas : $x \neq x_0$ ou $y \neq y_0$

– $x > x_0$ et $y > y_0$

$sign((x-x_0)+(y-y_0)) \times (\mid (x-x_0)+(y-y_0) \mid \mod n)$	
pair	$(x+1, y)$
impair	$(x, y+1)$

– x < x_0 et y < y_0

$sign((x-x_0)+(y-y_0)) \times (\mid (x-x_0)+(y-y_0) \mid \mod n)$	
pair	$(x-1, y)$
impair	$(x, y-1)$

– $x < x_0$ et $y > y_0$

$sign((x-x_0)+(y-y_0)) \times (\mid (x-x_0)+(y-y_0) \mid \mod n)$	
pair	$(x, y+1)$
impair	$(x-1, y)$

– $x > x_0$ et $y < y_0$

$sign((x-x_0)+(y-y_0)) \times (\mid (x-x_0)+(y-y_0) \mid \mod n)$	
pair	$(x, y-1)$
impair	$(x+1, y)$

Dans le premier cas, le message est envoyé aux 4 nœuds voisins. Pour le deuxième cas, en partant d'un nœud ayant la même abscisse ou ordonnée que l'origine, le message sera envoyé à deux nœuds. Pour le troisième cas, partant d'un nœud dont les coordonnées diffèrent de celles de l'origine, le message est envoyé à un unique voisin.
Cet algorithme est utilisable à la fois pour une topologie de type grille ou tore[YW99]. Dans le cas du tore, il faut un test supplémentaire pour s'assuer qu'un message ne soit pas envoyé à un nœud deux fois. Ainsi, pour un nœud (x_1,y_1) voisin du nœud (x, y) qui

est choisi pour la diffusion, et *dist* étant le nombre de liens à traverser pour aller de la source à la destination :

– si $dist((x_0, y_0), (x, y)) \leq dist((x_0, y_0), (x_1, y_1))$, alors on supprime (x_1, y_1) de la liste des voisins du nœud (x, y) qui est censé recevoir le message.

Étant donnée que les cibles changent selon les données demandées, la position du nœud qui fait la requête n'est généralement pas au centre du réseau lorsque la diffusion est déclenchée. La topologie tore virtualisée permet de garantir que l'émetteur est au centre, en conséquence nous utiliserons dans la suite l'algorithme pour cette topologie.

6.1.2 Abstraction de l'algorithme

Ayant défini le calcul des voisins (section 6.1.1), nous décrivons dans cette partie une abstraction de l'algorithme de diffusion implémenté au niveau des routeurs. L'idée est que le message issu de chaque nœud suit une stratégie de multidiffusion vers un ou plusieurs nœuds voisins. Afin d'éviter les contentions sur les liens physiques (liés à l'entrée et à la sortie d'un nœud), des tampons (*buffers*) ont été utilisés permettant la gestion des messages.

Algorithm 1 Algorithme de diffusion

1: **for** chaque nœud dans le réseau logique **do**
2:     **while** tampon de sortie non vide **do**
3:         retirer le message du tampon
4:         envoyer le message au tampon d'entrée du nœud voisin
5:     **end while**
6:     **while** tampon en entrée non vide **do**
7:         retirer le message du tampon
8:         calculer les adresses des nœuds logiques voisins auquel le message sera diffusé en utilisant l'algorithme pour la topologie tore
9:         multidiffusion du message aux tampons en sortie associés aux nœuds voisins
10:     **end while**
11: **end for**

6.2 Stratégie de multidiffusion

6.2.1 Micro architecture du routeur

Généralement, un routeur sert à aiguiller les paquets provenant d'un port en entrée vers le port approprié en sortie. Sa micro-architecture repose notamment sur des tampons permettant la mémorisations en entrée, des unités de contrôle accordant la priorité en cas de plusieurs demandes d'une même ressource et des unités de commutation (commutateur ou *switch*), afin de calculer le chemin de données et d'allouer les ressources nécessaires [DT03]. L'unité de commutation n'est qu'un ensemble de multiplexeurs déterminant un nombre fini de chemins réalisables au sein d'un routeur.

L'architecture de base de notre routeur est présentée figure 6.1. Elle englobe des ports d'entrées sortie, des mémoires tampons (VCs), une unité de routage, des unités de contrôles (*VC/SW Allocator*) et une unité de multiplexage (*Crossbar*). La structure des routeurs est généralement pipelinée et les étages sont au nombre de 5 : calcul du chemin de routage (RC), allocation du canal virtuel (VA), allocation du commutateur (SA), la traversé du commutateur (ST) et la traversé de la liaison (LT)

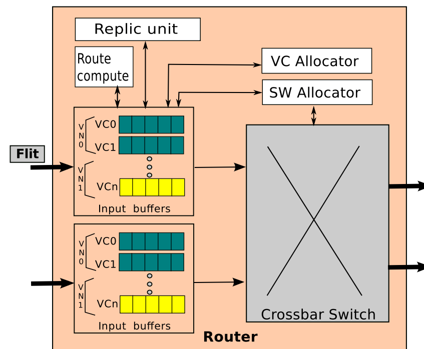


FIGURE 6.1 – Architecture d'un routeur

Le mécanisme de commutation utilisé pour ce routeur est la commutation de paquets de type « trou de ver » (*whormhole*). Ce dernier est caractérisé par un transfert des paquets sans réservation préalable du chemin. En outre, dès lors qu'un flit est reçu, il est transmis au prochain routeur si l'espace requis est libre. Ceci est assuré via le protocole de contrôle de flux utilisant des crédits d'émission. Cette stratégie d'allocation et d'arbitrage des tampons et des liaisons détermine la possibilité ou non d'emprunter une connexion lors du routage. Un signal "credit" du routeur en aval est émis vers le routeur en amont qui lui indiquant la quantité minimale de ressources de mémorisation disponibles.

Notre routeur doit supporter la diffusion (et la multidiffusion) donc l'ajout d'une unité de réplication est nécessaire (voir figure 6.1). Cette dernière est chargée de calculer les nœuds destinataires conformément à la position du routeur courant ; i.e les nœuds voisins éligibles à la diffusion et de construire autant de paquets que le nombre de destinataires. Les paquets subissant la multidiffusion sont formés d'un seul flit contenant uniquement l'information qu'ils sont une requête. Leur structure fera l'objet de la section 6.2.1.1.
Un paquet répliqué au sein d'un routeur est géré comme un paquet indépendant par les étages d'allocation qui suivent. En effet, chaque réplique est sauvegardée dans un canal virtuel du même port d'entrée, mais qui fait partie d'un réseau virtuel "VN" dédié à la diffusion. Ce réseau virtuel sera détaillé dans la section 6.2.1.2.

6.2.1.1 Structure du paquet de multidiffusion

Le paquet est formé d'un ou de plusieurs flits. Conformément à son type dans le protocole, une requête est représentée par un seul flit (en-tête et queue à la fois) alors qu'une réponse nécessite 5 flits (un en-tête, trois corps, un queue). Le flit d'en-tête d'un paquet contient les informations de routage destinées aux routeurs ainsi qu'à l'interface réseau destination. Il ouvre le chemin et réserve au même temps les ressources nécessaires à tout le paquet, les autres flits les suivent le long du chemin en séquence.

Pour la multidiffusion, le paquet de base est un paquet d'un flit de type "en-tête-queue" étant donné que réellement c'est de la multidiffusion d'une requête qu'il s'agit. Par ailleurs, pour le distinguer d'un simple paquet un champ d'un bit est ajouté. Mis à 1, il s'agit d'une requête de multidiffusion, mis à 0, il indique une requête unique. C'est au niveau du protocole de cohérence que ce bit est activé. D'autre part, comme nous l'avons évoqué dans le chapitre précédent, la solution de virtualisation du réseau que nous avons adoptée nécessite l'adaptation du format du paquet à ce réseau. En effet, un

en-tête est ajouté au paquet contenant l'adresse du nœud destinataire dans le réseau virtuel et un bit indique que le paquet contient un en-tête temporaire.
La figure 6.2 présente le codage des informations de routage pour ce type de paquet :

- **Flit original** : il contient les champs de base d'un flit d'en-tête ; un champ "V" qui indique que le mot présent sur le lien est valide, un champ type (2 bits) détermine la position du flit dans un paquet (head, body, tail et head-tail), un champ "VCID" traduit le canal virtuel sur lequel le paquet est transporté, un champ "srcID" contenant les cordonnées du routeur initiateur du paquet, un champ "Dest" qui utilise un codage binaire pour sauvegarder les coordonnées du destinataire. Nous ajoutons donc un champ "Multicast_bit" révélant que c'est un paquet de multidiffusion.
- **En-tête ajouté** : un champ "Dest_log" contenant l'adresse physique du destinataire logique et un bit "A_H" reflétant que ce flit n'est pas l'original.

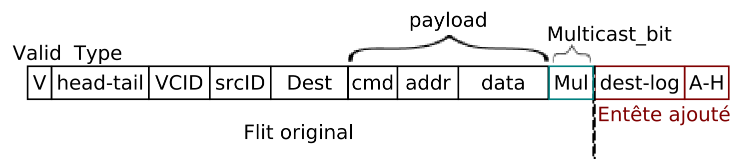


FIGURE 6.2 – Format du paquet de multidiffusion

6.2.1.2 Canaux virtuels et politique d'arbitrage

Notre réseau sur puce supporte des canaux virtuels (VCs) ayant comme charge la répartition d'un canal physique. Leurs avantages est qu'ils évitent les interblocages, optimisent l'utilisation des liens physiques et fournissent des services séparés. Par conséquent, le surcoût induit par leur utilisation est compensé par les avantages qu'ils apportent.

Ces canaux sont organisés en groupe (quatre VCs) et chaque groupe représente un réseau virtuel dédié à un type donné de paquet. Trois types existent déjà dans le protocole (request-response-unblock) auxquels nous ajoutons un quatrième (multidiffusion) qui n'est réellement qu'une sorte de requête. La raison de cet ajout est qu'il permet de séparer les trafics, de réduire les contentions sur les ressources tout en donnant l'illusion que le multidiffusion se produit dans un réseau séparé. Ainsi, chaque réseau virtuel dispose d'un ordre de priorité qui correspond à son numéro. Le premier est le plus prioritaire et dans notre situation le réseau virtuel supportant la multidiffusion est le plus prioritaire. De ce fait, le trafic de classe multidiffusion est privilégié sur les autres trafics qui évoluent sur le réseau.

La solution que nous avons adoptée pour que notre routeur supporte la multidiffusion est simple à implanter. Une fois le calcul de destinataire fait et les paquets répliqués formés, ces derniers sont sauvegardés dans les canaux du réseau virtuel dédié. Le nombre de répliques ne peut dépasser en aucun cas 4 puisqu'un routeur peut avoir au plus 4 voisins. À chaque canal virtuel est associé un numéro égal à son ordre de priorité. Selon le routeur, il existe deux manières d'allouer les canaux :

- Le routeur qui initie la multidiffusion se sert du paquet original uniquement pour construire les répliques, ensuite il le détruit. En effet, si nous gardons le paquet, il sera envoyé au port local pour consulter le banc de cache L2 en demandant le bloc, or ce dernier a été déjà consulté. Les paquets ainsi formés seront sauvegardés dans les canaux disponibles en respectant leur priorité.
- Les autres routeurs gardent le paquet original qui sera accueilli par le canal virtuel dont l'identifiant est défini dans le champ "VCID" du paquet. Les répliques occuperont les canaux libres qui suivent.

L'algorithme 2 détaille le processus de construction des paquets à diffuser au niveau du port d'entrée selon le nœud, qu'il soit un déclencheur, un voisin logique ou un nœud intermédiaire.

Algorithm 2 Algorithme de construction des paquets selon le nœud

```
 1: if (flit est de type entête-queue) or (flit de type entête) then
 2:     if (Multicast_bit==1) and (A_H)==0 then          ▷ nœud déclencheur du multidiffusion
 3:         Vérifier la disponibilité des 4 VCs
 4:         for i = 0 → 3 do
 5:             Répliquer le flit original
 6:             Insérer le flit dans le VC[i]
 7:         end for
 8:     else if (A_H==1) and (destination logique atteinte) then          ▷ nœud d'un voisin logique
 9:         nb_logiques = nombre de prochains voisins logiques
10:         Vérifier la disponibilité des nb_logiques VCs
11:         for i = 0 → nb_logiques do
12:             Répliquer le flit original
13:             Insérer le flit dans le VC[i]
14:         end for
15:     else          ▷ nœud intermédiaire
16:         Sans réplication insérer le flit dans le VC[i] qui lui est associé
17:     end if
18: end if
```

À noter qu'évidemment un paquet ne peut être consommé par l'unité d'entrée que si le canal virtuel qui doit l'accueillir est libre. Dans la situation de multidiffusion, avoir autant de canaux virtuels libres que le nombre de paquet à former est une exigence qui devra être remplie. Autrement, le paquet ne sera plus consommé et il reste bloqué jusqu'à la satisfaction de la condition.

Pour les canaux virtuels d'un même réseau virtuel la politique d'arbitrage adoptée est celle de la priorité tournante (tourniquet ou *round-robin*) qui permet d'éviter les famines. Avoir la priorité la plus élevée n'est qu'une condition nécessaire mais non suffisante pour obtenir un accès au lien. En effet, pour qu'un canal soit candidat à l'accès à un port de sortie, il doit avoir au moins un flit prêt à être envoyé et il doit disposer des crédits pour l'émission. Une fois qu'un canal a obtenu l'accès au port, les arbitres des canaux moins prioritaires sont gelés.

6.2.2 Contrôleur de l'interface réseau

La multidiffusion ne peut pas être élaborée isolément des contrôleurs associés aux unités de traitements et plus précisément des interfaces réseaux. En fait, cette interface réseau est nécessaire pour adapter les unités de traitement au réseau. Elle est chargée de la gestion de flux de paquets dans deux sens :

– **Processeur → Réseau**
Le processus construit un paquet de flits à partir des données envoyées par le processeur et les envoie au port local du routeur auquel l'interface est attachée. L'adresse de destination est présente au niveau du paquet, l'interface y ajoute juste un champ "TimeStamp". Ce champ est un artéfact du modèle de simulation nécessaire à la synchronisation des évènements au niveau du réseau. Comme décrit dans [TMG+05], c'est un concept de modélisation du trafic définissant l'instant d'injection du paquet dans le réseau. Les paquets destinés au réseau sont sauvegardés dans des files d'attente "inNode_buf" selon l'identifiant du réseau à traverser (VN). Ce dernier correspond aux types du message dans le protocole (resquest, response, ..). Ensuite, un module intitulé "packetizer" décompose le message en flits, les stocke dans un canal virtuel libre associé au VN auquel appartient le message (voir figure 6.3). Le calcul de l'identifiant du canal virtuel se fait en se basant sur un contrôle de flux basique "crédit d'émission" et les flits seront envoyés dès que suffisamment de crédits auront été reçu.

– **Réseau → Processeur**
Le processus assemble et ordonnance les flits en provenance d'un routeur et les transmet au processeur et/ou la mémoire. Tout comme précédemment, les flits venant du réseau sont mis dans les files d'attente adéquates. Le module "depacketizer" assure le processus inverse : il extrait l'en-tête et rassemble les flits pour former le message et le stocke dans l'une des files d'attente en sortie "outNode_buf". Dans la situation où deux paquets arrivent en même temps et qu'ils sont destinés à une même mémoire, un arbitrage de type tourniquet résout le problème.

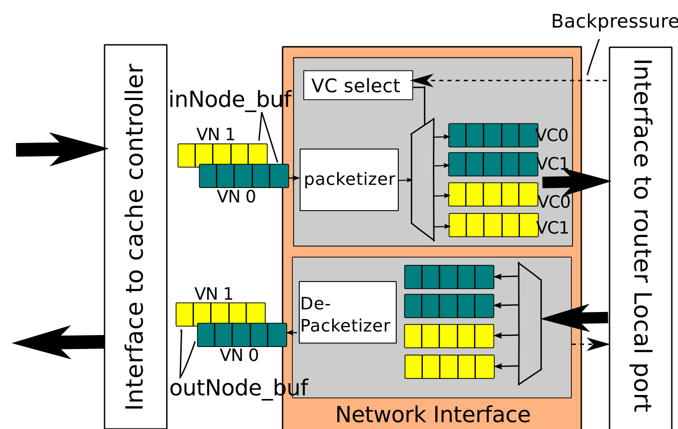


Figure 6.3 – Architecture de l'interface réseau

L'interaction de ce module avec les routeurs a le mérite de contrôler le flux lors de la construction des paquets de multidiffusion et particulièrement lors d'un blocage dû à l'indisponibilité des canaux virtuels. Donc, nous mettons l'accent sur le mécanisme de contrôle de flux implémenté pour garantir le transfert des flits.

Réellement, l'interface réseau est reliée à l'entrée du routeur via un canal de communication qui n'est autre qu'un lien physique. Ce canal influe différemment sur l'interface réseau et tout dépend de son architecture. Constitué de deux tampons comme indiqué

sur la figure 6.4, le premier à l'entrée du lien accueille les flits en provenance de l'interface réseau et le second ordonnance ces flits avant leur transmission à l'entrée du routeur. Le contrôle de flux au niveau de l'interface réseau informe de la disponibilité des crédits au niveau du premier tampon donc la transmission d'un flit à ce niveau n'implique aucun blocage. Par ailleurs, le phénomène de contention aura sûrement lieu au niveau du deuxième tampon lors de la multidiffusion. À cet effet, nous nous intéressons au contrôle de flux au niveau des liens.

6.2.3 Contrôle de flux au niveau des liens

Dans notre architecture nous utilisons une paire de liens unidirectionnels en sens opposés. C'est une méthode classique permettant d'éviter le contrôle d'accès par jetons lors de l'accès à un lien bidirectionnel [LYB96]. Le contrôle de flux au niveau lien (entre les interfaces et les routeurs ainsi qu'entre les routeurs) se fait également par crédit d'émission. Un compteur, associé à chaque canal virtuel, indique le nombre de places disponibles dans le tampon du canal au niveau du routeur en aval. Initialement, il contient la valeur correspondante à la profondeur du tampon. Chaque fois qu'un flit est envoyé, cette valeur est décrémenté de 1 et elle est incrémentée de 1 lors de la réception d'un crédit. L'intérêt d'une telle technique est de pipeliner l'envoi des flits sur le lien.
Le trafic de multidiffusion ne nécessite pas de contrôle de flux particulier, car un paquet de multidiffusion est géré comme un paquet normal. Lors de la réplication des paquets, une condition nécessaire est d'avoir autant de canaux virtuels libres que le nombre de répliques. Le cas échéant, le flit au niveau du lien (Link_buffer) ne sera pas consommé (voir figure 6.4). Et il se met en attente jusqu'à son réveil par l'ordonnanceur de l'interface réseau qui lui est associé.

Pour mieux assimiler le problème souligné précédemment, nous nous appuyons sur le trafic classique allant de l'interface réseau à l'entrée du routeur. En fait, à la réception d'un flit d'une interface réseau "src_queue", le flit sera stocké au niveau d'une file d'évènements "link_buffer" en attente de sa consommation par le module en aval (l'entrée du routeur). Le modèle du lien dans Gem5 nécessite l'ajout d'un délai dans le champ "TimeStamp" du flit afin d'indiquer l'instant auquel il sera prêt à être consommé. La valeur de ce délai est un cycle. L'ordonnanceur associé à cette file réveille cette entrée un cycle plus tard. Ainsi, le flit est transmis à l'entrée du routeur s'il est prêt à cet instant et que le canal à son entrée est libre.

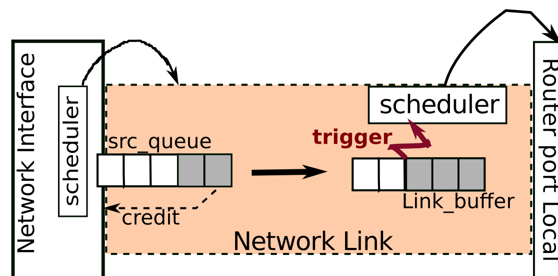


FIGURE 6.4 – Architecture des liens

Vu la manière dont le trafic de multidiffusion est géré, la file d'évènements est de plus en plus remplie au cours du temps et les flits pourraient finir par être bloqués

à ce niveau. Pour éviter une telle situation, un auto-ordonnancement a été ajouté au niveau du lien, étant donné que le nombre de flits répliqués dépasse nécessairement les messages du protocole (déclencheurs de l'ordonnanceur) envoyés à l'interface réseau. Ce mécanisme consiste à faire réveiller le port local par l'ordonnanceur du lien tant que le "link_buffer" contient plus qu'un flit. Donc à chaque cycle d'horloge, le port d'entrée vérifie :

– Le fait que le paquet soit prêt, i.e la latence dans son champ "TimeStamp" a été atteinte (supérieur ou égale au cycle courant).
– La disponibilité des canaux virtuels nécessaires (selon le VCID du flit à consommer et le nombre de répliques).

Une fois ces conditions remplies, le flit est transféré au routeur et il subit la duplication selon les destinataires. Les flits ainsi formés occupent chacun le canal virtuel dédié et ils seront routés de manière identique aux paquets du trafic normal.

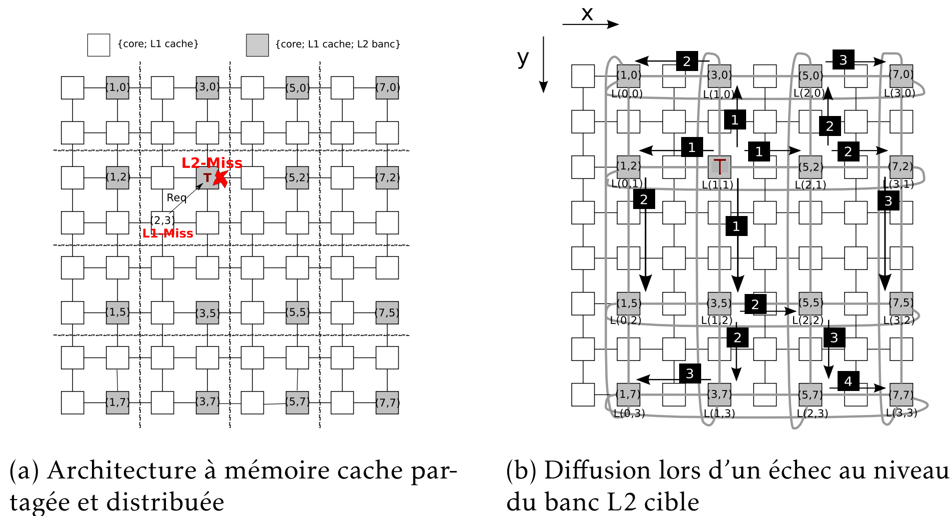
6.3 Exemple de recherche de données

Nous introduisons dans cette section un exemple qui détaille la diffusion et à travers lequel nous concrétisons notre solution. Nous utilisons comme base une architecture homogène à mémoire cache partagée. Chaque banc de cache L2 (cible) est associé à 4 cœurs (cf. figure 6.5a). Le scénario est comme suit : Suite à un échec dans son cache L1, le cœur associé au routeur (2,3) envoie une requête à sa cible L2 située au niveau du nœud (3,2) en lui demandant la donnée. Ce banc, n'ayant pas la donnée parmi ses lignes, annonce l'échec et la nécessité de consulter le reste des bancs de cache L2 via une diffusion. Elle est élaborée sur la base d'un réseau logique qui relie les bancs L2. Elle est mise en place comme l'illustre la figure 6.5b avec une topologie de type tore afin d'utiliser l'algorithme de diffusion décrit dans la section 6.1.1. Nous rappelons que la construction du réseau logique a eu lieu au moment de la configuration et que cela requiert le stockage des adresses physiques associées aux voisins logiques par routeur. Par exemple, le routeur (1,0) sauvegarde dans ses registres Est, Ouest, Nord et Sud les coordonnées (3,0), (7,0), (1,7) et (1,2) respectivement.

Lorsque le contrôleur d'interface réseau associé au routeur (T) de coordonnées (3,2) déclenche un message de multidiffusion, un bit est activé au niveau du paquet à injecter reflétant la requête. À l'entrée du port local, ce bit est vérifié et s'il est à 1, quatre paquets ayant comme destination les adresses des voisins Est, Ouest, Nord et Sud dans le réseau logique (L(2,1), L(0,1), L(1,0) et L(1,2)) seront construits. Conformément à ce qui a été évoqué précédemment, l'accès aux registres définis par direction permet de récupérer les adresses physiques de ces voisins logiques. Ces informations sont nécessaires pour pouvoir acheminer les paquets entre les nœuds du réseau logique. Ainsi, chaque paquet se voit greffé d'un en-tête supplémentaire contenant l'adresse physique et un bit indiquant que ce paquet est étendu.

Dans l'exemple, les paquets formés auront la même structure et seulement les champs "VCID" et "Dest" diffèrent (voir figure 6.6). Donc, à chaque paquet est ajouté l'en-tête correspondante : les en-têtes pour les paquets traversant les ports Est, Ouest, Nord et Sud contiennent respectivement les adresses physiques (5,2), (1,2), (3,0) et (3,5).

Une fois ces paquets construits et stockés dans les canaux virtuels adéquats, le paquet d'origine est détruit étant donné que le banc de cache L2 associé à ce routeur est le



(a) Architecture à mémoire cache partagée et distribuée

(b) Diffusion lors d'un échec au niveau du banc L2 cible

Figure 6.5 – Exemple de recherche de données

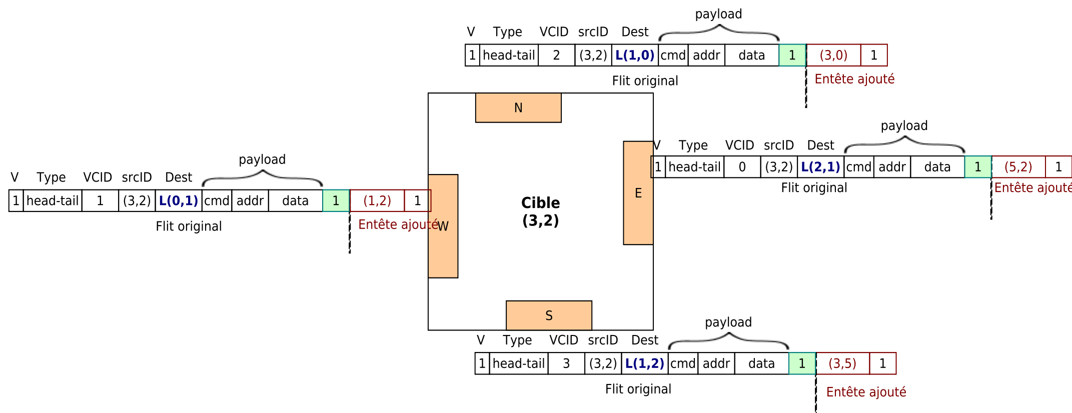


Figure 6.6 – Les paquets générés pour chaque direction

déclencheur de la multidiffusion [1]. Dans la suite, nous nous concentrons sur une branche de la diffusion, celle de la direction Est de la cible.

Le paquet ainsi formé traverse les différents étages du pipeline des routeurs (3,2) et (4,2) (*Step1* et *Step2* figure 6.7). Dès qu'il atteint le port d'entrée Ouest du routeur (5,2), l'en-tête est supprimé. Le paquet atteint sa destination logique et il sera dirigé vers le port de sortie local pour consulter le banc L2. Par ailleurs, vu que ce nœud ne représente pas le nœud le plus distant de la source (i.e. la diffusion n'est pas finie), le paquet sera répliqué. Le nombre de répliques est égal au nombre de voisins selon l'algorithme. À ce niveau, deux paquets seront construits en plus de l'original. Le premier destiné au routeur de cordonnées L(2,0) et le deuxième au routeur de cordonnées L(3,1). Le processus englobe ainsi l'envoi du paquet original au port de sortie local et l'ajout des en-têtes à chaque réplique pour assurer leur transmission aux nœuds logiques (*Step3*

1. Ce banc a été déjà consulté.

figure 6.7).

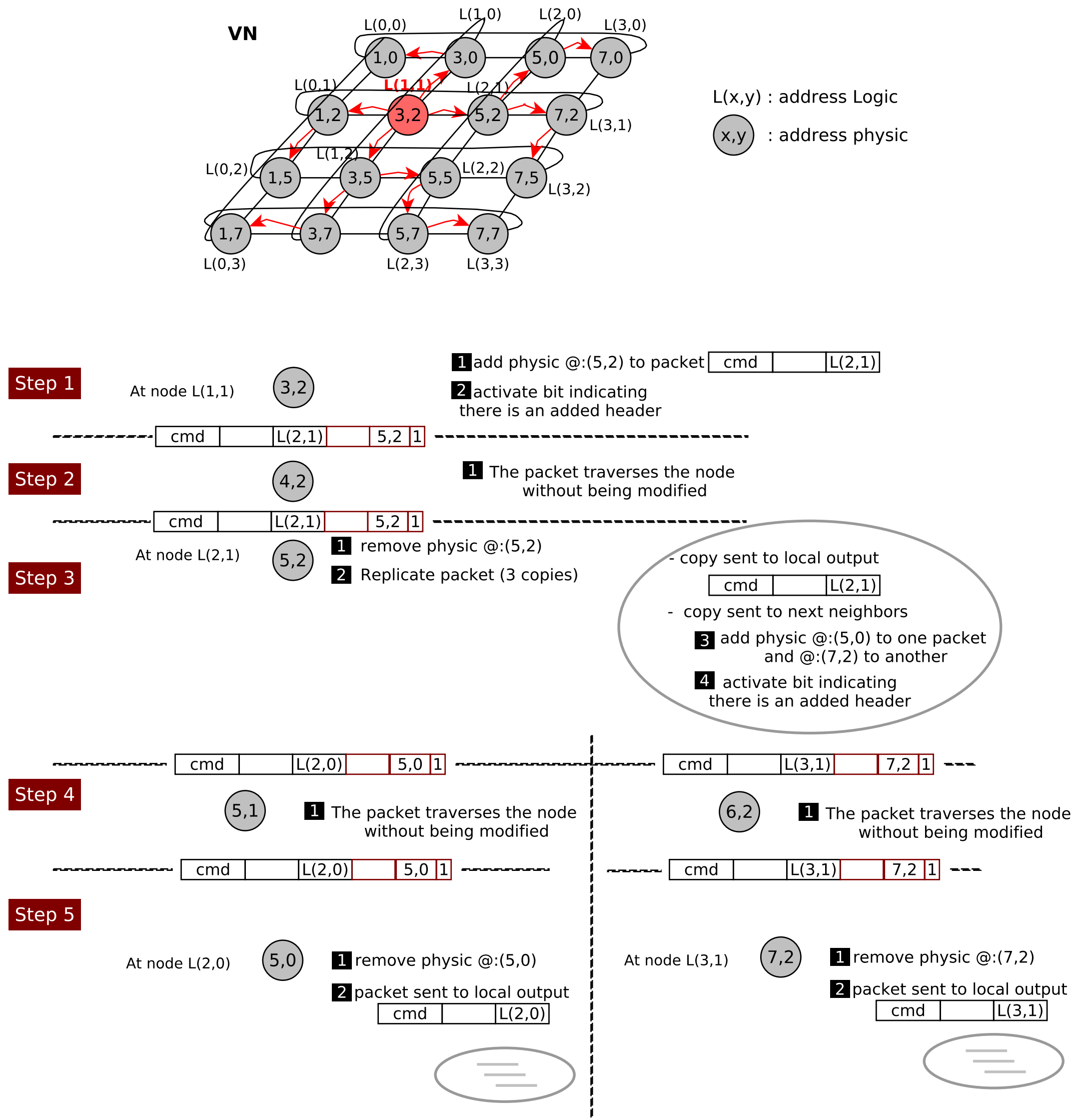


FIGURE 6.7 – Les bases du routage implémentant la diffusion

Par la suite, l'explication de ce que subit une réplique reste valable pour l'autre. Ainsi, le paquet destiné au nœud de coordonnées (5,0) traverse le routeur (5,1) sans être modifié (*Step4* figure 6.7). Dès son arrivé à destination, le même processus de suppression d'en-tête, de réplication et d'ajout d'en-tête aura lieu (*Step5* figure 6.7). Les paquets poursuivent chacun leur direction en subissant au niveau de chaque routeur le même processus jusqu'à atteindre la fin du réseau virtuel (i.e. le nœud le plus distant dans ce réseau).

6.4 Conclusion

Dans ce chapitre nous avons présenté une solution architecturale innovante permettant de gérer les données distribuées au niveau d'une hiérarchie mémoire dans laquelle les bancs d'un niveau donné (hormis le L1) sont partagés et sans copies. Cette solution consiste en particulier à diffuser une requête (pour la demande d'une donnée) à tous les bancs d'un même niveau de cache et ce n'est en réalité qu'une sorte de multidiffusion. Certes, la multidiffusion nécessite la modification de la micro-architecture du routeur et des composants en interactions avec ce dernier (liens, interface réseau) où il devient fondamental de contrôler le flux des données. De plus, vu que la solution repose sur des réseaux virtualisés interconnectant les bancs d'un même niveau, l'algorithme de routage des paquets doit être modifié pour en tenir compte.

La hiérarchie mémoire et son protocole de cohérence est un exemple d'utilisation du concept de réseau logique, la base de notre approche. La facilité offerte par ce concept atténue le coût de la diffusion. Néanmoins, un point qui mériterait d'être améliorer est l'arrêt de la multidiffusion dès lors que la donnée est trouvée.

Chapitre 7: Expérimentations & Résultats

Sommaire

6.1 Modèle de diffusion **69**
6.1.1 Calcul des différents voisins 69
6.1.2 Abstraction de l'algorithme 71
6.2 Stratégie de multidiffusion **71**
6.2.1 Micro architecture du routeur 71
6.2.2 Contrôleur de l'interface réseau 74
6.2.3 Contrôle de flux au niveau des liens 76
6.3 Exemple de recherche de données **77**
6.4 Conclusion **79**

Dans ce chapitre nous allons évaluer expérimentalement l'efficacité des mécanismes proposés dans cette thèse, ainsi que leur passage à l'échelle. Nous commençons d'abord par une présentation de la plate-forme utilisée puis nous illustrons les résultats des différentes études qui ont été menées.

Les premières évaluations concernent la performance du réseau logique virtuel (VN). Ce concept répond à la problématique de gestion de la communication pour une architecture distribuée, tout en se contentant des ressources matérielles disponibles. Nous avons recours aux générateurs de trafics servant à l'exploration du réseau virtualisé.

Les secondes évaluations se focalisent sur la nouvelle organisation de la hiérarchie des mémoires caches permettant l'exploitation maximale de l'espace de mémorisation embarqué sur la puce. C'est un exemple qui tire profit du concept de virtualisation réseau. Ce concept est bien adapté au protocole de cohérence de caches dont l'enjeu est de réduire la latence moyenne des accès, d'être économe en bande passante et d'avoir un impact positif sur la consommation. Cette partie offre un aperçu plus réaliste de la performance de l'approche pour les applications fortement contraintes par les transferts de données.

7.1 Plateforme d'expérimentation : Gem5

Gem5 est un simulateur très utilisé par les équipes de recherche du domaine des architectures de processeurs superscalaires et des systèmes multiprocesseurs [BBB+11]. Il a été développé par des universités, principalement MIT, Michigan, Princeton, Texas et Wisconsin, et des entreprises telles que HP, AMD, ARM, MIPS. Il est utilisé pour explorer les caractéristiques des architectures multiprocesseurs, du réseau d'interconnexion aux

processeurs eux-même. Il supporte une large gamme d'ISAs tels que MIPS, ALPHA, ARM, X86, SPARC et PowerPC. Il englobe plusieurs niveau d'abstraction pour les modèles de CPU (AtomicSimple, TimingSimple, In-Order and Out-Of-Order) et possède deux modes d'exécution. Un mode d'émulation d'appel système (SE) et un deuxième mode d'émulation complète de système (FS). Le premier exécute un binaire compilé statiquement, ce qui entraîne une simulation rapide, tandis que le deuxième charge une image complète du noyau et du disque pour l'exécution des applications. Ce dernier mode est le mode précis, il supporte les programmes multi-thread (les pthreads ne sont pas pris en charge en mode SE) mais en contrepartie, le temps de simulation est très élevé.

Ce simulateur offre deux modèles de mémoire. Le premier est un modèle « classique » simule une hiérarchie à plusieurs niveaux de cache avec un protocole de cohérence de type espionnage (*snooping*). Il fournit un système de mémoire rapide et facilement configurable. Le second est le modèle « Ruby » [1] dans lequel les contrôleurs de mémoires caches et les processeurs sont connectés via un réseau d'interconnexion. Ce modèle utilise un simulateur de NoC nommé Garnet [AKPJ09] qui s'intègre dans Gem5 et permet de modéliser le réseau d'interconnexion, assurant la communication entre les différents modules mémoire via des files d'attente comme l'illustre la figure 7.1. En outre, il fournit une multitude d'options permettant de personnaliser la micro-architecture des routeurs ainsi que l'algorithme de routage. Ce modèle de mémoire met l'accent sur la précision et fournit un support pour différents protocoles de cohérence de caches.

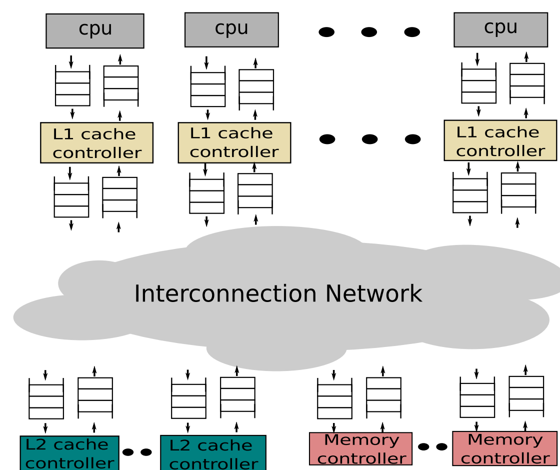


FIGURE 7.1 – Support de communication autour du système mémoire

Python, intégré au niveau du simulateur, est utilisé pour toutes les configurations. Cela fournit une interface souple pour la construction du système et permet d'adapter différents aspects du système à nos besoins.

7.2 Expérimentations I : Réseau virtualisé

Dans cette partie nous nous intéressons à l'exploration du réseau d'interconnexion sous sa nouvelle architecture. Plus précisément, nous nous focalisons sur les résultats

1. Rien à voir avec le langage de script Ruby.

de l'algorithme de routage lors de la création des réseaux logiques virtuels.

7.2.1 Garnet en mode autonome

Garnet, exécuté en mode autonome (*standalone*) permet de simuler le fonctionnement du réseau d'interconnexion et d'aboutir à des résultats précis au cycle près. La modélisation de ce réseau est basée sur quatre éléments : une interface réseau, un routeur, un ensemble de liens et un module de génération/consommation de paquets (processeur/mémoire).

- Le composant interface réseau connecté d'un côté à un routeur et de l'autre à un processeur ou à une mémoire assure l'assemblage des flits reçus depuis le routeur et la construction des flits à partir du message envoyé par le processeur.
- Le routeur fournit par Garnet est un routeur ayant plusieurs paramètres : nombre de canaux virtuels, taille des tampons, nombre de ports, etc.
- L'ensemble des liens entre les routeurs définit la topologie du réseau.
- Le module de génération de paquets, paramétrable, est modélisé comme un générateur de trafic (émulateur d'un processeur) et comme un réservoir de paquets (émulateur d'une mémoire).

Le flot de simulation est représenté dans la figure 7.2. Le fichier de configuration est une entrée du simulateur. Parmi les paramètres nécessaires on retrouve :
Les paramètres du NoC : topologie et taille du NoC
Les paramètres du routeur : nombre de ports, nombre de canaux virtuels, taille des tampons d'entrée, politique d'arbitrage et d'allocation de canal, algorithme de routage, métrique de congestion.
Les paramètres de simulation : type de trafic généré, période d'injection de paquets dans le réseau (intervalle de temps entre deux injections de paquets), nombre de paquets injectés et reçus durant la simulation.

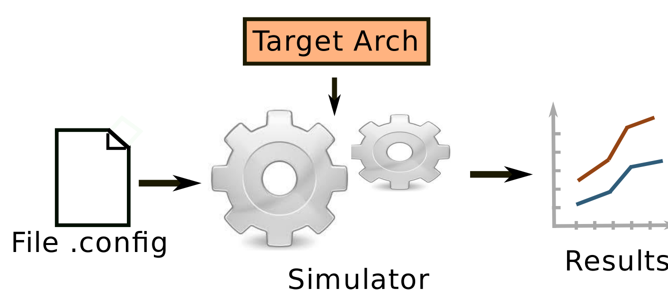


Figure 7.2 – Flot de simulation

Notre objectif est d'étudier la performance du réseau virtualisé en termes de latence. En effet, nous supposons que le réseau est formé de plusieurs réseaux logiques ayant chacun une topologie donnée. L'algorithme de routage basique supporté par ces derniers est le X-first qui a été adapté afin d'assurer l'acheminement des paquets au niveau des réseaux logiques. Nous avons associé à chaque nœud d'un réseau logique donné un type de générateur pour séparer les différents trafics. Pour ce faire, il nous a fallu adapter le fichier de configuration.

7.2.2 Trafics synthétiques utilisés

Évidemment, le générateur de trafic est destiné à produire un flot de messages permettant d'avoir un retour sur les performances sous différentes conditions. Parmi les différents paramètres, celui qui influe le plus notablement sur un réseau à commutation de paquets est la longueur des paquets, i.e. le nombre de flits le constituant. Trois types de trafics ont été utilisés pour l'évaluation de la nouvelle architecture du réseau que nous avons proposée :

- **Le trafic aléatoire uniforme (*uniform random*)** est composé de paquets dont les destinations se trouvent réparties uniformément sur tout le réseau. Une destination est choisie au hasard avec une probabilité qui est égale à $\frac{1}{N-1}$, avec N le nombre de nœuds du réseau.
- **Le trafic localisé (*localized*)** est composé de paquets dont les destinations se trouvent réparties dans le voisinage de la source. Il suit une loi de probabilité définie par l'équation 7.1 :
$$P(d) = 1/(A(D)2^d) \tag{7.1}$$
où D est le diamètre du réseau et A(D)= $\sum_{d=1}^{D}(1/2^d)$ est un facteur normalisé qui garantit que la somme de toutes les probabilités est 1 [WGP+09]. Ce type a été choisi car il permet d'exprimer la localité.
- **Le trafic complément de bit (*bit complement*)** choisit des destinations situées dans le coté opposé du réseau par rapport à la source. Le calcul de l'adresse de destination est une inversion des bits des coordonnées de la source. L'intérêt de ce type est que le trafic est propagé sur tous les liens de la bissection garantissant une charge équilibrée du réseau [DT03].

Il existe une multitude de types de trafics synthétiques, or nous nous limitons aux types présentés précédemment pour l'unique et la simple raison est que ces derniers modélisent les principaux trafics que l'on peut trouver dans la majorité des applications [GK10].

Cette section a permis d'introduire l'environnement de simulation, y compris le générateur de trafic qui servira à l'exploration du réseau. Il reste à comparer les résultats de notre approche à l'architecture de base afin de caractériser sa performance.

7.2.3 Expérimentation I.1

La première expérimentation consiste à simuler un réseau 4x4 de topologie grille 2D. Sur ce réseau, deux réseaux logiques ont été construits comme le montre la figure 7.3. À chaque réseau (physique/logique) est associé un type de générateur de trafic. Le trafic localisé est injecté au niveau de tous les nœuds du réseau physique. De manière équivalente, les nœuds du premier réseau logique injectent du trafic de type uniforme aléatoire, et ceux du second du trafic de type complement de bit. De ce fait, le nombre de générateurs associés à un nœud donné dépend de son affiliation aux différents réseaux. En effet, un seul générateur est attribué aux nœuds appartenant uniquement au réseau physique, deux pour les nœuds appartenant à un seul réseau logique (en plus du réseau physique) et trois pour les nœuds faisant partie de deux réseaux logiques (plus le physique).

Le tableau 7.1 récapitule les paramètres de configuration de l'architecture à simuler.

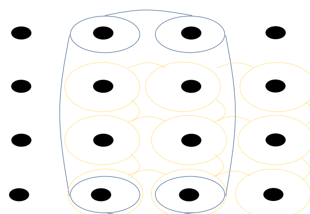


Figure 7.3 – Un réseau physique 4x4 sur lequel est sont projetés deux réseaux logiques de taille 2x2 et 3x3

Les nœuds sources commencent l'injection des paquets jusqu'à la fin de la simulation.

Tableau 7.1 – Configuration du simulateur

Topologie	Grille 2D
Politique d'arbitrage	Tourniquet
Taille des réseaux logiques	2x2, 3x3
Nombre des réseaux logiques (VNs)	3
Nombre de VCs	2/VNs
Taille des tampons	4 flits
Taille des paquets	16 flits
Durée de simulation(cycles)	10.000.000

Notre approche a été comparée à un réseau supportant une architecture qui ignore le concept de réseaux logiques. Ainsi, les trois types de trafics envoient les messages à travers la structure physique du réseau. Pour cela un seul VN a été utilisé. Ce dernier supporte 6 canaux virtuels pour mener une comparaison juste en nombre de ressources.

7.2.4 Résultats I.1 : Performance du réseau

Dans cette section nous présentons les résultats obtenus. La figure 7.4 présente les latences moyennes en fonction de la charge réseau. La courbe en bleu traduit la moyenne des latences selon le taux de trafic injecté par les trois générateurs dans le réseau physique. C'est le résultat de référence. La courbe en rouge est celle d'un réseau qui supporte le concept de réseaux logiques. En effet, l'information sur le réseau à traverser est inscrite au niveau de l'entête de chaque paquet généré, et pour la circulation des paquets au niveau de cette architecture de réseau, c'est le mécanisme de meta-routage décrit dans le chapitre 4 qui a été utilisé.

Comme indiqué, notre approche est plus performante avec une saturation qui se produit à une charge réseau *une fois et demi* plus élevée que celle de la version de référence. Ainsi, le temps de traversée d'un réseau physique supportant des réseaux logiques est rapide et cela s'explique par le fait que les trafics générés par les différents types de générateurs étaient séparés. Cette séparation implique l'absence de d'interférence qui pourrait induire la contention des ressources du réseau, et donc à la dégradation de sa performance. Enfin, le routage des paquets au niveau de cette architecture de réseau est réalisé grâce à un mécanisme spécifique. Ce dernier fait partie intégrante de la solution et en a renforcé l'efficacité.

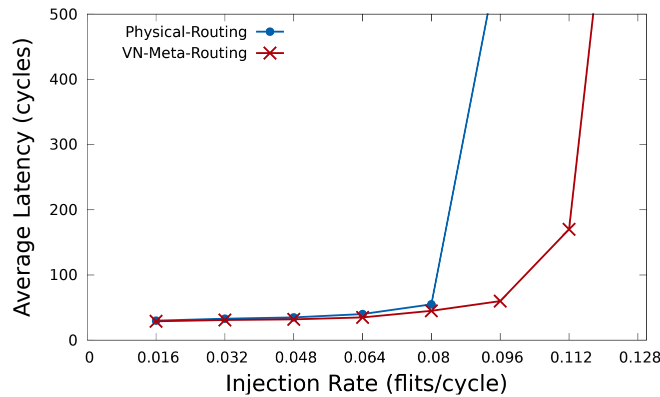


FIGURE 7.4 – Latence moyenne des paquets en utilisant des trafics synthétiques pour un réseau physique de taille 4x4

Au vu de ces résultats, nous pouvons affirmer que le concept des réseaux logiques projetés sur un réseau physique mène à de meilleures performances. Cependant, il faut s'assurer que ce résultat reste vrai lors du passage à l'échelle. La section suivante fait l'objet de la vérification de cette hypothèse.

7.2.5 Résultats I.2 : Passage à l'échelle

Notre but est d'étudier la scalabilité de notre approche. Nous simulons un réseau physique de taille 8x8 de topologie grille 2D sur lequel nous projetons deux réseaux logiques ; un premier de taille 4x4 et le deuxième de taille 5x5 (voir figure 7.5). Les paramètres du routeur conservent les mêmes valeurs que celles utilisées dans l'expérimentation précédente (tableau 7.1).

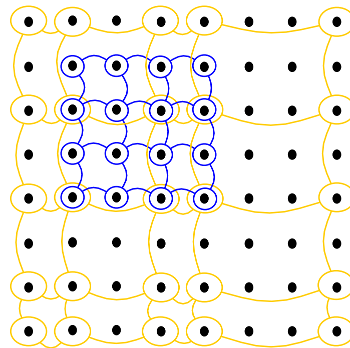


FIGURE 7.5 – Un réseau physique 8x8 sur lequel sont projetés deux réseaux logiques de taille 4x4 et 5x5

La figure 7.6 fait apparaître que les résultats de notre solution sont nettement meilleures que ceux de la version de référence. En fait, nous pouvons remarquer que la saturation cette fois se produit à une charge *2 fois et demi* plus élevée que celle du réseau ignorant le concept proposé. La raison de ce gain de performance est la répartition du trafic entraînant une limitation des contentions au niveau du réseau. En effet, pour un

taux d'injection de 0.096 (flits/cycle) le nombre de paquets injectés est égal à 420.631 pour la version à base d'un routage physique alors que pour notre approche il atteint 5.487.760.
Un point à remarquer sur cette figure est que lors de l'augmentation de la taille des réseaux les latences moyennes sont très élevées. Ceci est dû l'augmentation de la distance entre les nœuds (source-destination) parcourue par la majorité des paquets. Notre approche n'a aucun impact sur ces valeurs.

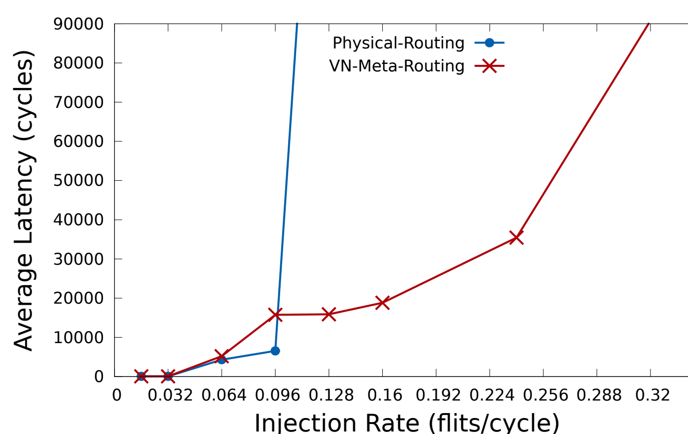


FIGURE 7.6 – Latence moyenne des paquets en utilisant les trafics synthétiques pour un réseau physique de taille 8x8

Nous tirons la conclusion que l'approche proposé passe à l'échelle en offrant de meilleurs performances en terme de latence. Néanmoins, la latence n'est pas l'unique critère à prendre en compte pour juger la performance. En fait, la consommation en terme de surface et d'énergie est un autre critère. C'est pourquoi la section suivante se charge d'estimer ces métriques.

7.2.6 Surface et puissance consommées

Cette section est consacrée à l'estimation de la consommation de puissance et de surface de notre réseau. Pour ce faire, nous utilisons DSENT [SCK+12], un outil de modélisation de la puissance statique et dynamique. Les paramètres technologique sont les suivants : transistors à basse tension de seuil (*low VT*) en CMOS bulk 45nm avec une fréquence de fonctionnement de 1 GHz. Le tableau 7.2 donne une estimation de la surface ainsi que la puissance consommées par un routeur du réseau supportant le concept des réseaux logiques comme présente la figure 7.5. Dans ce tableau, *1TG-router* correspond aux routeurs qui appartiennent uniquement au réseau physique, *2TG-router* est le routeur qui fait partie à la fois d'un réseau logique et du réseau physique, et *3TG-router* est celui associé à tous les réseaux (les deux logiques et le physique).
Les paramètres du routeur présentés dans le tableau 7.1 ont été conservés (taille des tampons, taille des paquets, ...).

En considérant le routeur associé à un seul générateur (1TG-router) comme référence, la surface consommée par les routeurs faisant partie d'un ou de deux réseaux logiques ne dépasse pas le **0.3 %**. En effet, notre approche a été élaborée avec les mêmes ressources que celles utilisées dans la version de référence (nombre et taille des tampons). Or,

Tableau 7.2 – Surface et puissance consommées par différents types de routeurs

Type	Area	Power
1TG-router	78342 μm^2	19.36 mW
2TG-router	78468 μm^2	19.41 mW
3TG-router	78594 μm^2	19.45 mW

comme il a été mentionné, l'algorithme de routage nécessite les adresses physiques des voisins logiques. Cette information doit être sauvegardée dans des registres (4 par réseau virtuels), mais la surface de ces derniers reste négligeable étant donné qu'au niveau d'un routeur la surface des tampons est très largement dominante. Ceci explique la faible augmentation en surface pour notre réseau supportant le nouveau concept.

Il en va de même pour la puissance consommée, nous remarquons une très faible augmentation due au processus d'ajout et de suppression d'entête (partie de l'algorithme de routage) qui requiert des lectures et/ou des écritures dans les registres. Les résultats obtenus reflètent que notre approche n'induit pas de surcoûts de surface et de consommation.

Synthèse Les résultats de cette partie sont très encourageants et montrent que supporter le concept de réseaux logiques permet d'obtenir de meilleurs performances, du moins en utilisant les trafics synthétiques. De plus, cette approche n'a pas d'impact visible sur la surface et la consommation. Il reste à confirmer ces résultats sur des systèmes complets, c.-à-d. des architectures formées de processeurs et des mémoires organisés autour d'un réseau sur puce, et avec des applications réelles. C'est l'objet de la section suivante.

7.3 Expérimentation II : Protocole de cohérence exploitant le concept de VN

7.3.1 Architecture simulée

Nous reprenons la plateforme de simulation présentée dans la section 7.1 et modélisons dessus une architecture NUCA. Le modèle mémoire est Ruby étant donné qu'il décrit précisément les interactions au niveau du réseau d'interconnexion et que ce modèle fournit une panoplie de protocoles de cohérence de caches. Le jeu d'instructions des processeurs utilisés est ALPHA.

Nous avons modélisé une architecture composée de 16 nœuds connectés à l'aide d'un réseau Garnet [AKPJ09] 4x4. Ce dernier, comme représenté sur la figure 7.7, est constitué réellement d'un ensemble de sous réseaux virtuels permettant de gérer les différents flots de communication. En effet, selon son type (Requête-Réponse-déblocage-multidiffusion), le message traverse le réseau virtuel dédié. Chaque réseau n'est en réalité qu'un ensemble de canaux virtuels de priorités différentes dans le réseau de base. Ceci permet d'éviter les interblocages et assure une communication efficace.

Comme évoqué précédemment, la hiérarchie mémoire est modulaire. Typiquement, le premier niveau de la hiérarchie est un cache séparé (instructions et données) associé à chaque processeur, le second niveau, unifié, est logiquement partagé mais physiquement

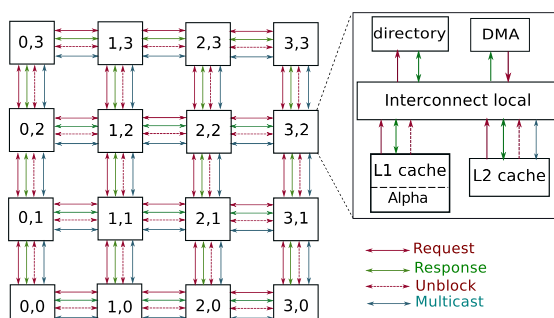


Figure 7.7 – Plateforme utilisée pour les simulations

distribué. Ayant comme intérêt l'étude des performances du cache L2, les caches de niveau 3 n'ont pas été ajoutés. La structure de la hiérarchie proposée élimine le besoin de cohérence des données au niveau des bancs de cache L2. Nonobstant, la cohérence des données au niveau des caches L1 reste nécessaire. Le protocole MESI classique a été choisi comme base et il a été modifié pour l'adapter au besoin de la nouvelle structure. Les détails de ces modifications ont été introduits dans le chapitre 2. Ce protocole est représenté par des contrôleurs décrits en langage SLICC (*Specification Language for Implementing Cache Coherence*) [Sli] pour Gem5.
Les éléments paramétrables des caches englobent la taille, l'associativité, la taille d'une ligne, les latences d'accès aux caches, la politique de remplacement. À cela s'ajoutent la politique d'écriture (*write through* ou *write back*) et la modélisation des tampons d'accès aux caches.
Le tableau 7.3 présente l'ensemble des paramètres de l'architecture modélisée.

Tableau 7.3 – Paramètres des architectures modélisées

Processors	16 cores, alpha ISA, running vanilla Linux SMP
L1 Caches	Split I/D, Private, 32KB, 2-way set associative, 64-Byte blocks, 1-cycle latency
L2 cache Banks number L2 cache bank	Distributed **Config 1** : 4 banks, 1 bank for 4 L1 cache (processor), 256KB/bank **Config 2** : 16 banks, 1 bank for 1 L1 cache, 1MB/bank Shared, 8-way set assoc, 2-cycle latency, 64 Bytes block, inclusive, Pseudo-LRU replacement
Directory Caches	Distributed shared by all cores, 6-cycle latency
Main Memory	512MB, 100-cycle access time
NoC	Tiled 4x4 2D Mesh network, X-Y routing, 64-bit flits and links, 1 cycle per hop, 1-flit control packets, 4-flit data packets, 4 VNETs with 3 VCs/VNET
Coherence Protocol	MESI protocol modified

Afin d'évaluer l'efficacité de notre stratégie nommé H_NUCA, nous l'avons comparé à d'autres stratégies d'architecture NUCA caractérisées par leur performances tels que S_NUCA et R_NUCA (une variante de NUCA dynamique). Dans la première stratégie, les données sont uniformément distribuées au niveau des bancs. En fait, le placement des données en mémoire est déterminé par les bits de poids faibles de l'adresse du bloc.

Les caches L1 et L2 sont inclusifs. Pour la deuxième stratégie, les blocs sont classés dynamiquement comme privés, partagés, ou en lecture seule. Ceci est effectué grâce à une modification au niveau de l'OS permettant le classement des accès. Ainsi, les données privées sont associées à un seul et unique banc de cache, les instructions sont entrelacés d'une manière rotationnelle dans un groupe de bancs de taille 4 et les données partagées sont entrelacées dans l'ensemble des bancs.

7.3.2 Applications utilisées

Dans cette partie, nous avons recours à des applications parallèles relevant du domaine du HPC (*High Performance Computing*). Ce sont les applications des benchmarks Splash-2. La raison est qu'elles sont dotées d'un degré de parallélisme variable, qu'elles sont fortement communicantes et que les schémas de communication en mémoire cache partagée entre les taches d'une même application sont différentes. Ces applications écrites en langage C, expriment le parallélisme de tâches via le standard des *pthreads*. La tableau 7.4 montre la configuration des entrées pour les différentes applications utilisés.

Tableau 7.4 – Paramètres des applications

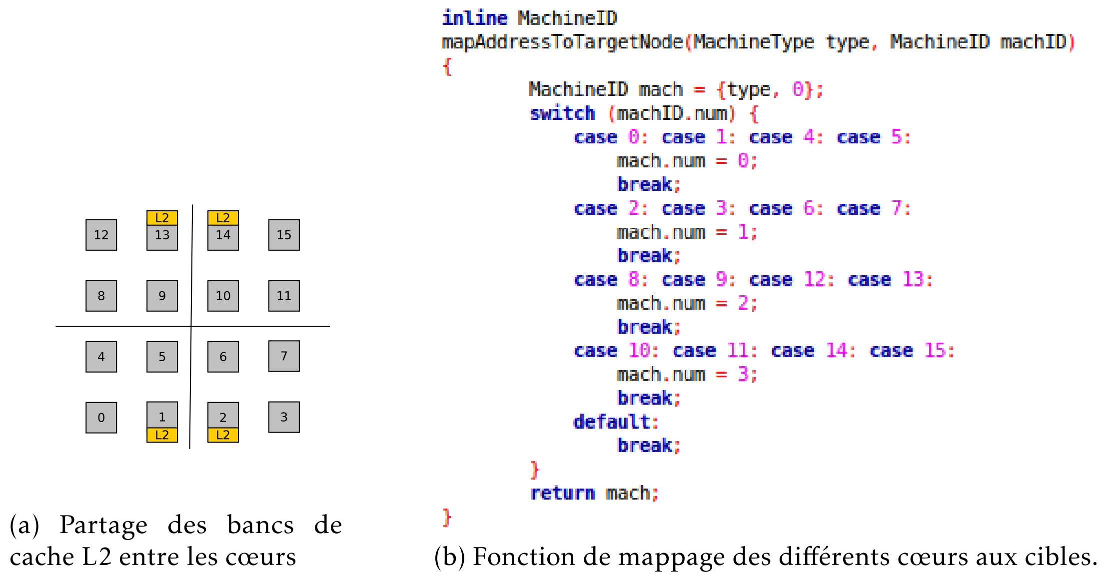
Application	**Données d'entrée**
FFT	-m 20
Radix	1M-keys
Lu	1024 x 1024
Fmm	32768 particules
Water-sp	512 molécules
Water-ns	512 molécules
Ocean	contiguous partitions, 258
Raytrace	teapot (256 x 256)

7.3.3 Résultat II.1 : Taux de défauts de cache

Cette partie a comme objectif d'évaluer les performances de la nouvelle structure de la hiérarchie de caches. Plus précisément, nous mettons l'accent sur la performance du cache L2 lors de la recherche des données. Le principal indice de performance est le taux de défauts de cache (*miss rate*). Cet indicateur représente le rapport entre le nombre de défauts de cache et le nombre total d'accès. En effet, notre hypothèse est que la nouvelle organisation du cache L2 réduit ce taux vu que l'espace de mémorisation réservé au cache L2 sera exploité au maximum.

En premier lieu nous simulons une puce 4x4 de 16 cœurs avec 4 bancs de cache L2. Chaque banc est partagé par 4 cœurs et il est défini au moment de la configuration comme la cible de ces derniers. La figure 7.8a présente cette configuration : les bancs de cache L2 attribués à la tuile 1, 2, 13 et 14 sont les cibles des ensembles de cœurs {0,1,4,5}, {2,3,6,7}, {8,9,12,13} et {10,11,14,15}, respectivement. Ainsi, une fonction traduit cette association dont le bout de code est détaillé dans la figure 7.8b.

Les résultats sont visibles sur la figure 7.9. On remarque immédiatement que l'hypothèse formulée précédemment n'est pas toujours valide : le taux est amélioré pour 7 applications parmi 8. L'amélioration est d'environ 20% et 11% par rapport à S_NUCA et



(a) Partage des bancs de cache L2 entre les cœurs

(b) Fonction de mappage des différents cœurs aux cibles.

Figure 7.8 – Configuration de l'architecture à simuler

R_NUCA, respectivement. Analysons dans la suite le comportement de l'application *Radix* dont le résultat s'est dégradé et essayons de comprendre la raison de l'augmentation de son taux d'échec.

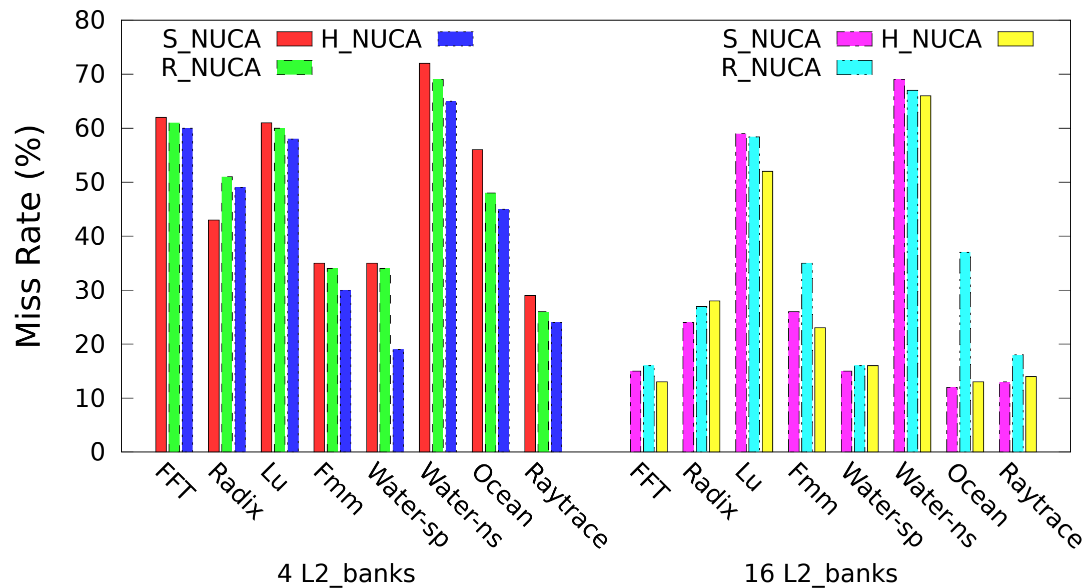


Figure 7.9 – Taux de défauts du cache L2 pour deux configurations différentes : une avec 4 bancs et l'autre avec 16 bancs

Étant donné que le niveau de cache L1 n'a pas été modifié, nous nous concentrons sur le niveau de cache L2 et nous mentionnons deux facteurs qui influent sur sa performance : le degré de partage et l'équilibrage de charge qui est fortement lié à la demande des

processeurs. Le premier facteur est pris en compte par notre organisation grâce à l'aspect « unicité » des blocs. Effectivement, une seule copie peut exister dans le second niveau de cache indépendamment du nombre de cœurs qui y ont accès. Pour le deuxième facteur, le cache L2 peut présenter un déséquilibre à l'utilisation des bancs et cela favorise l'apparition de deux types de défauts : défaut de capacité et défaut de conflits. Nous essayons d'examiner cet aspect dans *Radix* en mesurant le nombre de blocs évincés de tous les bancs de cache L2 durant l'exécution de l'application. La figure 7.10 donne ces mesures :

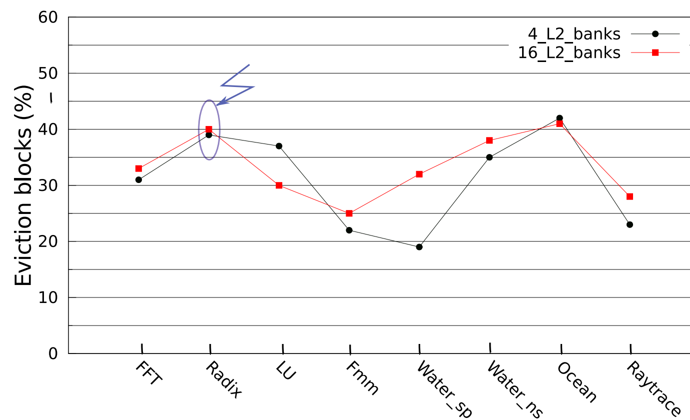
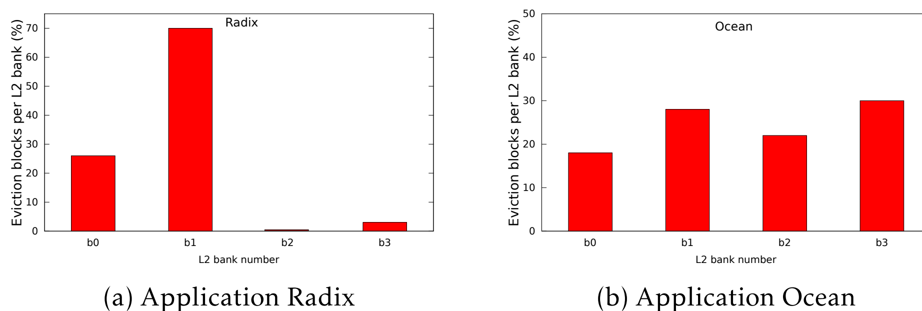


Figure 7.10 – Pourcentage des blocs évincés de l'ensemble des bancs du cache L2 pour différentes applications

À noter que le pourcentage de blocs évincés dans *Radix* est assez important et ceci est *a priori* la raison de la dégradation du taux d'échec. Néanmoins, pour l'application *Ocean*, bien que le pourcentage de blocs évincés soit élevé, le taux d'échec a été amélioré. Il convient donc d'observer les valeurs par banc de cache (voir figure 7.11). La différence est que dans *Ocean* l'évincement est équilibré pour les différents bancs (figure 7.11b) alors que pour *Radix*, il ne l'est pas (figure 7.11a). Il faudrait ainsi étudier la technique de remplacement dans certains cas et essayer de favoriser la mémorisation des données dans le voisinage afin de répartir la charge sur une zone plus large.



(a) Application Radix

(b) Application Ocean

Figure 7.11 – Pourcentage de blocs évincés par banc de cache L2

Par la suite nous avons procédé à la simulation d'un système sur puce de 16 cœurs

ayant un nombre des bancs de cache L2 qui est égale à 16. Cette fois, chaque cœur est associé à un seul banc L2 qui est sa cible. Le but est d'évaluer la scalabilité de la stratégie de multidiffusion. Plusieurs observations peuvent être soulignées (figure 7.9).

Par rapport à R_NUCA, le taux d'échec est réduit lors de l'utilisation de la stratégie S_NUCA, à l'exception des applications *LU* et *Water-ns*. L'amélioration s'étend de 9% à 74%. En fait, cette réduction est d'autant plus importante dans les applications *Fmm* et *Ocean*. Le profilage du comportement de ces dernières dévoile que les accès partagés et privés sont équilibrés. Donc S_NUCA est meilleure dans une telle situation.
En revanche, pour *LU* et *Water-ns*, la stratégie qui consiste à ranger les données privées dans un banc proche du demandeur telle que prônée par R_NUCA est une bonne solution, car ce type d'applications est caractérisé par de nombreux accès privés.

D'autre part, notre stratégie réduit le taux d'échec en moyenne à 24.4% par rapport à S_NUCA et 20.5% par rapport à R_NUCA. La comparaison à S_NUCA prouve à nouveau que le taux d'échec pour l'application *Radix* est élevé. Afin de mieux comprendre la cause de cette dégradation, nous mettons l'accent sur le pourcentage des blocs évincés. Nous observons que sa valeur est importante (40%). Ensuite, une analyse des pourcentages des blocs évincés par banc de cache a été menée et la figure 7.12 reporte ces valeurs. Notons que les bancs 2, 3 et 4 révèlent d'importantes activités. Cette situation peut survenir lors d'un placement d'un bloc récent mais qui est peu utilisé à la place d'un ancien bloc plus important. Le premier peut ne pas être référencé encore une fois plus tard, tandis que le deuxième sera demandé à nouveau prochainement. En raison de la forte variation des accès et à cause de la répartition des données au niveau des bancs, la technique de remplacement peut entraîner des pertes supplémentaires.

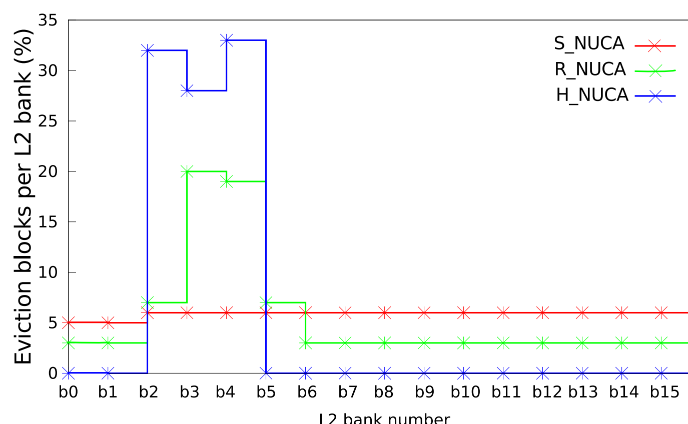


FIGURE 7.12 – Pourcentage de blocs évincés par bancs de cache L2 pour l'application *Radix*

En ce qui concerne R_NUCA, notre approche offre de meilleurs résultats (à l'exception de *Radix*). Sur *LU* et *Water-ns* le gain en taux d'échec est de 12% et 3%. En effet, la distribution des données comme l'impose notre architecture conduit à une utilisation équilibrée des bancs de cache L2. Ceci se confirme par les résultats obtenus et il est d'autant plus notable pour les applications *LU* et *Water-ns*. *A contrario*, le déséquilibre des bancs induit de la contention sur quelques uns et c'est le cas de *LU* et *Water-ns* avec R_NUCA.

L'application *Radix* avec notre approche n'est pas idéale. Les bancs n'ont pas la même

charge et il est clair, d'après la figure 7.12, que les bancs 2, 3 et 4 sont les plus consultés et cette fois le pourcentage des blocs évincés par notre approche est beaucoup plus élevé que celui du à R_NUCA.

La particularité de notre stratégie, qui repose sur l'accessibilité des processeurs à tous les bancs sans restriction, offre de bons résultats et ceci a été observé à travers les simulations précédentes pour une majorité d'applications. Pour les exceptions, il faut mettre l'accent sur l'algorithme de remplacement et mener une étude approfondie afin de cerner les facteurs de la dégradation. De plus, la distribution des blocs implique parfois une latence accrue due, d'une part, à la distance entre le demandeur et le banc hébergeant la donnée et d'autre part à la contention au niveau du réseau d'interconnexion. Par conséquent, il convient d'étudier l'effet de notre approche sur le réseau.

7.3.4 Résultat II.2 : Latence moyenne du réseau

La latence moyenne du réseau est exprimée comme fonction de la charge injectée à l'entrée du réseau durant toute l'exécution de l'application. Elle tient compte des caractéristiques statiques du réseau et du comportement dynamique sous régime conflictuel pour lesquels les flots se trouvent dans l'obligation de partager les mêmes ressources (tampons, ports de sortie,...).

La figure 7.13 compare la latence moyenne du réseau pour deux architectures ayant 16 cœurs, l'une avec 4 bancs de cache L2 et l'autre avec 16 bancs de cache L2. Nous remarquons que notre stratégie offre toujours de meilleures latences. Cet avantage est plus marqué dans la deuxième architecture. Pour la première architecture la latence est réduite, en moyenne de 21% par rapport à S_NUCA et de 10% par rapport à R_NUCA. Pour la deuxième, notre stratégie réalise un gain de 30.5% par rapport à S_NUCA et de 22.3% par rapport à R_NUCA.

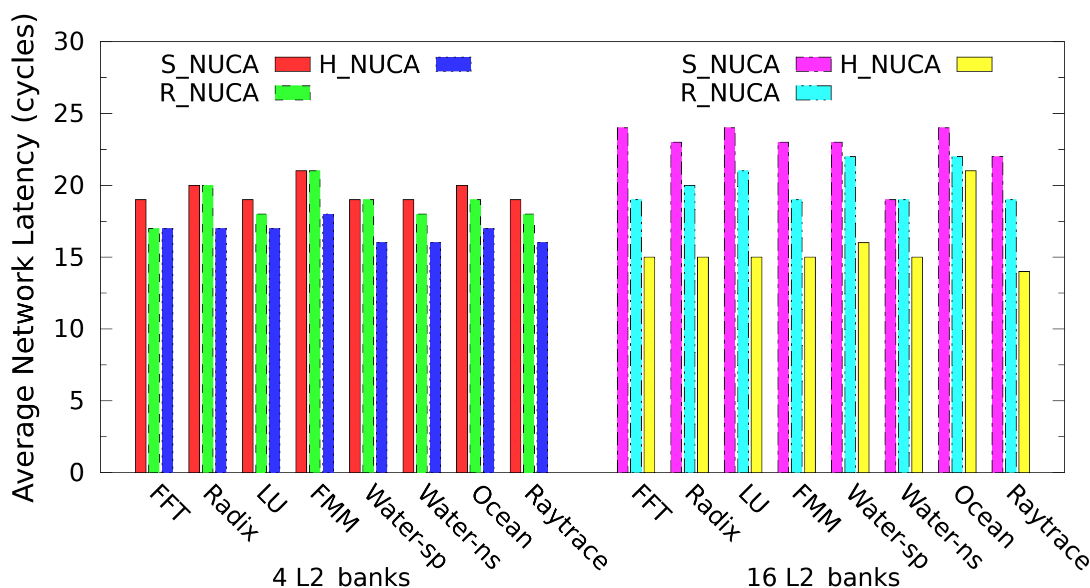


FIGURE 7.13 – Latence moyenne du réseau

Nonobstant le recours à un mécanisme de diffusion pour la recherche de la donnée lors d'un échec dans notre approche, la latence au niveau du réseau est largement améliorée. Deux facteurs contribuent à cette amélioration : (1) la façon dont nous organisons les données au niveau des bancs ; en priorisant le placement dans le banc cible, (2) l'utilisation du concept du réseau virtuel logique qui consiste à définir un réseau virtuel dédié aux transferts des paquets de la diffusion. Ce concept permet d'isoler le trafic (le trafic de la multidiffusion ne se mélange pas avec le trafic ordinaire) et ainsi de réduire la congestion des ressources qui a un impact important sur la latence. C'est le facteur dominant de la performance du réseau, en particulier dans la deuxième configuration (16 bancs de cache L2), car à chaque échec au niveau de la cible, la requête devrait être transmise à 15 bancs.

Toutefois, en s'appuyant sur le premier facteur, nous abordons la question de la distribution des blocs dans le cache de second niveau afin de connaître la raison de l'amélioration de la latence d'accès à ces blocs. L'un des points à souligner est que d'après ce qui précède, le nombre de défauts de cache L2 a diminué pour la plupart des applications, ce qui mène à alléger le trafic au niveau du réseau. Par la suite, nous classons les accès aux différents bancs du cache L2 comme banc cible (généralement c'est le banc local) ou éloigné. Comme le montre la figure 7.14, pour *LU* et *Ocean*, les pourcentages d'accès aux bancs éloignés sont égaux à ceux servis par les bancs locaux (système avec 16 bancs L2). Pour *Radix*, 62% des requêtes ont été servies par les bancs distants. En revanche, même dans des telles situations, la latence du réseau a été améliorée grâce au concept du réseau logique virtuel qui permet de mieux répartir le trafic et donc, de limiter la contention des files d'attente (FiFOs).

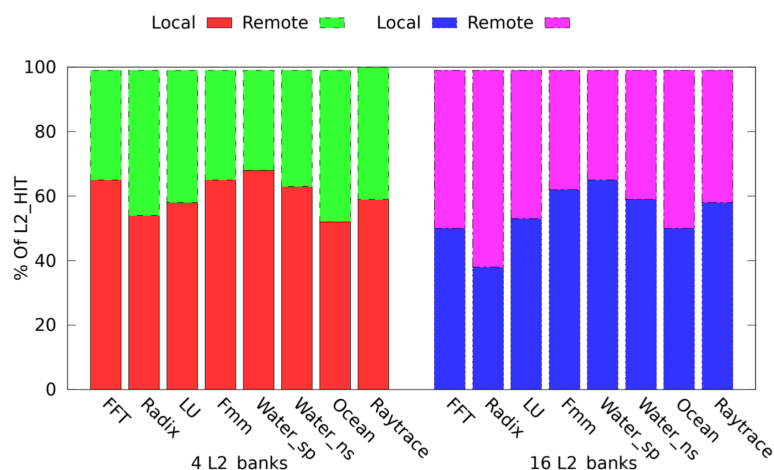


Figure 7.14 – Distribution des accès aux bancs de cache L2

Clairement, le trafic engendré par le mécanisme de diffusion peut encombrer le réseau et saturer les files d'attente, cependant la virtualisation du NoC semble fournir une meilleure performance de communication et réduire la latence moyenne.

7.3.5 Résultat II.3 : Temps d'exécution

Il est important de voir la performance de notre stratégie en terme de temps d'exécution. Nous utilisons les architectures simulées précédemment et nous montrons l'impact

de notre stratégie H_NUCA sur les différentes applications parallèles tout en comparant ses résultats à S_NUCA et R_NUCA.

La figure 7.15 présente l'accélération obtenue pour les différentes applications. Tous les résultats ont été normalisés par rapport à S_NUCA. Notre stratégie réalise en moyenne un gain de 7% par rapport à S_NUCA et 10% par rapport à R_NUCA.

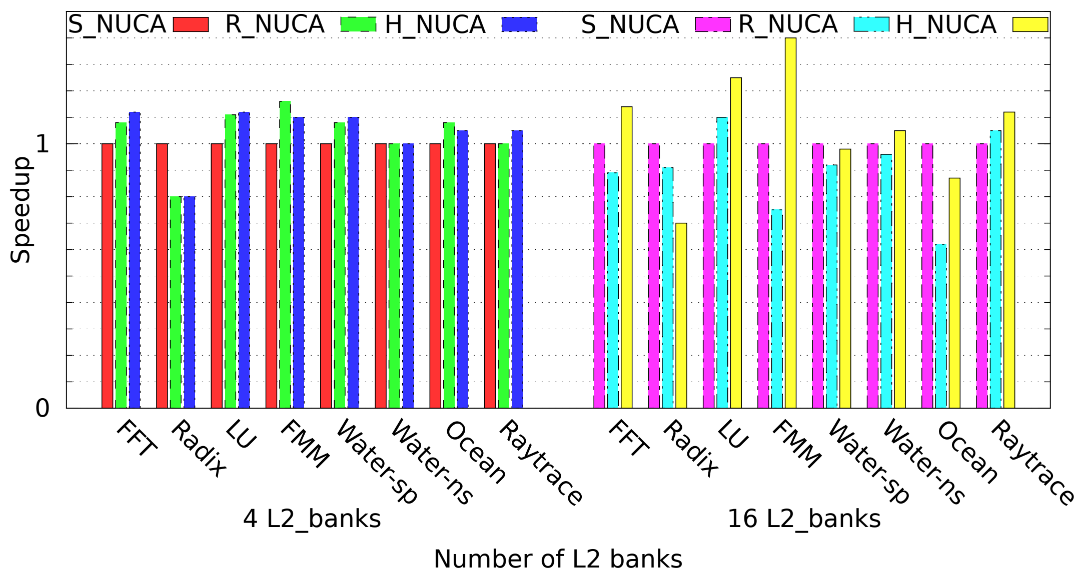


Figure 7.15 – Accélération pour les différentes applications

Pour la première configuration, notre approche est légèrement meilleure. L'accélération approche 8% par rapport à S_NUCA et 6.5% par rapport à R_NUCA. Les résultats de notre stratégie sont meilleurs pour 7 applications sur 8 vis-à-vis de S_NUCA et pour 6 applications pour R_NUCA. Dans la suite, nous détaillons les résultats par applications.

Pour les applications *FMM* et *Ocean*, R_NUCA offre de meilleures performances par rapport à notre stratégie. Ces résultats semblent contradictoires aux résultats obtenus précédemment concernant les deux métriques : taux de d'échec et la latence du réseau. La raison de cette dégradation est que les deux types d'accès, privé et partagé, sont équilibrés. Donc, même si le taux de d'échec est plus faible en utilisant notre approche (voir la figure 7.9), la pénalité d'accès aux données privées dans les bancs distants est coûteuse. Ainsi, le bénéfice d'accès à des bancs proches du demandeur lorsque les données sont privées, explique pourquoi R_NUCA est meilleur.

Pour *Radix*, le temps d'exécution en utilisant notre schéma est similaire aux résultats obtenus par R_NUCA, mais par rapport à S_NUCA la situation est pire. Cela s'explique pour une multitude de motifs, telles que l'augmentation du taux d'échec et du nombre de blocs évincés en raison d'un déséquilibre lors de la distribution des blocs au niveau des bancs L2. En outre, les améliorations de la latence du réseau soulignées dans la section 7.3.4 (15.18%), n'ont pas été traduites par une performance globale bien meilleure. Nous estimons que les résultats ont été affectés par le nombre élevé d'accès distants.

Lors de la deuxième configuration (16 bancs L2) notre approche améliore 6 applications sur 8 par rapport aux deux stratégies, S_NUCA et R_NUCA. Cette amélioration est

de 6% par rapport à la première est de 14.6% par rapport à la seconde.

Les résultats de la figure 7.15 montrent que pour l'application *FMM*, H_NUCA permet d'avoir une accélération très élevée relativement aux deux autres stratégies. Ceci reste en corrélation avec l'amélioration illustrée par le taux d'échec. En revanche, la dégradation du temps d'exécution avec la stratégie R_NUCA est liée aux résultats de taux d'échec (figure 7.9) affectés par la distribution des blocs au niveau des bancs L2.

D'autre part, les applications telles que *Radix* et *Ocean* sont moins efficaces. Leur temps d'exécution est réduit de 30 et 12% par rapport à S_NUCA. Par rapport à R_NUCA, dans *Radix*, la situation s'aggrave et une perte de 20% en temps d'exécution est obtenue. Inversement, dans *Ocean* l'accélération réalisée par notre approche est de 1.5x par rapport à R_NUCA. Tous ces résultats sont en cohérence avec le taux d'échec.

Par analogie à ce qui a été conclu lors de la première simulation, nous nous rendons compte que lorsque nous augmentons le nombre de bancs, l'accélération est dégradée pour certaines applications en raison du déséquilibre de la charge qui est plus important. D'après les expérimentations, nous déduisons que même avec un nombre élevé de bancs L2, notre approche est compétitive.

Synthèse Après analyse des résultats de cette partie, nous pouvons déduire les points suivants :

- Lors de l'utilisation de notre stratégie, l'amélioration du temps d'exécution ne suit pas toujours les rythmes du taux d'échec et de la latence moyenne du réseau, car la pénalité d'échec au niveau de la cible est aussi un facteur déterminant.
- L'accès à tous les bancs L2 sans aucune restriction assure une distribution des données qui élimine le goulot d'étranglement lors du traitement des demandes. Éventuellement, la mauvaise performance en terme d'accélération pour quelques applications est essentiellement due au déséquilibre de la charge.
- Il est notable que R_NUCA a des performances inférieures à celle de S_NUCA, surtout dans la deuxième configuration. Cela semble être en contradiction partielle avec les résultats originaux de cette stratégie [HFFA09]. Cependant, les résultats positifs dans l'article original concernent les applications de type serveur web tel que *OLTP*. Pour les programmes scientifiques caractérisés par un large *working set* et des accès relativement partagés, les résultats ne sont pas si favorables et une telle conclusion a également été mentionnée dans [AHS+15]. Donc, nous estimons que notre approche est meilleure pour ce type d'applications.

7.3.6 Consommation énergétique

L'un des critères de performance de la hiérarchie mémoire est la consommation d'énergie. Cette consommation est majoritairement due à l'accès des processeurs à la mémoire à plusieurs niveaux. Nous nous intéressons aux transferts au niveau du réseau de communication vu que notre approche est à la base d'un support de multidiffusion. Il convient donc d'étudier la consommation d'énergie au niveau réseau. Concrètement, cette métrique englobe l'énergie du routeur et des liens.

La figure 7.16 illustre les résultats pour les deux configurations étudiées dans ce qui précède.Les valeurs sont données par Gem5. Nous pouvons observer que, dans la première configuration, notre stratégie devrait permettre de réduire considérablement

la consommation d'énergie. L'économie réalisée en moyenne serait de l'ordre de 50% par rapport à S_NUCA et R_NUCA.

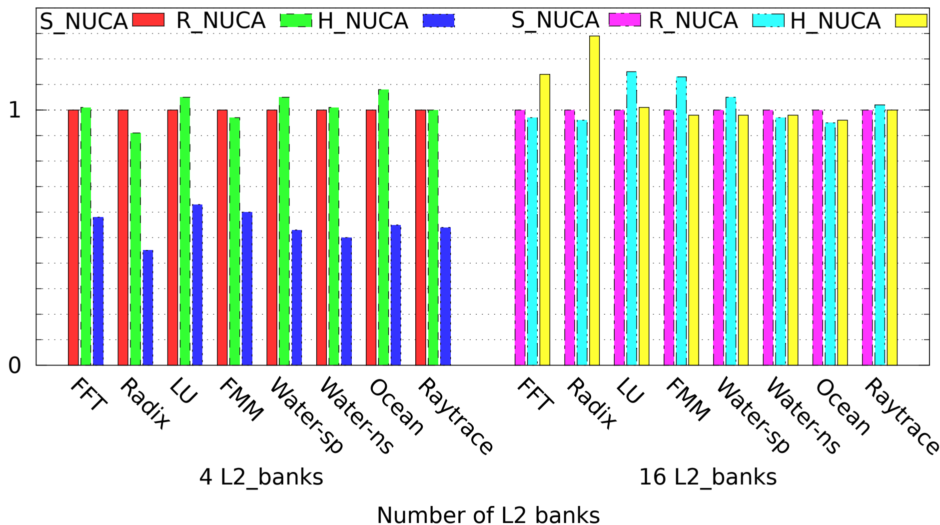


FIGURE 7.16 – Consommation énergétique au niveau du réseau d'interconnexion normalisé à S_NUCA

Nous remarquons que R_NUCA offre des performances égales ou inférieures à celles de la stratégie S_NUCA pour la majorité des applications, à part *Radix* et *FMM*. Le gain apporté pour ces dernières est faible (de l'ordre de 9% et 2% respectivement).

Lors de la deuxième configuration, la consommation d'énergie est faiblement améliorée (1-5%) par rapport à la stratégie de référence pour la majorité des applications. Par contre, pour *Radix* et *FFT*, notre solution implique une perte de 30% et 12% respectivement. Bien qu'une étude approfondie serait nécessaire, nous pouvons considérer, en première approximation, que la consommation est proportionnelle à la distance parcourue par les accès. Or, pour les deux applications, l'accès à des bancs distants est plus fréquent, 65% pour *Radix* et 55% pour *FFT* (voir figure 7.14).

Par rapport à R_NUCA, à part *FFT* et *Radix*, le gain énergétique réalisé par H_NUCA est de l'ordre de 6% en moyenne. Sur *LU*, *FMM* le gain atteint 13%, mais moins, 5%, sur *Water-sp*. La consommation plus importante dans R_NUCA est probablement dû à l'interrogation des TLBs pour le classement des accès pour ce type d'applications.

En ce qui concerne *FFT* et *Radix*, R_NUCA réduit la consommation d'énergie car les données accédées sont le plus souvent trouvées et n'implique pas un taux d'échec important au niveau des TLBs. En revanche, pour H_NUCA l'échec au niveau de la cible induit l'accès à tous les bancs distants du cache. En effet, la fin de la multidiffusion est détectée lorsque l'on atteint le nœud le plus distant et non pas le nœud contenant la donnée, et plus les cibles de la diffusion sont lointaines, plus la diffusion est coûteuse en terme de consommation d'énergie.

7.3.7 Surcoût matériel

Étant donné l'utilisation de la multidiffusion suite à un échec au niveau de la cible, nous cherchons à présent à mettre en évidence les coûts associés à son support. En fait, le mécanisme de diffusion cible les bancs d'un même niveau de cache, généralement le niveau du banc censé être l'hébergeant de la donnée au moment de sa demande. Ainsi la construction du réseau virtuel reliant les différents bancs d'un même niveau et ayant comme topologie un tore nécessite 4 registres par routeur. Ces registres contiennent les coordonnées des routeurs logiques voisins. Ensuite, nous optons pour la séparation du trafic de la multidiffusion du trafic ordinaire en autorisant le premier à emprunter un ensemble de canaux virtuels (VCs) dédiés. Les canaux virtuels sont coûteux, donc l'ajout incontrôlé est pénalisant en terme de surface. La solution était de dédier un VC de chaque ensemble à un type de trafic classique donné. Ainsi, le nombre de VCs total est le même utilisé dans les stratégies S_NUCA et R_NUCA et les comparaisons sont par la suite équitables.

7.3.8 Passage à l'échelle

L'objectif majeur des systèmes sur puce est le passage à l'échelle en garantissant les bonnes performances de la hiérarchie mémoire. Nous étudions dans cette section l'efficacité de notre stratégie H_NUCA pour un système de 64 cœurs. Notre analyse est basée sur les paramètres de performance, le taux de défauts de cache, la latence moyenne du réseau et le temps d'exécution pour l'évaluation de notre approche.

Nous reprenons l'environnement de simulation déjà présenté dans la section 7.1 et nous modélisons une architecture à 64 cœurs avec 16 bancs de cache L2 dans laquelle chaque banc est partagé par 4 cœurs. Là encore, les comparaisons de nos résultats sont faites par rapport à S_NUCA et R_NUCA.

7.3.8.1 Taux de défauts de cache

D'une manière générale notre stratégie H_NUCA induit beaucoup plus de défauts de cache L2, comme le reportent les histogrammes de la figure 7.17, par rapport aux autres stratégies pour la majorité des applications : en moyenne 14% pour S_NUCA et 10.42% pour R_NUCA.

En s'appuyant sur l'organisation de la hiérarchie des caches comme elle a été proposée, la logique voudrait que l'accès sans aucune restriction à tous les bancs formant un cache L2 réduise le taux d'échec. Toutefois ceci n'est pas le cas comme l'attestent nos résultats avec H_NUCA. La cause qui peut expliquer ce phénomène est la diminution du taux de partage des données. En fait, avec l'augmentation du nombre de cœurs, la répartition des tâches sur ces derniers influe sur ce taux de partage. Par conséquent, le placement des données au niveau de certains bancs oblige l'évincement d'autres données. D'où la nécessité de revisiter la stratégie de remplacement utilisée avec H_NUCA et en particulier d'étudier la priorité de remplacement lorsqu'un banc est très fortement sollicité.

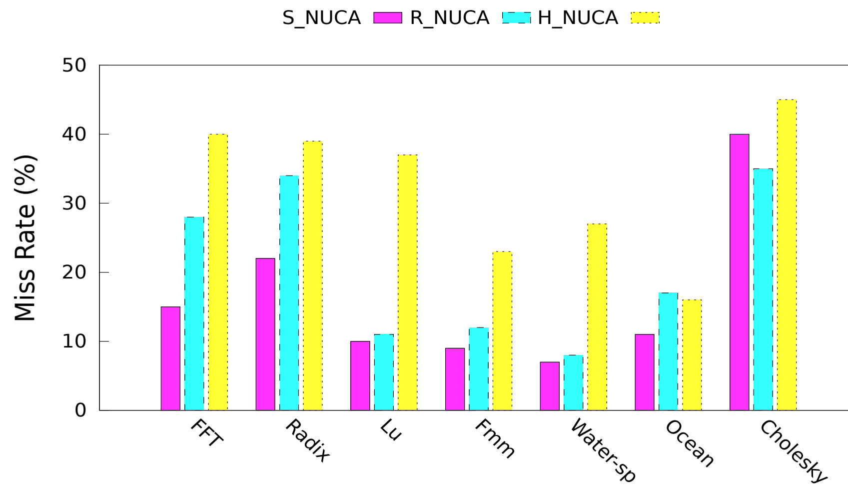


FIGURE 7.17 – Taux de défauts de cache L2

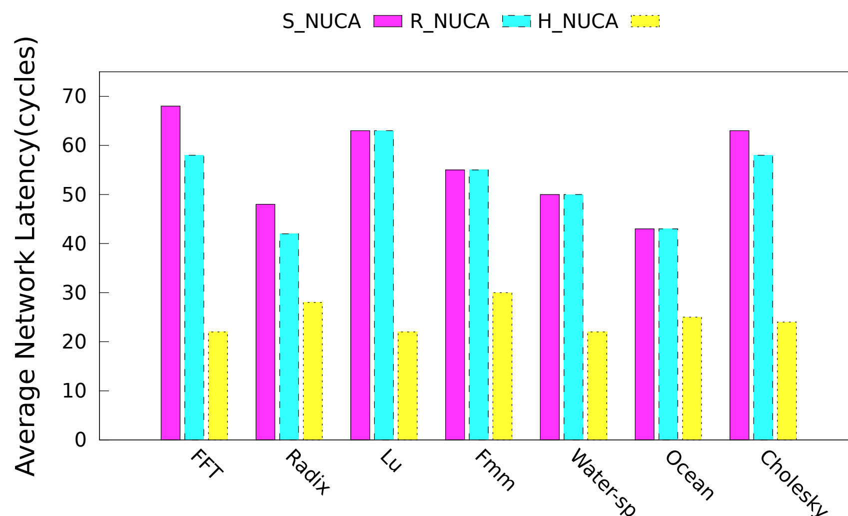


FIGURE 7.18 – Latence moyenne du réseau

7.3.8.2 Latence moyenne du réseau

Les résultats de cette métrique sont présentés dans la figure 7.18. Nous observons pour l'ensemble des applications que la stratégie H_NUCA offre de meilleurs latences comparé à S_NUCA et R_NUCA.

Cet écart de performance est obtenu grâce à deux facteurs illustrés précédemment : le placement dans le banc cible et l'utilisation du réseau virtuel. Le premier facteur lié à la répartition des données a une légère influence sur le trafic généré. Le second facteur permet de réduire la congestion au niveau des ressources et ceci grâce à la séparation des trafics.

En outre, il est à noter que la bande passante cumulée disponible sur le réseau croît avec le nombre de nœuds connectés. Ceci reste en faveur de notre stratégie basée sur le mécanisme de diffusion.

7.3.8.3 Temps d'exécution

Contrairement à nos attentes, la stratégie H_NUCA offre de moins bonnes performances que les autres stratégies dans la majorité des applications. Cette différence de performance ne dépasse pas les 25% est en général inférieur à 10% par rapport à S_NUCA. D'une manière similaire, elle atteint en moyenne 15% par rapport à R_NUCA. La figure 7.19 illustre l'accélération de H_NUCA et R_NUCA par rapport à S_NUCA pour les différentes applications.

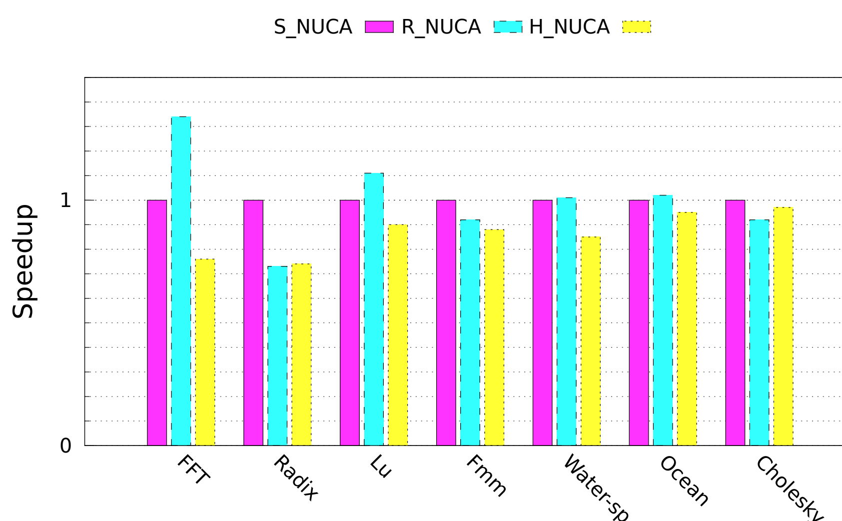


Figure 7.19 – Accélération pour les différentes applications

Cette perte en temps d'exécution est en corrélation avec les pourcentages de défauts de cache de la section 7.3.8.1. En effet, les accès inéquitables aux différents bancs induisent certainement des points de congestion et ceci contribue à faire croître de façons démesurée la latence des accès. Nous observons également cette dégradation malgré l'amélioration des latences moyennes du réseau pour les différentes applications. Ceci est dû probablement à l'inconvénient majeur de la diffusion dont la fin n'est pas conditionnée par la satisfaction de la requête mais par l'envoi de cette dernière à tous les bancs cibles.

Il convient donc de relativiser les résultats qui au premier abord semblent très bons. En vue du passage à l'échelle, plusieurs faiblesses doivent être identifiées et étudiées pour être compétitif.

7.4 Conclusion

Au vu des analyses précédentes, l'utilisation du concept du réseau logique virtuel permet d'améliorer de manière significative les performances du réseau et de mieux passer à l'échelle dès lors qu'on s'appuie sur les générateurs de trafics synthétiques.

En s'appuyant sur un protocole de cohérence de caches qui exploite ce concept pour une architecture multicœurs ayant une hiérarchie mémoire distribuée, les résultats ont montré que les performances obtenues sont généralement meilleures pour des

architectures de taille moyenne. Ils dépassent ceux obtenus en appliquant deux autres stratégies : S_NUCA et R_NUCA.

Pour une minorité d'applications les résultats sont moins stables. Certes, les performances sont fortement dépendantes des différents schémas de données dans une application. À cela s'ajoute le degré de partage des données : qu'elles soient à usage unique ou multiple, leur distribution au niveau de la hiérarchie influe sur la performance. En fait, cette distribution peut éliminer le goulot d'étranglement au niveau des bancs, mais le déséquilibre de la charge résulte du besoin des tâches. Par conséquent, l'évincement des données devra être optimisé par l'exploration de différentes techniques de remplacement. L'effet de ce dernier facteur est d'autant plus notable que les architectures sont de plus grande taille.

Nous détaillerons dans le prochain chapitre les limitations de cette étude et les perspectives de recherche ainsi que les pistes à explorer.

Chapitre 8: Conclusions et Perspectives

8.1 Conclusions

Nous avons tenté à travers cette thèse d'apporter une solution aux problèmes liés à l'accès aux données dans une hiérarchie mémoire. Notre démarche consiste à virtualiser la hiérarchie et à exploiter les capacités du réseau qui interconnecte les mémoires caches.

Nous avons soulevé dans le chapitre 2 un certain nombre de questions. À la lumière de nos travaux, nous sommes en mesure d'y apporter des réponses, au moins partielles.

En s'appuyant sur les capacités des réseaux sur puce, quelle organisation de la hiérarchie mémoire doit-on adopter pour avoir des accès efficaces ?

Nous avons étudié en premier lieu les contraintes fondamentales des systèmes multiprocesseurs et nous nous sommes focalisés sur l'aspect mémoire de ces derniers étant donné que l'accès aux donnés est le point clé de la performance de ces systèmes.

Notre approche prend en considération l'organisation hiérarchique des architectures à mémoire partagée et distribuée interconnectées par un NoC. Il s'agit de l'architecture NUCA où le temps d'accès à ces bancs diffère de la perspective d'un processeur à un autre. Nous avons réalisé une étude des différentes variantes de ce type d'architecture, celle qui réduit l'utilisation du réseau au détriment de la performance et celle qui négocie l'utilisation du réseau pour des performances élevées.

Étant donné que les transferts de données entre les niveaux de la hiérarchie déterminent les performances de l'unité de mémorisation, nous avons redéfini la structure de la hiérarchie des caches comme suit : Tous les bancs d'un niveau donné (hormis le L1) peuvent être accédés sans aucune restriction par tous les bancs du niveau supérieur. En effet, le transfert intra-niveau est permis et il est justifié dans la mesure où il sera opportun pour une donnée non trouvée dans un banc de s'affranchir de la traversé de toute la structure hiérarchique du cache pour atteindre la donnée qui existe dans un autre banc du même niveau.

Comment placer les données dans les bancs mémoires afin de minimiser le trafic sur la puce, en gardant à l'esprit la contrainte de capacité limitée de ces bancs ?

Le placement optimal des données au regard de la localisation de l'unité de calcul qui les utilise est un défi qui a préoccupé les concepteurs de l'architecture NUCA. En effet, la latence de communication, dépendante de l'organisation des caches, est fortement liée à la manière dont les données sont gérées. De par la capacité limitée de l'espace de mémorisation disponible sur la puce, les bancs de cache ne peuvent pas contenir

toutes les données. La solution utilisée consiste à considérer que l'ensemble des bancs d'un niveau donné forme un cache unifié. Un banc est défini comme cible à consulter en premier lieu. Si ce dernier n'a pas pu satisfaire la requête, les autres bancs seront consultés. Ainsi, la probabilité d'avoir une réponse de la part d'un niveau supérieur est élevée et ceci a permis d'augmenter le taux de succès par niveau de cache. Nous nous sommes appuyés sur la métrique du taux de défauts de cache et nous avons réalisé un ensemble d'expérimentations, à l'aide du simulateur Gem5, pour évaluer notre solution et la comparer à la hiérarchie classique.

Les résultats obtenus sont généralement acceptables, mais nous ne pouvons néanmoins pas conclure à un gain en performance (réduction du temps d'exécution). Ce serait le cas si la latence d'accès moyenne à un niveau donné était compensée par la latence économisée grâce à la réduction du taux d'échec.

Comment assurer la cohérence des données en utilisant la diffusion dès lors que l'on s'appuie sur la nouvelle structuration de la hiérarchie mémoire ?

La question primordiale pour la nouvelle organisation de la hiérarchie des caches est le maintien de la cohérence dès lors que l'on souhaite utiliser des applications parallèles multitâches. Pour cela nous avons proposé un mécanisme inspiré des protocoles d'espionnage (*snoop*). Ce mécanisme est basé sur la diffusion dans un niveau de cache si le banc cible consulté ne contient pas la donnée. Dans ce cas, les lignes résidantes dans un banc autre que la cible seront mises à jour à chaque fois qu'elles sont demandées. Cette diffusion n'est qu'une sorte de multidiffusion étant donné que les cibles sont les bancs de cache d'un niveau (L2/L3). Donc, le surcoût en bande passante est acceptable dans la plupart des cas.

Nous avons apporté des modifications au protocole MESI afin de l'adapter à la hiérarchie proposée. L'objectif est de fournir une hiérarchie mémoire cohérente, sans interblocage qui pourra être utilisée comme support de mémorisation. Nous avons réussi à exécuter des applications réelles sur une architecture à base de la nouvelle hiérarchie, ce qui confirme que notre protocole est bien fonctionnel. En outre, une vérification plus formelle, basée sur une représentation abstraite décrite dans le langage *ad-hoc* de Gem5, a été réalisée dans le cadre d'un stage de Master [Roq17].

Comment adapter la structure du réseau afin de faciliter les interactions entre les modules de la hiérarchie mémoire ?

Afin d'assurer les transferts exigés par la hiérarchie, en particulier les transferts intra-niveau, nous avons opté pour une technique de virtualisation du réseau. Cette technique donne l'illusion d'avoir un système multi-couches dans lequel les différents niveaux de cache sont manipulés séparément. En effet, les bancs de chaque niveau sont reliés par l'intermédiaire d'un réseau virtuel (VN). Ces bancs ne sont pas nécessairement des voisins physiques.

L'implantation de ces réseaux requiert l'ajout de quelques registres (selon la topologie) pour contenir les coordonnées des voisins logiques. Ainsi, le surcoût de ces réseaux est faible.

Ayant en tête que cette architecture permet de limiter la contention qui apparaît dans un réseau d'interconnexion unique, son exploitation dans le cadre de notre hiérarchie était le but de ce travail. De fait, le trafic de la multidiffusion sera séparé du

trafic ordinaire et ceci pourrait alléger la charge au niveau du réseau physique.

Quel algorithme de routage permet d'implanter la stratégie de diffusion et quelles modifications cela implique-t-il au niveau du support matériel ?

Le réseau virtualisé comme il a été défini nécessite un algorithme de routage adéquat pour l'acheminement des paquets. C'est un algorithme distribué et générique qui pourra être utilisé avec n'importe quelle topologie. Sa mise en œuvre nécessite l'ajout que quelques portes logiques au niveau du routeur. Les résultats obtenus en évaluant le réseau en mode autonome avec des générateurs de trafics sont très encourageants. Ils montrent que la solution offre un gain en performance.

L'exploitation de ce réseau dans le cadre du protocole associé à la hiérarchie proposée requiert aussi un mécanisme de routage qui supporte la multidiffusion. Cette dernière exige l'ajout d'une unité de réplication au niveau du routeur et la modification des composants avec lesquels il interagit afin de pouvoir contrôler le flux de données.

La latence moyenne du réseau a été largement améliorée en utilisant notre approche pour toutes les applications et sur les différentes architectures. Cependant, les temps d'exécution ne suivent pas le même rythme. En effet, sur une architecture à 16 processeurs, un gain a été obtenu pour la majorité des applications. Les quelques exceptions sont dues à un déséquilibre dans l'utilisation des différents bancs. En revanche, sur une architecture à 64 processeurs les déséquilibres s'accentuent, les latences de la diffusion augmentent, et les applications s'exécutent moins vite.

La performance de la hiérarchie dépend de plusieurs facteurs tels que le comportement des applications et la manière dont la charge est répartie. Un placement optimisé des données reste primordial pour l'amélioration des performances d'un système.

8.2 Perspectives

Nous présentons ici des réflexions sur les améliorations qui devraient être apportées à notre solution.

- **La multidiffusion** : Caractérisée par sa simplicité et son faible coût (en terme de ressources) de mise en place, son inconvénient majeur est qu'elle est conditionnée par la disponibilité des canaux virtuels (VCs) comme décrit dans le chapitre 6, ce qui induit potentiellement des latences supplémentaires. Une solution envisageable serait de déclencher la réplication au niveau de l'étage ST. En effet, dès qu'un flit est candidat à l'unité de contrôle du commutateur, il subirait la réplication. De cette manière, le routeur n'aura plus besoin de sauvegarder les répliques et la latence sera réduite. Ceci nécessite la modification de l'arbitrage au niveau du l'étage SA.
- **Mettre fin à la multidiffusion dès que la requête est satisfaite** : Un autre point qui mériterait d'être amélioré concerne le mécanisme de multidiffusion. Ce dernier vise à chercher une donnée pour la mettre à jour dans le contexte de nos travaux. Or, dès que la donnée est trouvée, la consultation du reste des bancs est inutile vu qu'un niveau de cache garantit l'unicité des données.
  Il serait souhaitable d'interrompre la diffusion en ajoutant un générateur qui se charge d'envoyer des requêtes "abort" à tous les nœuds dans toutes les directions en dehors de celle à partir de laquelle la donnée a été reçue. Le déclenchement de

ce mécanisme aura lieu dès que la cible (source de la diffusion) reçoit la donnée et un nombre d'acquittement inférieur au nombre totale prévu. Ainsi, la cible n'aura pas besoin d'attendre les acquittements pour transférer la donnée au demandeur initial et la latence sera réduite.

- **L'algorithme de remplacement** : Lors de nos expérimentations, nous avons découvert une pathologie qui est liée à un déséquilibre dans la charge du travail pour quelques applications. Ceci a favorisé l'apparition de deux types de défauts, l'un lié à la capacité et l'autre aux conflits. Dans ce genre de situation il serait utile de s'appuyer sur l'algorithme de remplacement des lignes. En effet, le placement d'un bloc récent qui est peu utilisé à la place d'un ancien bloc plus important peut entraîner des effets non négligeables sur les latences.
  En raison de la forte variation des accès et à cause de la répartition des données, une étude approfondie doit être menée afin de cerner les facteurs de dégradation.
- **Vérification formelle du protocole de cohérence de caches** : Bien entendu, l'assurance que les applications utilisées dans nos expérimentations s'exécutent de façon conforme à leur spécification est basé sur l'hypothèse que le simulateur n'a pas détecté un conflit et/ou n'a pas signalé un interblocage. Cela nous rassure sur le fait que le protocole implémenté et associé à la nouvelle hiérarchie des caches est bien fonctionnel.
  Tester un plus grand nombre d'applications augmente la probabilité de détection d'une anomalie. Mais, il serait intéressant de procéder par les méthodes formelles. Un travail de modélisation et de preuve a été réalisé en utilisant des techniques de preuve de théorèmes [Roq17]. Certes, cette approche est basée sur une abstraction, aussi une vérification à un niveau plus bas reste un axe de recherche ouvert.
- **Changement dynamique de la cible** : Bien que notre solution permette de réduire le taux de défauts par niveau de cache, cela ne s'est pas traduit nécessairement par une amélioration du temps d'accès. Ce constat est d'autant plus notable que des architectures contiennent un grand nombre de processeurs.
  Le choix de la cible dans nos travaux a été fixé au moment de la configuration d'une manière statique. Or, chaque application comporte plusieurs phases de comportement et chaque phase est caractérisée par une charge mémoire et un schéma d'accès spécifique. Le problème qui se pose est de choisir la cible qui permet de maximiser le taux de succès local pour une application donnée tout au long de son exécution.
  Une nouvelle perspective est de développer une méthode permettant de changer la cible de manière dynamique, selon des critères à définir, et d'étudier son impact sur les performances du système.

Glossaire

Ack	Acknowledgement
AAL	Average Access Latency
ACM	Adaptive Controlles Migration
AH	Header Addition
AMAT	Average Memory Access Time
AMT	Average Miss Time
CCE	Centralized Cooperative Engine
CCM	Cluster Cache Monitor
CMP	Chip Multi-Processor
CMOS	Complementary Metal Oxide Semiconductor
COMA	Cache Only Memory Access
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DCC	Distributed Cooperative Caching
DOR	Dimension Ordered Routing
DRAM	Dynamic Random Access Memory
E	East
EVC	Express Virtual Channel
flit	FLow control unIT
FS	Full System
IP	Intellectual Property
ISA	Instruction Set Architecture
L	Logic
LLC	Last Level Cache
LRU	Least Recently Used
LT	Link Traversal
MC	MultiCast
N	North
NoC	Network On Chip
NUCA	Non Uniform Cache Access
NUMA	Non Uniform Memory Access
OS	Operating System
OT	Ordinary Transmission
p	Physic
RAM	Random Access Memory
RC	Routing Computation
RH	Header Removal
ROK	Read OK
S	South
SA	Switch Allocation
SE	System-call Emulation
ST	Switch Traversal
SLICC	Specification Language for Implementing Cache Coherence
SoC	System On Chip
TG	Traffic Generator
T	Target
UMA	Uniform Memory Access
VA	Virtual-channel Allocation
VT	Threshold Voltage
VN	Virtual Network
VC	Virtual Channel
W	West
WOK	Write OK

Annexe : Protocole de cohérence

Machines d'états des caches L1 et L2

Les pages suivantes donnent les machines d'états des caches L1 et L2, et les tableaux de transition associés. Un tableau énumère les actions menées et l'état de la ligne de cache final pour une transaction donnée (1ère ligne) et un état de ligne de cache initial (1ère colonne).

Pour des raisons de lisibilité, les dessins prennent toute la page. Le tableau associé au cache L2 s'étend sur quatre pages.

Store

Fwd_GETS_B, Fwd_GET_INSTR_B,
Fwd_GETS, Fwd_GET_INSTR

Ifetch, Load

L1_Replacement, Ifetch, Load, Store

Load, Ifetch, Store, L1_Replacement

L1_Replacement, Store

E

M_I

Fwd_GETS_B, Fwd_GET_INSTR_B,
Fwd_GETS, Fwd_GET_INSTR, Fwd_GETX, Inv

Data_Exclusive, Data_Exclusive_fwd

Fwd_GETX, Inv

WB_Ack

Fwd_GETX, Inv

Inv, Load, Ifetch,
Store, L1_Replacement

WB_Ack

Inv, L1_Replacement

SINK_WB_ACK

I

Data_Exclusive, Data_Exclusive_fwd

Data_all_Acks, Data_fwd,
Data_fwd_S, Data_fwd_S_ISS, DataS_fromL1

Inv, Load, Ifetch,
Store, L1_Replacement

IS_I

M

Store

Ifetch, Load

Load, Ifetch, Store, L1_Replacement

Ifetch, Load

Inv

IS

Data_all_Acks, Data_fwd, Data_fwd_S,
Data_fwd_S_ISS, DataS_fromL1

Inv, L1_Replacement

Ifetch, Load

Inv, L1_Replacement

Fwd_GETS_B, Fwd_GET_INSTR_B,
Fwd_GETS, Fwd_GET_INSTR

Data_all_Acks, Data_Exclusive,
Data_fwd, Data_fwd_all

Ack_all, Ack_all_fwd,
Ack_all_up, Data_Exclusive

NP

Ack, Ack_H, Inv, Load,
Ifetch, Store, L1_Replacement

Store

IM

Store

Data, Data_fwd_S

Ack, Ack_H, Load,
Ifetch, Store, L1_Replacement

Ifetch, Load

Inv

Store

SM

S

Machine d'état du cache L1

Tableau de transition associé au cache L1

	STATE	Processor request			L1 action		From L2											From other L1		
		Load	Store	I-fetch	Eviction	Data/ Data_fwd_S	Data_Exc/ Data_Exc_fw	Data_S_from_L1	Data_fwd	Data_fwd_S/ Data_fwd_S_ISS	Data_All_acks	Invalid	Ack/ Ack_H	Ack_all	Ack_all_fwd	Ack_all_up	WB_ack (L1_PUTX)	Fwd_Gets / Fwd-Get-inst	Fwd_Getx	Fwd_Gets_B / Fwd-Get-inst_B
Etats stables	**NP/I**	Gets → IS	Getx → IM	Get Instr → IS	Dealloc $block							Send ack to L2(requestor) Inv-**ack**								
	S	hit	Getx(upgrade)→ SM	hit	evict→ I							Inv-**ack** & eviction to cpu → I								
	M	hit	hit	hit	send data to L2(putx)& evict →M_I							Inv-**data** & eviction to cpu → I						Send data to requestor& send to L2 → S	Send data to requestor & eviction to cpu → I	Send data to requestor → S
	E	hit	hit → M	hit	send data to L2(putx) & evict →M_I							Inv-**ack** & eviction to cpu → I						Send data to requestor& send to L2 → S	Send data to requestor & eviction to cpu → I	Send data to requestor → S
Etats transitoires	**IS** (read_miss)	stall	stall	stall	stall		Write data &load hit & Exc_unblock to L2 → E	Write data &load hit & unblock to L2 → S	Write data &load hit → S	Write data &load hit & unblock to L2 →S	Write data &load hit → S	Inv-ack → IS_I								
	IM (write_miss)	stall	stall	stall	stall	write done & update ack count → SM	Write data &store hit & Exc_unblock to L2 (broad) → M		Write data & Exc_unblock to L2 (broad) & store hit → M		write done &store hit& Exc_unblock to L2 → M	Inv-ack	Update ack → IM							
	SM (upgrade)	stall	stall	stall	stall							Inv-ack → IM	Update ack → SM	Send ExcUnblock,hit store, dealoc tbe → M	Send ExcUnblock, hit store, dealoc tbe → M	Send ExcUnblock, hit store, dealoc tbe → M				
	M_I	stall	stall	stall	stall							Send data to L2 from tbe (wb-data) → Sink_wb_ack					Dealloc tbe → I	Send data to requestor & to L2 from tbe (resp data) → Sink_wb_ack	Send data to requestor from tbe (resp data) → Sink_wb_ack	Send data to requestor from tbe (resp data) → Sink_wb_ack
	Sink_wb_ack	stall	stall	stall	stall							Inv-ack					Dealloc tbe → I			
	IS_I	stall	stall	stall	stall		write data &load hit & Exc_unblock to L2 → E	Write data &load hit & unblock to L2 → I	Write data &load hit → I	Write data &load hit & unblock to L2 → I	Write data &load hit → I	Inv-ack								

Ack, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETX,
L1_UPGRADE, L1_GETS, L1_GET_INSTR,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX,
wb_Data_fwd, wb_Data_fwd_S

I_I

L2_Replacement_clean

Ack_all

Ack_all, WB_Data, WB_Data_clean

WB_Data, WB_Data_clean

ack_data, Unblock

Exclusive_Unblock

ack_data, Fwd_Putx, Fwd_Putx_old,
L1_GETS, L1_GET_INSTR, L1_PUTX,
L1_PUTX_old, L2_Fwd_GET_INSTR, Unblock

ack_data, Unblock

dir_Inv, Fwd_Putx, Fwd_Putx_old,
L1_GETS, L1_GET_INSTR, L1_GETX,
L1_UPGRADE, L1_PUTX, L1_PUTX_old,
L2_Fwd_GETS, L2_Fwd_Upgrade, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean,
wb_Data_fwd, wb_Data_fwd_S

SS

L1_GETX, L1_UPGRADE, L2_Fwd_GETS,
L2_Fwd_GETX, L2_Fwd_Upgrade,
wb_Data_fwd, wb_Data_fwd_S

SS_MB

Exclusive_Unblock

ack_data_S

L2_Replacement, dir_Inv

L1_GET_INSTR

Ack, dir_Inv, Fwd_Putx,
Fwd_Putx_old, L1_GETX, L1_UPGRADE,
L1_GETS, L1_GET_INSTR, L1_PUTX,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX,
wb_Data_fwd, wb_Data_fwd_S

S_I

dir_Inv, Fwd_GETX, Fwd_Putx,
Fwd_Putx_old, L1_GETX, L1_UPGRADE,
L1_GETS, L1_GET_INSTR, L1_PUTX,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX,
wb_Data_fwd, wb_Data_fwd_S

Ack_all

M_I

WB_Data, WB_Data_clean, Ack_all

ack_data_f

dir_Inv, Fwd_GETX, Fwd_Putx,
Fwd_Putx_old, L1_GETX, L1_GETS,
L1_GET_INSTR, L1_PUTX_old, L1_PUTX,
L2_Fwd_GETS, L2_Fwd_Upgrade, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean,
wb_Data_fwd, wb_Data_fwd_S

ack_B_all, ack_B_all_token

IM_B

ack_B, ack_Pri, dir_Inv,
Fwd_GETX, Fwd_Putx, Fwd_Putx_old,
L1_GETX, L1_GETS, L1_GET_INSTR,
L1_PUTX, L1_PUTX_old, L2_Fwd_GETS,
L2_Fwd_GET_INSTR, L2_Fwd_GETX, L2_Replacement,
L2_Replacement_clean, WB_Ack,
wb_Data_fwd, wb_Data_fwd_S

IM

Mem_Data

Mem_Data

Mem_Data

Mem_Data

Mem_Data

Ack_all, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETS,
L1_GET_INSTR, L1_GETX, L1_UPGRADE,
L1_PUTX, L1_PUTX_old, L2_Fwd_GETS,
L2_Fwd_Upgrade, L2_Fwd_GET_INSTR, L2_Fwd_GETX,
L2_Replacement, L2_Replacement_clean, wb_Data_fwd,
wb_Data_fwd_S, WB_Data, WB_Data_clean,
WB_Data_F, WB_Data_clean_F

MT_MB

Exclusive_Unblock

L1_GETX, wb_Data_fwd

WB_Data_F, WB_Data_clean_F,
WB_Data, WB_Data_clean

Fwd_Putx, L1_PUTX

ack_data

Fwd_Putx, Fwd_Putx_old,
L1_PUTX, L1_PUTX_old

L1_GETS, L1_GETX, L2_Fwd_GETS, L2_Fwd_GETX

M

ack_B_up, ack_data

L2_Replacement_clean, L2_Replacement, dir_Inv

wb_Ack

dir_Inv, Fwd_Putx, Fwd_Putx_old,
L1_GETX, L1_UPGRADE, L1_GETS,
L1_GET_INSTR, L1_PUTX, L1_PUTX_old,
L2_Fwd_GETS, L2_Fwd_Upgrade, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, wb_Data_fwd, wb_Data_fwd_S

MT_I

L2_Replacement, dir_Inv

dir_Inv, Fwd_Putx, Fwd_Putx_old,
L1_GETS, L1_GET_INSTR, L1_GETX,
L1_UPGRADE, L1_PUTX, L1_PUTX_old,
L2_Fwd_GETS, L2_Fwd_Upgrade, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean

MT_IB

Unblock

ack_data, dir_Inv, Fwd_Putx,
Fwd_Putx_old, L1_GETS, L1_GET_INSTR,
L1_GETX, L1_UPGRADE, L1_PUTX,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX, L2_Replacement,
L2_Replacement_clean, wb_Data_fwd, wb_Data_fwd_S

L1_GETS, L1_GET_INSTR, wb_Data_fwd_S

MT_IIB

WB_Data_F, WB_Data_clean_F,
WB_Data, WB_Data_clean

dir_Inv, Fwd_Putx_old, Fwd_Putx,
L1_GETS, L1_GET_INSTR, L1_GETX,
L1_UPGRADE, L1_PUTX, L1_PUTX_old,
L2_Fwd_GETS, L2_Fwd_Upgrade, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean,
wb_Data_fwd, wb_Data_fwd_S

Exclusive_Unblock

MT_SB

WB_Data, WB_Data_clean, WB_Data_F, WB_Data_clean_F

Fwd_Putx_old, L1_PUTX_old

MT

L2_Fwd_GETS, L2_Fwd_GET_INSTR

Fwd_Putx, Fwd_Putx_old, L1_GETS,
L1_GET_INSTR, L1_GETX, L1_UPGRADE,
L1_PUTX, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX, L2_Replacement,
L2_Replacement_clean, wb_Data_fwd, wb_Data_fwd_S

L2_Replacement_clean

MT_IBB

Fwd_GETX, L2_Fwd_GETX

dir_Inv, Fwd_GETX, Fwd_Putx,
Fwd_Putx_old, L1_GETX, L1_UPGRADE,
L1_GETS, L1_GET_INSTR, L1_PUTX,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_Upgrade,
L2_Fwd_GET_INSTR, L2_Fwd_GETX,
wb_Data_fwd, wb_Data_fwd_S

MCT_I

dir_Inv, Fwd_Putx, Fwd_Putx_old,
L1_GETS, L1_GET_INSTR, L1_GETX,
L1_UPGRADE, L1_PUTX, L2_Fwd_GETS,
L2_Fwd_Upgrade, L2_Fwd_GET_INSTR, L2_Fwd_GETX,
L2_Replacement, L2_Replacement_clean,
wb_Data_fwd, wb_Data_fwd_S

MT_MBB

L2_Fwd_GET_INSTR, wb_Data_fwd, wb_Data_fwd_S

ack_B_up_last, ack_data_f

L1_GETX

ack_data_f, FWD_Inv

ack_B, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L2_Fwd_GET_INSTR,
L2_Fwd_GETS, L2_Fwd_GETX, L2_Fwd_Upgrade,
L2_Replacement_clean, wb_Data_fwd, wb_Data_fwd_S

ack_B, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETX,
L1_PUTX, L1_PUTX_old, L2_Fwd_GETS,
L2_Fwd_GET_INSTR, L2_Fwd_GETX,
L2_Replacement, L2_Replacement_clean

ack_B, ack_B_all, ack_B_all_token,
ack_Pri, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETX,
L1_PUTX_old, L1_PUTX, L2_Fwd_GETS,
L2_Fwd_GET_INSTR, L2_Fwd_GETX, L2_Replacement,
L2_Replacement_clean, WB_Ack, wb_Data_fwd, wb_Data_fwd_S

ack_data

ack_B_all, ack_B_all_token, ack_Pri

ISS_B

ack_B_up, ack_data

ack_B, ack_Pri, Fwd_Putx,
Fwd_Putx_old, L1_GETS, L1_GET_INSTR,
L1_GETX, L1_UPGRADE, L1_PUTX_old,
L1_PUTX, L2_Fwd_GET_INSTR, L2_Fwd_GETS,
L2_Fwd_GETX, L2_Fwd_Upgrade, L2_Replacement_clean,
WB_Ack, wb_Data_fwd, wb_Data_fwd_S, WB_data_w

ack_B_up_last, ack_data_f

NP

L1_GETS

ISS

Mem_Data

L1_GETS, L1_GET_INSTR

L1_GETS, L1_GET_INSTR

ack_B_all, ack_B_all_token,
ack_B_last, WB_Ack_all, WB_Ack_data

NP_B

ack_B_up, ack_data

L1_GET_INSTR

ack_B, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETS,
L1_GET_INSTR, L1_GETX, L1_PUTX,
L1_PUTX_old, L2_Fwd_GETS, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean

IS

ack_B, ack_B_all, ack_B_all_token,
ack_Pri, dir_Inv, Fwd_GETX,
Fwd_Putx, Fwd_Putx_old, L1_GETS,
L1_GET_INSTR, L1_GETX, L1_PUTX_old,
L1_PUTX, L2_Fwd_GETS, L2_Fwd_GET_INSTR,
L2_Fwd_GETX, L2_Replacement, L2_Replacement_clean,
WB_Ack, WB_Ack_all, wb_Data_fwd, wb_Data_fwd_S

ack_data

ack_B_up_last, ack_data_f

ack_B_all, ack_B_all_token, ack_Pri

IS_B

ack_data_f, FWD_Inv

L1_PUTX, L1_PUTX_old

Machine d'état du cache L2

Tableau de transition associé au cache L2 (1/4)

	State	L1 request			L2 action										Other L2					
		GETS/ GET_INSTR	GETX	Upgrade	Eviction	Replacement (clean data)	Ack_B	ACK_B_all	ack_B_last	ack_B_up	Ack_data	Ack_data_f	ack_Pri	Ack_B_all_token	ack_B_w	L2_Fwd_GETX	L2_Fwd GETS	L2_Fwd_GET_INSTR	L2_Fwd_Upgrade	Fwd_GETX	FWD_Inv
Etats stables	**NP**	Broadcast Fwd_gets → IS/ISS	Broadcast Fwd_getx → IM													Fwd_ ack to req	Fwd_ ack to req	Fwd_ ack to req	Fwd_ ack to req	Fwd_ ack to req	
	NP_B	stall	stall	stall			Update ack-number	Dealloc l2$ → NP	Dealloc l2$ → NP				Update ack-number	Dealloc l2$ → NP		Fwd_ ack to req					
	SS	Send data to L1 requestor & add sharer → SS	Send data→req Invalid sharers →SS_MB	Invalid sharers & ack→upgrader →SS_MB	Invalid sharers & deallocate l2 → S_I	Invalid sharers & deallocate l2 → I_I					Update sharer list → SS					fwd_data to target & inv sharer → SS_MB	fwd_data to target & add sharer → MT_MB	fwd_data to target & add sharer	Ack to upgrader & inv sharer & Ack to req(target) → SS_MB		
	M	Send dataExc to req → MT_MB data→ req & add sharer →SS	data→ req → MT_MB		Ex-replacement & deallocate l2→ M_I	Ex-clean replacement & deallocate l2→ M_I										fwd_data to target(without write) → MT_MB	fwd_data_exc to target → MT_MB	fwd_data to target →MT_SB			
	MT	Fwd req to L1 exc → MT_IIB	Fwd req to L1 exc → MT_MB		Invalid sharers & deallocate l2 →MT_I	Invalid sharers & deallocate l2 →MCT_I										fwd_req_to exclusive → MT_MBB	fwd_req_to exclusive → MT_IBB	fwd_req_to exclusive → MT_IBB		fwd_req_to exclusive → MT_MBB	
Etats liés aux requêtes mémoires	**IS** (read_miss)	Add L1 sharer & record gets_id	stall		stall	stall	Update ack-number	Fetch to mem → IS_B		Update ack-number → NP_B	Activ flag data & send data to gets_id& update ack-number → NP_B	send data to gets_id & dealloc ate l2$ → NP		Update token & fetch to mem → IS_B		Send ack_pri to target &upt token	Send ack_pri to target &upt token	Send ack_pri to target &upt token		stall	
	IS_B	Add L1 sharer & record gets_id	stall		stall	stall	Update ack-number	Fetch to mem						Update token & fetch to mem	update ack	stall	Send ack_pri to target &upt token	Send ack_pri to target &upt token		stall	Dealloc tbe → NP
	ISS	Add L1 sharer & record gets_id → IS	stall		stall	stall	Update ack-number	Fetch to mem → ISS_B		Update ack-number → NP_B	Activ flag data & send data to gets_id& update ack-number → NP_B	send data to gets_id & dealloc ate l2$ → NP		Update token & fetch to mem → ISS_B		Send ack_pri to target &upt token	Send ack_pri to target &upt token	Send ack_pri to target &upt token		stall	
	ISS_B	Add L1 sharer & record gets_id → IS_B	stall		stall	stall	Update ack-number	Fetch to mem						Update token & fetch to mem	update ack	stall	Send ack_pri to target &upt token	Send ack_pri to target &upt token		stall	Dealloc tbe → NP
	IM (write_miss)	stall	stall		stall	stall	Update ack-number	Fetch to mem → IM_B		Update ack-number → NP_B	Activ flag data & send data to getx_id & update ack-number → NP_B	send data to getx_id & dealloc ate l2$ → NP	Nnnnnn/update ack &token → IM_B	Update token & fetch to mem → IM_B		Send ack_pri to target &upt token	Send ack_pri to target &upt token	Send ack_pri to target &upt token		stall	
	IM_B	stall	stall		stall	stall		Fetch to mem → IM_B							update ack	stall				stall	Dealloc tbe → NP

Tableau de transition associé au cache L2 (2/4)

	State	L1 request			L2 action											Other L2					
		GETS/ GET_INSTR	GETX	Upgrade	Eviction	Replacement (clean data)	Ack_B	ACK_ B_all	ack_B_ last	ack_B_ up	Ack_data/ Ack_data_S	Ack_ data_f	ack_Pri	Ack_B_all_ token	ack_ B_w	L2_Fwd_ GETX	L2_Fwd GETS	L2_Fwd_GET_ INSTR	L2_Fwd_ Upgrade	Fwd_GE TX	FWD_Inv
Etats liés s à l'évincement	M_I (evicted)	stall														Stall				stall	
	MT_I (evicted)																				
	MCT_I (evicted)																			stall	
	I_I (evicted)																				
	S_I (evicted)																				
Etats bloquants	SS_MB	stall			stall						Update sharer list → S					stall					
	MT_MB										Write data to $→ M Update sharer list → S									stall	
	MT_IIB																				
	MT_SB										Update sharer list → S										
	MT_IB																				
	MT_MBB																				
	MT_IBB																				

Tableau de transition associé au cache L2 (3/4)

	State	From Mem							From L1								
		Mem_Data	Wb_Data_fwd/ Wb_Data_fwd_S	Dir Inv	wb_ack (replace)	wb_ack_all (replace)	wb_ack_data (replace)	wb_data_ w	Ack_all	Ack	L1-putx (replacing data)	L1-putx-old	Fwd_Putx/ Fwd_Putx_old	wb_data	Wb_data_cle an	Exc_unblo ck	unblock
Etats stables	**NP**		Send to Mem	Pop incoming resp queue							Broadcast Fwd_putx → NP_B	Broadcast Fwd_putx → NP_B	send wb-ack to req (L2 target)				
	NP_B		Send to Mem	Pop incoming resp queue	Update ack	Send wb_ack to getx_id → NP	Send wb_ack to getx_id & re-send data → NP	Updte ack & re-send data			stall	stall	send wb-ack to req (L2 target)				
	SS		Invalid sharer & fwd dataexc →SS_MB Send data to target →SS_MB	Invalid sharers & deallocate l2 → S_I							send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)				Add to sharer list
	M		Send data to target →MT_SB	Ex-replacement & deallocatel2→ M_I							send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)				
	MT		Invalid previous owner →MT_MB Invalid previous owner →MT_IIB	Invalid sharers & deallocatel2 →MT_I							clear L1 sharers & write data & send wb-ack to requestor → M	send wb-ack to requestor	Rmv L1 sharers & send wb-ack to requestor → MT_MB send wb-ack to requestor				
Etats liés aux requêtes mémoires	**IS** (read_miss)	Data to gets_id & write data to $ → SS	recycle	recycle							send wb-ack to requestor	stall	send wb-ack to req (L2 target)				
	IS_B		recycle	recycle	Updte ack	Send wb_ack to getx_id → NP					stall	stall	send wb-ack to req (L2 target)				
	ISS	Data_Exc to gets_id & write data to $ → MT-MB		recycle							send wb-ack to requestor	stall	send wb-ack to req (L2 target)				
	ISS_B		recycle	recycle	Updte ack						stall	stall	send wb-ack to req (L2 target)				
	IM (write_miss)	DATA to getx_id & write data to $ → MT-MB	recycle	recycle	Updte ack						send wb-ack to requestor	stall	send wb-ack to req (L2 target)				
	IM_B		recycle	recycle							stall	stall	send wb-ack to req (L2 target)				
Etats liés s à l'évincement	**M_I** (evicted)		recycle	Pop incoming resp queue	Deallocate tbe → NP						send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)				
	MT_I (evicted)		recycle	Pop incoming resp queue					Ex-replac from tbe →M_I		stall	stall	stall	Write data to TBE & Ex-replac from tbe → M_I	Ex-replac from tbe → M_I		
	MCT_I (evicted)		recycle	Pop incoming resp queue					Ex-clean replacement → M_I		stall	stall	stall	Write data to TBE &Ex-replac from tbe → M_I	Ex-clean replacement → M_I		
	I_I (evicted)		recycle	Pop incoming resp queue					Ex-clean replacement → M_I	Update ack	send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)				

Tableau de transition associé au cache L2 (4/4)

State	From Mem							From L1								
	Mem_Data	Wb_Data_fwd/ Wb_Data_fwd_S	Dir Inv	wb_ack (replace)	wb_ack_all (replace)	wb_ack_data (replace)	wb_data_w	Ack_all	Ack	L1-putx (replacing data)	L1-putx-old	Fwd_Putx/ Fwd_Putx_old	wb_data	Wb_data_clean	Exc_unblock	unblock
S_I (evicted)		recycle	Pop incoming resp queue					Ex-replac from tbe →M_I	Update ack	send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)				
SS_MB		recycle	recycle							stall	stall	stall			Set exc owner → MT	Add to sharer list→ SS
MT_MB		recycle	recycle					Write data to $ & Fwd_inv to L2		stall	stall	stall	Write data to $ & inv L2 that allocatethe same block	Write data to $ & inv L2 that allocatethe same block	Set exc owner → MT	
MT_IIB		recycle	recycle							stall	stall	stall	Write data to $ → MT_SB	Write data to $ → MT_SB		Add to sharer list→ MT_IB
MT_SB		recycle	recycle							send wb-ack to requestor	send wb-ack to requestor	stall			Set Exc owner →MT	Add to sharer list →SS
MT_IB		recycle	recycle							send wb-ack to requestor	send wb-ack to requestor	send wb-ack to req (L2 target)	Write data to $ →SS	Write data to $ →SS		
MT_MBB		recycle	recycle							stall	stall	stall	Write data to $ & fwd to target (data found) → MT_MB	Write data to $ & fwd to target (data found) → MT_MB		
MT_IBB		recycle	recycle							stall	stall	stall	Write data to $ & fwd to target (data found) → MT_SB	Write data to $ & fwd to target (data found) → MT_SB		

Etats bloquants (SS_MB – MT_IBB)

Références

[AHS+15] A. Arora, M. Harne, H. Sultan, A. Bagaria, and S. R. Sarangi. Fp-nuca : A fast noc layer for implementing large nuca caches. *IEEE Transactions on Parallel and Distributed Systems*, pages 2465–2478, Sept 2015.

[AJM12] A. Abousamra, A. K. Jones, and R. Melhem. Codesign of noc and cache organization for reducing access latency in chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, (6) :1038–1046, June 2012.

[AKPJ09] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha. Garnet : A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 33–42, April 2009.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, pages 1–7, August 2011.

[BBE+13] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani. Data center network virtualization : A survey. *IEEE Communications Surveys Tutorials*, pages 909–928, Second 2013.

[BGC+07] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority : Noc based distributed cache coherency. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 117–126, May 2007.

[BJM11] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. *Multi-Core Cache Hierarchies*. Morgan & Claypool Publishers, 2011.

[BMW06] B. M. Beckmann, M. R. Marty, and D. A. Wood. Asr : Adaptive selective replication for cmp caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 443–454, Dec 2006.

[BPG05] Stefan Bieschewski, Joan-Manuel Parcerisa, and Antonio González. Memory bank predictors. In *23rd International Conference on Computer Design (ICCD 2005), 2-5 October 2005, San Jose, CA, USA*, pages 666–670, 2005.

[BW04] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 319–330, Dec 2004.

[CA95] C. M. Cunningham and D. R. Avresky. Fault-tolerant adaptive routing for two-dimensional meshes. In *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, pages 122–131, 1995.

[CCW+09] Shu-Hsuan Chou, Chien-Chih Chen, Chi-Neng Wen, Yi-Chao Chan, Tien-Fu Chen, Chao-Ching Wang, and Jinn-Shyan Wang. No cache-coherence : A single-cycle ring interconnection for multi-core l1-nuca sharing on 3d chips. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 587–592, July 2009.

[CF78] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27 :1112–1118, 1978.

[CGS99] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture : A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[CPK+13] C. H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L. S. Peh. Smart : A single-cycle reconfigurable noc for soc applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 338–343, March 2013.

[CPV03] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 55–66, Dec 2003.

[CPV05] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 357–368, June 2005.

[CRM07] Michael Chu, Rajiv Ravindran, and Scott Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 369–380, 2007.

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. *SIGARCH Comput. Archit. News*, pages 264–276, May 2006.

[CS07] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412, 2007.

[DK15] S. Das and H. K. Kapoor. Exploration of migration and replacement policies for dynamic nuca over tiled cmps. In *2015 28th International Conference on VLSI Design*, pages 141–146, Jan 2015.

[DS07] Haakon Dybdahl and Per Stenström. An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In *13th International Symposium on High Performance Computer Architecture*. IEEE, 2007.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[EPS06] N. Eisley, L. S. Peh, and L. Shang. In-network cache coherence. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 321–332, Dec 2006.

[FMP05] P. Foglia, D. Mangano, and C. A. Prete. A nuca model for embedded systems cache design. In *3rd Workshop on Embedded Systems for Real-Time Multimedia, 2005.*, pages 41–46, Sept 2005.

[FRD08a] J. Flich, S. Rodrigo, and J. Duato. An efficient implementation of distributed routing algorithms for nocs. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 87–96, April 2008.

[FRD+08b] J. Flich, S. Rodrigo, J. Duato, T. Sødring, Å G. Solheim, T. Skeie, and O. Lysne. On the potential of noc virtualization for multicore chips. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 801–807, March 2008.

[GG00] Pierre Guerrier and Alain Greiner. Architecture for on-chip packet-switched interconnections. In *Proc. of the Design Automation and Test in Europe Conference*, pages 250–256, Paris, France, March 2000.

[GK10] Paul Gratz and Stephen W Keckler. Realistic workload characterization and analysis for networks-on-chip design. In *4th workshop on chip multiprocessor memory systems and interconnects*, pages 477–484, Jan 2010.

[GKM+06] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger. Implementation and evaluation of on-chip network architectures. In *2006 International Conference on Computer Design*, pages 477–484, Oct 2006.

[GN92] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 278–287, 1992.

[HBH12] F. Hameed, L. Bauer, and J. Henkel. Dynamic cache management in multi-core architectures through run-time adaptation. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 485–490, March 2012.

[HCM09] Mohammad Hammoud, Sangyeun Cho, and Rami G. Melhem. ACM : an efficient approach for managing shared caches in chip multiprocessors. In *High Performance Embedded Architectures and Compilers, Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings*, pages 355–372, 2009.

[HFFA09] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive nuca : Near-optimal block placement and replication in distributed caches. *SIGARCH Comput. Archit. News*, pages 184–195, June 2009.

[HGC08] Enric Herrero, José González, and Ramon Canal. Distributed cooperative caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 134–143, 2008.

[HGC10a] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching : An autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 419–428, 2010.

[HGC10b] Enric Herrero, José González, and Ramon Canal. Power-efficient spilling techniques for chip multiprocessors. In *16th International Euro-Par Conference on Parallel processing : Part I, Ischia, Italy*, pages 256–267, 2010.

[HGG+12] A. Huang, J. Gao, W. Guo, W. Shi, M. Zhang, and J. Jiang. Psa-nuca : A pressure self-adapting dynamic non-uniform cache architecture. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, pages 181–188, June 2012.

[HGP+03] J.L. Hennessy, D. Goldberg, D.A. Patterson, D. Etiemble, and K. Asanovic. *Architecture des ordinateurs : une approche quantitative*. Les Classiques de l'informatique. Vuibert, 2003.

[HKS+05] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, pages 31–40, 2005.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[JKY07] Y. Jin, E. J. Kim, and K. H. Yum. A domain-specific on-chip network design for large scale cache systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 318–327, Feb 2007.

[JPL08] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Circuit-switched coherence. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 193–202. IEEE Computer Society, 2008.

[KBK02] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, pages 211–222, October 2002.

[KBM+10] N. A. Kurd, S. Bhamidipati, C. Mozak, J. L. Miller, T. M. Wilson, M. Nemani, and M. Chowdhury. Westmere : A family of 32nm ia processors. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 96–97, Feb 2010.

[KKC+08] Tushar Krishna, Amit Kumar, Patrick Chiang, Mattan Erez, and Li-Shiuan Peh. Noc with near-ideal express virtual channels using global-line communication. In *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, HOTI '08, pages 11–20. IEEE Computer Society, 2008.

[KKP14] Woo-Cheol Kwon, Tushar Krishna, and Li-Shiuan Peh. Locality-oblivious cache organization leveraging single-cycle multi-hop nocs. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 715–728. ACM, 2014.

[KLIS08] M. Kandemir, F. Li, M. J. Irwin, and S. W. Son. A novel migration-based nuca design for chip multiprocessors. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2008.

[KPKJ07] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K. Jha. Express virtual channels : Towards the ideal interconnection fabric. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 150–161. ACM, 2007.

[LCC11] H. Lee, S. Cho, and B. R. Childers. Cloudcache : Expanding and shrinking private caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 219–230, Feb 2011.

[LF13] M. Lodde and J. Flich. An noc and cache hierarchy substrate to address effective virtualization and fault-tolerance. In *Networks on Chip (NoCS), 2013 Seventh IEEE/ACM International Symposium on*, pages 1–8, April 2013.

[LFA12] M. Lodde, J. Flich, and M. E. Acacio. Heterogeneous noc design for efficient broadcast-based coherence protocol support. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 59–66, May 2012.

[LFA13] Mario Lodde, José Flich, and Manuel E Acacio. Towards efficient dynamic llc home bank mapping with noc-level support. In *European Conference on Parallel Processing*, pages 178–190. Springer, 2013.

[LJ13] C. Y. Lee and N. K. Jha. Variable-pipeline-stage router. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1669–1682, Sept 2013.

[LTL+13] G. Li, O. Temam, Z. Liu, D. Wang, S. Guo, and C. Li. Cluster cache monitor. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*, pages 1–8, Oct 2013.

[LYB96] Yen-Wen Lu, G. K. Yeh, and J. B. Burr. A 15 mw 1.6 gb/s wormhole data router for 2-d meshes. In *1996 Symposium on VLSI Circuits. Digest of Technical Papers*, pages 138–139, June 1996.

[MB07] Naveen Muralimanohar and Rajeev Balasubramonian. Interconnect design considerations for large nuca caches. *SIGARCH Comput. Archit. News*, pages 369–380, June 2007.

[MBJ07] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, Dec 2007.

[MFD+07] A. Mejia, J. Flich, J. Duato, S. A. Reinemo, and T. Skeie. Boosting ethernet performance by segment-based routing. In *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, pages 55–62, Feb 2007.

[MH07] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. *SIGARCH Comput. Archit. News*, pages 46–56, June 2007.

[MPF+09] Andres Mejia, Maurizio Palesi, Jose Flich, Shashi Kumar, Pedro López, Rickard Holsmark, and José Duato. Region-based routing : A mechanism to support efficient routing algorithms in nocs. *IEEE Trans. VLSI Syst.*, pages 356–369, 2009.

[MPG10] J. Merino, V. Puente, and J. A. Gregorio. Esp-nuca : A low-cost adaptive non-uniform cache architecture. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10, Jan 2010.

[MPPG08] Javier Merino, Valentín Puente, Pablo Prieto, and José Ángel Gregorio. Sp-nuca : A cost effective dynamic non-uniform cache architecture. *SIGARCH Comput. Archit. News*, pages 64–71, May 2008.

[MTCM05] A. Mello, L. Tedesco, N. Calazans, and F. Moraes. Virtual channels in networks on chip : Implementation and evaluation on hermes noc. In *2005 18th Symposium on Integrated Circuits and Systems Design*, pages 178–183, Sept 2005.

[MWM06] Robert Mullins, Andrew West, and Simon Moore. The design and implementation of a low-latency on-chip network. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 164–169. IEEE Press, 2006.

[OSV+14] M. Ortin, D. Suârez, M. Villarroya, C. Izu, and V. Vinals. Dynamic construction of circuits for reactive traffic in homogeneous cmps. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, pages 412–421, July 1974.

[PGJ+05] Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, pages 1025–1040, Aug 2005.

[PKH06] Maurizio Palesi, Shashi Kumar, and Rickard Holsmark. A method for router table compression for application specific routing in mesh topology noc architectures. In *Embedded Computer Systems : Architectures, Modeling, and Simulation, 6th International Workshop, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, pages 373–384, 2006.

[PPCR12] Francesca Palumbo, Danilo Pani, Andrea Congiu, and Luigi Raffo. Concurrent hybrid switching for massively parallel systems-on-chip : The cyber architecture. In *Proceedings of the 9th Conference on Computing Frontiers*, pages 173–182. ACM, 2012.

[RBGB06] Robert Ricci, Steve Barrus, Dan Gebhardt, and Rajeev Balasubramonian. Leveraging bloom filters for smart search within nuca caches. In *7th Workshop on Complexity-Effective Design (WCED)*, 2006.

[Roq17] Steve Roques. Formal verification of a new cache coherence protocol. Master's thesis, MOSiG Master, Université Grenoble Alpes, 2017.

[SCK+12] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 201–210. IEEE, 2012.

[SDM+12] D. Suárez, G. Dimitrakopoulos, T. Monreal, M. G. H. Katevenis, and V. V. Yufera. Lp-nuca : Networks-in-cache for high-performance low-power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1510–1523, Aug 2012.

[SHZG05] M. K. F. Schafer, T. Hollstein, H. Zimmer, and M. Glesner. Deadlock-free routing and component placement for irregular mesh-based networks-on-chip. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 238–245, Nov 2005.

[Sli] SLICC Language. Available : http ://gem5.org/SLICC.

[SMBM10] C. Seiculescu, S. Murali, L. Benini, and G. De Micheli. A method to remove deadlocks in networks-on-chips with wormhole flow control. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1625–1628, March 2010.

[SMV+09] Darío Suárez, Teresa Monreal, Fernando Vallejo, Ramón Beivide, and Víctor Viñals. Light nuca : A proposal for bridging the inter-cache latency gap. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 530–535, 2009.

[SS08] M. B. Stensgaard and J. Sparsø. Renoc : A network-on-chip architecture with reconfigurable topology. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 55–64, April 2008.

[SSJR+09] T. Skeie, F. O. Sem-Jacobsen, S. Rodrigo, J. Flich, D. Bertozzi, and S. Medardoni. Flexible dor routing for virtualization of multicore chips. In *System-on-Chip, 2009. SOC 2009. International Symposium on*, pages 073–076, Oct 2009.

[SSP+07] Xudong Shi, Feiqi Su, Jih-kwon Peir, Ye Xia, and Zhen Yang. Cmp cache performance projection : Accessibility vs. capacity. *SIGARCH Comput. Archit. News*, pages 13–20, March 2007.

[SZL+10] Guang Sun, Yuanyuan Zhang, Yong Li, Li Su, Depeng Jin, and Lieguang Zeng. Convex-based dor routing for virtualization of noc. In *Proceedings of the 2010 IFIP International Conference on Network and Parallel Computing*, pages 462–469, 2010.

[TKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor : A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2) :25–35, 2002.

[TMG+05] L. Tedesco, A. Mello, D. Garibotti, N. Calazans, and F. Moraes. Traffic generation and performance evaluation for mesh-based nocs. In *2005 18th Symposium on Integrated Circuits and Systems Design*, pages 184–189, Sept 2005.

[TSAF11] Francisco Triviño, José L. Sánchez, Francisco José Alfaro, and José Flich. Virtualizing network-on-chip resources in chip-multiprocessors. *Microprocessors and Microsystems - Embedded Hardware Design*, pages 230–245, 2011.

[VSG+12] S. Volos, C. Seiculescu, B. Grot, N. K. Pour, B. Falsafi, and G. De Micheli. Ccnoc : Specializing on-chip interconnects for energy efficiency in cache-coherent servers. In *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, pages 67–74, May 2012.

[WCK08] I. Walter, I. Cidon, and A. Kolodny. Benoc : A bus-enhanced network on-chip for a power efficient cmp. *IEEE Computer Architecture Letters*, 7(2) :61–64, July 2008.

[WGP+09] A. Y. Weldezion, M. Grange, D. Pamunuwa, Z. Lu, A. Jantsch, R. Weerasekera, and H. Tenhunen. Scalability of network-on-chip communication architecture for 3-d meshes. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 114–123, May 2009.

[WM] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall : Implications of the obvious. *SIGARCH Comput. Archit. News*, pages 20–24.

[WWF+06] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. Simflex : Statistical sampling of computer system simulation. *IEEE Micro*, pages 18–31, 2006.

[YW99] Yuanyuan Yang and Jianchao Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 290–299, Jan 1999.

[ZA05a] M. Zhang and K. Asanovic. Victim replication : maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 336–345, June 2005.

[ZA05b] Michael Zhang and Krste Asanović. Victim migration : Dynamically adapting between private and shared cmp caches. Technical report, 2005.

[ZGB10] Z. Zhang, A. Greiner, and M. Benabdenbi. Fully distributed initialization procedure for a 2d-mesh noc, including off-line bist and partial deactivation of faulty components. In *2010 IEEE 16th International On-Line Testing Symposium*, pages 194–196, July 2010.

[ZGT08] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for a fault-tolerant 2d-mesh network-on-chip. In *Proceedings of the 45th annual Design Automation Conference*, pages 441–446. ACM, 2008.

[ZIUN08] Li Zhao, Ravi Iyer, Mike Upton, and Don Newell. Towards hybrid last level caches for chip-multiprocessors. *SIGARCH Computer Architecture News*, pages 56–63, 2008.

Hiérarchie mémoire dans les systèmes intégrés multiprocesseurs construits autour de réseaux sur puce

Résumé – Les systèmes parallèles de type multi/pluri-cœurs permettant d'obtenir une grande puissance de calcul à bas coût énergétique sont de nos jours une réalité. Néanmoins, l'exploitation des performances de ces architectures dépend de l'efficacité du système à gérer les accès aux données. Le but de nos travaux est d'améliorer l'efficacité de ces accès en exploitant les caractéristiques de l'architecture matérielle.

Dans une première partie, nous proposons une nouvelle organisation de la hiérarchie des mémoires caches qui maximise l'utilisation de l'espace de mémorisation disponible à chaque niveau. Cette solution, basée sur les architectures à accès non uniforme au cache (NUCA), supporte les transferts inter et intra-niveau de la hiérarchie. Elle requiert un protocole de cohérence de caches qui s'adapte à ses spécifications.

Certes, le transfert des données au niveau de la hiérarchie est aussi un déterminant de la performance du système. Dans une seconde partie, nous prenons en compte les besoins de communication spécifiques du protocole. Nous proposons un réseau virtualisé comme support de communication *ad-hoc* afin de gérer le trafic de cohérence à moindre coût. Ce dernier relie les caches d'un même niveau pour supporter les transferts intra-niveaux, qui sont une spécificité de notre protocole, en vue de réduire la latence moyenne d'accès.

Mots-clés *: Système Multiprocesseur sur Puce (MPSoC), Hiérarchie mémoire, Cohérence, Réseau sur Puce (NoC), Non Uniform Cache Access (NUCA)*

Memory hierarchy in embedded multiprocessor system built around networks on chip

Abstract – Multi/many-cores parallel systems for high-power computing at low energy costs are nowadays a reality. However, exploiting the performance of these architectures depends on the efficiency of the system in managing data accesses. The aim of our work is to improve the efficiency of these accesses by exploiting the hardware architecture characteristics.

In a first part, we propose a new cache hierarchy organization that aims at maximizing the use of the available storage space at each level. This solution, based on non-uniform cache access architectures (NUCA), supports inter and intra-level transfers of the hierarchy. It requires a cache coherency protocol that suits its specifications.

Obviously, the transfer of data in the hierarchy is also a determinant of the system performance. In a second part, we consider the specific communication needs of the protocol. We suggest the use of a virtualized network as an *ad-hoc* communication medium to manage consistency traffic at a lower cost. It links the caches of the same level to support intra-level transfers, which are a specificity of our protocol, in order to reduce the average access latency.

Keywords *: Multi-Processor System-On-Chip (MPSoC), Memory hierarchy, Consistency, Network-on-Chip (NoC), Non Uniform Cache Access (NUCA)*

Thèse préparée au laboratoire TIMA
46 Avenue Félix Viallet, 38031, Grenoble Cedex, France
ISBN : 978-2-11-129231-4