Représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence de cache

Julie Dumas

► To cite this version:

Julie Dumas. Représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence de cache. Architectures Matérielles [cs.AR]. Université Grenoble Alpes, 2017. Français. NNT : 2017GREAM093 . tel-01879032

HAL Id: tel-01879032
https://theses.hal.science/tel-01879032

Submitted on 21 Sep 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 DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Julie DUMAS

Thèse dirigée par **Frédéric PÉTROT**

préparée au sein du **laboratoire TIMA et au CEA-Leti**
et de l'École Doctorale **Mathématiques, Sciences et Technologies de l'In-formation, Informatique**

Représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence de cache

Thèse soutenue publiquement le **13 décembre 2017**
devant le jury composé de :

M. Florent de Dinechin
INSA Lyon/CITI, Président
M. Daniel Etiemble
Université Paris Sud/LRI, Rapporteur
M. Gilles Sassatelli
CNRS/Université Montpellier 2/LIRMM, Rapporteur
M. Quentin Meunier
UMPC/LIP6, Examinateur
M. Frédéric Pétrot
TIMA, Directeur de thèse
M. Éric Guthmuller
CEA-Leti, Co-Encadrant de thèse

Résumé

Le problème du passage à l'échelle des protocoles de cohérence de caches qui se pose pour les machines parallèles se pose maintenant aussi sur puce, suite à l'émergence des architectures *manycores*. Il existe deux classes de protocoles : ceux basés sur l'espionnage et ceux utilisant un répertoire. Les protocoles basés sur l'espionnage, qui doivent transmettre à tous les caches les informations de cohérence, engendrent un nombre important de messages dont peu sont effectivement utiles. En revanche, les protocoles avec répertoires visent à n'envoyer des messages qu'aux caches qui en ont besoin. L'implémentation la plus évidente utilise un champ de bits complet dont la taille dépend uniquement du nombre de cœurs. Ce champ de bits représente la liste des copies. Pour passer à l'échelle, un protocole doit émettre un nombre raisonnable de messages de cohérence et limiter le matériel utilisé pour la cohérence et en particulier pour la liste des copies.

Afin d'évaluer et de comparer les protocoles et leurs représentations de la liste des copies, nous proposons tout d'abord une méthode de simulation basée sur l'injection de traces dans un modèle de cache à haut niveau. Cette méthode permet d'effectuer rapidement l'exploration architecturale des protocoles.

Nous proposons également une nouvelle représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence de caches. Pour une architecture à 64 cœurs, 93% des lignes de cache sont partagées par au maximum 8 cœurs, sachant par ailleurs que le système d'exploitation cherche à placer les tâches communicantes proches les unes des autres. Notre représentation dynamique de la liste des copies tire parti de ces deux observations en utilisant un champ de bits pour un sous-ensemble des copies et une liste chaînée. Le champ de bits correspond à un rectangle à l'intérieur duquel la liste des copies est exacte. La position et la forme de ce rectangle évoluent au cours de la durée de vie des applications. Plusieurs algorithmes pour le placement du rectangle cohérent sont proposés et évalués. Pour finir, nous effectuons une comparaison avec les représentations de la liste des copies de l'état de l'art.

Abstract

Cache coherence protocol scalability problem for parallel architecture is also a problem for on chip architecture, following the emergence of manycores architectures. There are two protocol classes : snooping and directory-based. Protocols based on snooping, which send coherence information to all caches, generate a lot of messages whose few are useful. On the other hand, directory-based protocols send messages only to caches which need them. The most obvious implementation uses a full bit vector whose size depends only on the number of cores. This bit vector represents the sharing set. To scale, a coherence protocol must produce a reasonable number of messages and limit hardware ressources used by the coherence and in particular for the sharing set.

To evaluate and compare protocols and their sharing set, we first propose a method based on trace injection in a high-level cache model. This method enables a very fast architectural exploration of cache coherence protocols.

We also propose a new dynamic sharing set for cache coherence protocols, which is scalable. With 64 cores, 93% of cache blocks are shared by up to 8 cores. Futhermore, knowing that the operating system looks to place communicating tasks close to each other. Our dynamic sharing set takes advantage from these two observations by using a bit vector for a subset of copies and a linked list. The bit vector corresponds to a rectangle which stores the exact sharing set. The position and shape of this rectangle evolve over application's lifetime. Several algorithms for coherent rectangle placement are proposed and evaluated. Finally, we make a comparison with sharing sets from the state of the art.

Remerciements

Je voudrais tout d'abord remercier mon directeur de thèse, Frédéric Pétrot, pour sa disponibilité et son implication en particulier lors des phases de rédaction. Je tiens à remercier mon encadrant au CEA, Éric Guthmuller, pour son aide lors du développement de PyCCExplorer, ses relectures, sa patience et surtout sa disponibilité. Je les remercie également d'avoir toujours fait en sorte de ne pas se contredire, et ainsi d'avoir eu un encadrement cohérent.

Je tiens également à remercier les membres de mon jury. Tout d'abord, Daniel Etiemble et Gilles Sassatelli qui ont accepté d'être mes rapporteurs. Je remercie également Florent de Dinechin et Quentin Meunier qui ont accepté de participer à mon jury de thèse. Je les remercie pour les échanges que nous avons pu avoir sur mes travaux de thèse.

Je souhaite également remercier l'ensemble du laboratoire LISAN pour leur accueil et en particulier Fabien Clermidy puis Jerôme Martin chefs du laboratoire au cours de ma thèse. Je remercie en particulier les autres doctorants que j'ai croisés au cours de ma thèse : Vincent, Florent, Maxime (les deux), Alex, Andréa, Réda, Thomas, Jimmy (et les repas du jeudi), Camille, Simon (et ses cannelés) et surtout Soundous. Je tiens à remercie particulièrement César Fuguet Tortolero pour son aide et ses conseils tout au long de ma thèse. Je remercie également Christian Bernard pour son intérêt à mes travaux. Je souhaite également remercier les responsables scientifiques : Edith Beigné, Marc Belleville et Pascal Vivet pour leurs conseils.

Je voudrais également remercier les membres de l'équipe SLS du laboratoire TIMA avec qui j'ai pu échanger et en particulier Hela avec qui j'ai pu partager mes difficultés avec gem5.

Je remercie également tous les professeurs avec qui j'ai pu intervenir à l'Ensimag : Olivier Muller, Claire Maiza, Matthieu Moy, Stéphane Mancini, François Broquedis, Frédéric Pétrot, Matthieu Chabanas, Sébastien Viardot pour leur soutien et leurs conseils en particulier durant les derniers mois de ma thèse.

Je souhaiterais remercie mes amis du conservatoire et de l'harmonie Echo des Balmes pour m'avoir permis de faire de la musique tout au long de ma thèse, même lorsque je n'avais pas beaucoup de temps. Je remercie également Lorraine de m'avoir soutenue pendant la rédaction. Je souhaite remercier Yannick, pour les chocolats chauds et le festival

du film d'animation à Annecy. Je remercie également ma famille, et particulièrement mes parents. Enfin, je remercie Thomas pour son soutien et sa compréhension.

Table des matières

Table des figures xi

Liste des tableaux xv

1 Introduction 1

2 Problématique 3

2.1 Architectures *manycores* . . . 3

2.1.1 Modèle mémoire . . . 3

2.1.2 Hiérarchie mémoire . . . 5

2.2 Caches et cohérence des caches dans les architectures *manycores* . . . 6

2.2.1 Architecture des caches . . . 6

2.2.2 Cohérence des caches . . . 7

2.3 Représentation de la liste des copies . . . 10

2.4 Caractéristiques des applications parallèles . . . 12

2.5 Évaluation des représentations de la liste des copies . . . 13

2.6 Conclusion . . . 14

3 Cohérence des caches dans une architecture *manycore* à mémoire partagée 15

3.1 Protocoles basés sur l'espionnage . . . 15

3.1.1 *Flexible snooping* . . . 16

3.1.2 *In-Network Coherence Filtering* . . . 17

3.2 Protocoles avec répertoire limité . . . 18

3.2.1 Ackwise . . . 18

3.2.2 *Coarse bit-vector directory* . . . 19

3.2.3 Virtualisation des CIDs . . . 19

3.3 Protocoles de cohérence de caches dynamiques . . . 20

3.3.1 Liste chaînée . . . 20

3.3.2 Champ de bits limité . . . 21

3.3.3 *Sponge directory* . . . 22

3.3.4 SPACE . . . 23

3.4 Autres propositions . . . 24
3.4.1 *Scalable Interface Coherence* . . . 24
3.4.2 *In-Network Cache Coherence* . . . 24
3.4.3 Classifications des données . . . 25
3.4.4 Fenêtres temporelles . . . 26
3.5 Conclusion . . . 27

4 Représentation dynamique de la liste des copies 29
4.1 Présentation du concept du protocole DCC . . . 29
4.2 Représentation du répertoire du protocole *Dynamic Coherent Cluster* (DCC) 31
4.2.1 Entrée du répertoire . . . 31
4.2.2 Liste chaînée . . . 32
4.2.3 Exemple d'entrée du répertoire DCC . . . 32
4.2.4 Coût matériel du répertoire . . . 33
4.3 Algorithmes de placement du rectangle cohérent . . . 33
4.3.1 Algorithme **idéal** . . . 34
4.3.2 Algorithme **premier touché** . . . 36
4.3.3 Algorithme **combinatoire** . . . 38
4.4 Détails de l'algorithme **combinatoire** . . . 39
4.4.1 Comportement de l'algorithme **combinatoire** . . . 40
4.4.2 Évaluation théorique du coût matériel . . . 43
4.5 Conclusion . . . 45

5 Méthode d'évaluation de la représentation de la liste des copies 47
5.1 Méthodes de simulation existantes . . . 47
5.1.1 Simulations par interprétation d'instructions . . . 48
5.1.2 Simulations basées sur l'injection de traces . . . 50
5.2 Flot de simulation . . . 51
5.3 Extraction des requêtes d'accès mémoire . . . 52
5.3.1 Choix des paramètres de gem5 . . . 52
5.3.2 Modifications apportées à gem5 . . . 53
5.3.3 Choix des applications . . . 53
5.3.4 Simulation et extraction du trafic . . . 54
5.4 PyCCExplorer : un simulateur haut niveau pour la cohérence de caches . 55
5.4.1 Architecture logicielle . . . 56
5.5 Choix des métriques et instrumentations du simulateur . . . 59
5.5.1 Latence d'accès aux données . . . 59
5.5.2 Trafic induit par la cohérence de caches . . . 60
5.5.3 Évaluation du nombre de messages de diffusion . . . 60

5.6 Conclusion 60

6 Évaluation de l'influence de la représentation de la liste des copies 63

6.1 Expérimentations avec le simulateur PyCCExplorer 63

6.1.1 Implémentation dans le simulateur 64

6.1.1.1 Protocole basé sur l'espionnage (snoop) 64

6.1.1.2 Répertoire avec champ de bits complet (bit-vector) 64

6.1.1.3 Répertoire Ackwise 65

6.1.1.4 Répertoire utilisant une liste chaînée 65

6.1.2 Exploration architecturale avec PyCCExplorer 66

6.1.2.1 Exploration architecturale pour Ackwise 66

6.1.2.2 Exploration architecturale du protocole basé sur une liste chaînée 67

6.1.3 Classement des représentations de la liste des copies 70

6.1.3.1 Latence 70

6.1.3.2 Trafic 74

6.1.4 Validation du simulateur PyCCExplorer 76

6.1.4.1 Comparaison entre PyCCExplorer et gem5 concernant l'évaluation de la représentation de la liste des copies . . 76

6.1.4.2 Comparaison des résultats à ceux d'autres travaux 77

6.1.4.3 Complexité de PyCCExplorer 78

6.2 Évaluation du protocole DCC avec PyCCExplorer et implémentation matérielle 79

6.2.1 Simulation avec PyCCExplorer 79

6.2.1.1 Environnement de simulation 79

6.2.1.2 Étude des paramètres du protocole 80

6.2.1.3 Comparaison des représentations de la liste des copies . 83

6.2.1.4 Conclusion sur l'ensemble des métriques 89

6.2.2 Implémentation matérielle du protocole DCC 89

6.2.2.1 Paramètres de synthèse 89

6.2.2.2 Fréquence de fonctionnement 90

6.2.2.3 Surface du bloc *tiling* 90

6.3 Conclusion 91

7 Travaux futurs 95

7.1 Améliorations à apporter à PyCCExplorer 95

7.1.1 Simulation d'architecture avec plus de 64 cœurs 95

7.1.2 Ajout d'autres topologies de réseau 95

7.1.3 Ajout d'autres représentations de la liste des copies 95

7.1.4 Ajout de méthodes décentralisées 96
7.2 Limites des simulations du protocole DCC 96
7.2.1 Simulation des applications avec l'initialisation 96
7.2.2 Simulation sans le placement optimal des cœurs 96
7.2.3 Simulation avec d'autres architectures 97
7.3 Pistes de recherche autour du protocole DCC 97
7.3.1 Choix du *keeper* 97
7.3.2 Intégration du bloc *tiling* dans une architecture 97
7.3.3 Ajouter un tas partagé pour mémoriser les rectangles de cohérence 97

8 Conclusion **101**

A Algorithme de placement des tâches **107**
A.1 Présentation de l'algorithme de recuit simulé 107
A.2 Placement des caches sur la grille avant et après l'algorithme 109

Publications **113**

Bibliographie **115**

Table des figures

2.1 Architecture à passage de messages 4
2.2 Architecture à mémoire partagée 4
2.3 Pyramide des différents types de mémoire 5
2.4 Exemple avec 4 cœurs partageant des données 8
2.5 Protocole de cohérence de caches basé sur l'espionnage 8
2.6 Protocole de cohérence de caches basé sur un répertoire 9
2.7 Machine à états finis pour un protocole de type MESI 10
2.8 Format d'entrée dans un répertoire avec la liste des copies 11
2.9 Exemple de liste des copies représentée avec un champ de bits complet . 11
2.10 Nombre de lignes de cache partagées par n cœurs 12

3.1 Trois types de communication dans un réseau en anneau 16
3.2 INCF : exemple d'architecture avec des filtres au niveau des routeurs . . . 17
3.3 Répertoire du protocole Ackwise 18
3.4 Représentation de la liste des copies basée sur une liste chaînée 20
3.5 Structure du répertoire utilisant des segments 21
3.6 Formats des entrées de *Sponge directory* 22
3.7 Répertoire SPACE 23
3.8 Création de l'arbre de cohérence du protocole INCC 25
3.9 Répertoire du protocole INCC 25

4.1 Principe de DCC : les étoiles représentent les copies, les coordonnées (x,y) sont dans un plan 2D dont l'origine est le coin en bas à gauche de la grille, et les valeurs entre crochets représentent la position du cœur dans le rectangle 30
4.2 Entrée du répertoire DCC 31
4.3 Exemple d'entrée du répertoire DCC avec le tas 32
4.4 Exemple de rectangle englobant un rectangle de taille inférieure 34
4.5 Nombre de rectangles d'origine et de forme différentes en fonction de l'aire du rectangle pour 64, 256 et 1024 cœurs 35

4.6 Évolution du rectangle de cohérence après chaque accès avec l'algorithme **premier touché** . . . 36
4.7 Exemple de choix de rectangle **premier touché** en vert, **idéal** en rouge . . 37
4.8 Évolution du rectangle de cohérence après chaque accès avec l'algorithme **combinatoire** . . . 39
4.9 Exemple de choix de rectangle **premier touché** en vert, **idéal** en rouge et **combinatoire** en bleu . . . 39
4.10 Bloc *tiling* pour la recherche du rectangle de cohérence . . . 40
4.11 Bloc de calcul des rectangles pour la combinaison [1, 2] . . . 42
4.12 Bloc d'élagage pour une combinaison de copies . . . 42
4.13 Bloc de sélection implémenté avec un encodeur de priorité . . . 43

5.1 Compromis entre temps de simulation et précision pour les différents modes de gem5 extrait de [BGOS12] . . . 49
5.2 Flot de simulation en trois étapes . . . 51
5.3 Architecture simulée sur gem5 avec l'ajout des moniteurs . . . 53
5.4 Architecture logicielle de PyCCExplorer . . . 56

6.1 Latence moyenne relative à Ackwise avec un seuil de 5 pour plusieurs valeurs de seuil . . . 67
6.2 Pourcentage de lignes de cache en mode diffusion avec un tas infini pour la représentation basée sur une liste chaînée . . . 68
6.3 Pourcentage des lignes qui ne sont pas en mode diffusion pour chaque taille de liste chaînée . . . 68
6.4 Nombre d'entrées du tas utilisées pour la représentation avec une liste chaînée (avec un tas infini) . . . 69
6.5 Pourcentage de lignes de cache en mode diffusion pour la représentation de la liste des copies avec et sans limite sur la taille du tas . . . 70
6.6 Latence moyenne en nombre de cycles . . . 71
6.7 Répartition des requêtes de lecture et de lecture exclusive . . . 72
6.8 Répartition du taux de succès/défaut de cache au niveau du *Level Two* (L2) 73
6.9 Moyenne de la répartition de la latence des requêtes de lecture et de lecture exclusive pour l'ensemble des applications . . . 73
6.10 Moyenne et maximum du trafic pour Ackwise, champ de bits, liste chaînée et espionnage . . . 75
6.11 Pourcentage des requêtes incohérentes en entrée de PyCCExplorer et provenant de gem5 . . . 77
6.12 Moyenne de l'utilisation du rectangle de cohérence avec le protocole DCC idéal pour les copies partagées . . . 80

6.13 Pourcentage des lignes de cache en mode diffusion avec le protocole DCC idéal et un tas illimité . . . 82
6.14 Nombre d'entrées du tas utilisées pour mémoriser la liste chaînée de DCC 82
6.15 Moyenne de la latence en nombre de cycles . . . 84
6.16 Moyenne et maximum du trafic . . . 85
6.17 Moyenne du trafic normalisé par rapport au champ de bits . . . 86
6.18 Nombre de messages envoyés en diffusion . . . 88
6.19 Nombre de messages envoyés en diffusion normalisé par rapport à DCC **idéal** . . . 88
6.20 Fréquence de fonctionnement du bloc *tiling* avec des multiplications . . . 90
6.21 Fréquence de fonctionnement du bloc *tiling* avec des *Look-Up Table* (LUT)s 91
6.22 Surface du bloc *tiling* à la fréquence maximale de fonctionnement en fonction du nombre d'entrées . . . 91

7.1 Répertoire DCC avec ajout d'un niveau d'indirection selon si la ligne est privée ou partagée . . . 98
7.2 Répertoire DCC avec ajout d'un niveau d'indirection selon le mode de la ligne . . . 98

A.1 Placement des caches avant et après l'algorithme pour `blackscholes` . 109
A.2 Placement des caches avant et après l'algorithme pour `bodytrack` 109
A.3 Placement des caches avant et après l'algorithme pour `canneal` 110
A.4 Placement des caches avant et après l'algorithme pour `dedup` 110
A.5 Placement des caches avant et après l'algorithme pour `ferret` 110
A.6 Placement des caches avant et après l'algorithme pour `fft` 110
A.7 Placement des caches avant et après l'algorithme pour `fluidanimate` . 111
A.8 Placement des caches avant et après l'algorithme pour `freqmine` 111
A.9 Placement des caches avant et après l'algorithme pour `ocean_cp` 111
A.10 Placement des caches avant et après l'algorithme pour `ocean_ncp` 111
A.11 Placement des caches avant et après l'algorithme pour `streamcluster` 112
A.12 Placement des caches avant et après l'algorithme de recuit pour `vips` . . 112
A.13 Placement des caches avant et après l'algorithme pour `x264` 112

Liste des tableaux

5.1 Paramètres du simulateur gem5 52
5.2 Liste des applications 54
5.3 Temps d'exécution de certaines applications PARSEC et Splash2 sur gem5 55

6.1 Comparaisons des représentations de la liste des copies 89

Chapitre 1: Introduction

Bien que dans son rapport de 2015, l'International Technology Roadmap for Semiconductors (ITRS) prédise la fin de la diminution de la taille des transistors en 2021 [Cou16], d'autres technologies d'intégration, par exemple la 3D, ou encore la photonique intégrée, sont des candidates sérieuses pour continuer d'augmenter la puissance de calcul des ordinateurs, principalement en augmentant le nombre de cœurs interconnectés et en leur adjoignant de la mémoire locale.

En revanche, et même si des progrès notables ont été faits également du coté de la mémoire de masse, ses temps d'accès et débits restent loin de ceux que l'on peut obtenir sur une puce, et le resteront encore durablement à cause des technologies employées pour assurer un faible coût de fabrication. C'est pourquoi les machines actuelles utilisent une hiérarchie mémoire avec des mémoires rapides mais de faible capacité proche des unités de calcul et, plus on s'éloigne, plus la latence augmente mais la capacité augmente également. Aujourd'hui de nombreuses équipes travaillent à améliorer les performances des mémoires.

Par ailleurs, pour répondre aux besoins en calcul de plus en plus grands, la solution généralement admise est de découper les problèmes en plusieurs sous-problèmes et d'effectuer chaque calcul correspondant à un sous-problème sur un processeur différent. Le modèle de programmation sous-jacent le plus largement utilisé reste celui d'une mémoire partagée entre les différentes tâches logicielles, indépendamment des processeurs sur lesquels elles s'exécutent. En conséquence, la mémoire doit être hiérarchisée, c'est-à-dire que les parties les plus utiles de la mémoire de masse doivent être copiées dans des mémoires rapides proches des processeurs appelées « caches », pour satisfaire les besoins en performance à un coût raisonnable.

Les données présentes dans les caches sont donc des copies des données présentes dans la mémoire de masse. Ces copies induisent une difficulté connue depuis toujours sur les machines parallèles à mémoire partagée : la nécessité d'assurer la cohérence des caches. En effet, si une donnée copiée est modifiée localement, il faut que les processeurs qui possèdent une autre copie de cette donnée soient, avant qu'ils utilisent la dite donnée, avertis de cette modification, pour garantir qu'un prochain accès à leur propre copie contiendra bien la valeur attendue de la donnée. Ce problème possède des solutions efficaces mais dont le passage à l'échelle reste un objet d'étude avec l'augmentation du nombre de cœurs sur puce.

La difficulté du passage à l'échelle peut être abordée sous de multiples angles, allant de l'optimisation de la machine à états abstraite du protocole à l'organisation de la

hiérarchie mémoire autour de réseau sur puce avec des opérations plus avancées et effectuées au sein même du réseau d'interconnexion. Celui que nous choisissons dans cette thèse est plus modeste, et porte sur un point très particulier qui est un frein pratique important au passage à l'échelle : la représentation de la liste des processeurs possédant une copie d'une ligne de mémoire dans leur cache. Ce point, qui pourrait paraître anecdotique est en fait un problème très concret qui amène à un besoin démesuré en mémoire, de l'ordre de grandeur de celle utilisée par le cache de dernier niveau si l'on utilise des approches naïves.

Plan de la thèse

Dans le chapitre 2, nous exposons les problématiques de cette thèse liées à la cohérence de caches et plus particulièrement la représentation de la liste des copies de ces protocoles dans les architectures *manycores*.

Dans le chapitre 3, nous présentons un état de l'art sur les différents protocoles de cohérence de caches et les différentes représentations de la liste des copies. Nous nous intéressons principalement aux différentes représentations de la liste des copies pour les protocoles de cohérence avec un répertoire.

Le chapitre 4 présente notre représentation de la liste des copies que nous nommons DCC. Dans ce chapitre nous présentons le principe de DCC ainsi que ses différentes variantes.

Dans le chapitre 5, nous nous intéressons aux méthodes de simulation pour l'évaluation de la représentation de la liste des copies. Pour cela, nous présentons un rapide état de l'art des méthodes de simulation puis nous exposons notre méthode basée sur le simulateur PyCCExplorer que nous avons développé dans cette thèse.

Le chapitre 6 se focalise sur l'évaluation de l'influence de la représentation de la liste des copies. Pour cela, nous présentons les expérimentations faites avec PyCCExplorer ainsi que la validation de notre méthodologie de simulation. Nous nous intéressons également à l'évaluation de notre protocole DCC à l'aide de PyCCExplorer, puis nous présentons les résultats de synthèse obtenus lors de l'implémentation matérielle du protocole DCC.

Le chapitre 7 présente les pistes d'améliorations et les travaux futurs à mener dans la continuité des travaux de cette thèse.

Enfin, le chapitre 8 conclut cette thèse en répondant aux questions posées précédemment dans le chapitre 2.

Chapitre 2: Problématique

Sommaire

2.1 Architectures *manycores* . . . **3**
2.1.1 Modèle mémoire . . . 3
2.1.2 Hiérarchie mémoire . . . 5
2.2 Caches et cohérence des caches dans les architectures *manycores* . . . **6**
2.2.1 Architecture des caches . . . 6
2.2.2 Cohérence des caches . . . 7
2.3 Représentation de la liste des copies . . . **10**
2.4 Caractéristiques des applications parallèles . . . **12**
2.5 Évaluation des représentations de la liste des copies . . . **13**
2.6 Conclusion . . . **14**

Les progrès dans les technologies d'intégration et la diminution de la taille des transistors permettent aujourd'hui d'intégrer plusieurs centaines de cœurs sur un même circuit tout en limitant la consommation énergétique. En contrepartie, des difficultés particulières liées aux communications sont inhérentes à ce type d'architecture. En effet, lorsque le problème à résoudre est divisé en plusieurs sous-problèmes, les cœurs qui effectuent chacun une partie du problème ont besoin de se synchroniser et de partager les mêmes données.

2.1 Architectures *manycores*

Dans le cadre de cette thèse, nous nous intéressons aux architectures composées de plusieurs dizaines de cœurs que l'on qualifie d'architectures *manycores*. Les architectures *manycores* peuvent être composées uniquement de processeurs généraux de type *Central Processing Unit* (CPU) ou de plusieurs types d'unités de calcul, par exemple de CPUs ainsi que des cœurs spécialisés tel que des *Graphics Processing Unit* (GPU). Dans le premier cas, on parle d'architecture homogène alors que dans le deuxième cas, on qualifie ce genre d'architecture d'hétérogène. Dans le cadre de cette thèse, nous nous sommes focalisés sur les architectures homogènes.

2.1.1 Modèle mémoire

Les architectures *manycores* peuvent être classifiées selon leur type de modèle mémoire. Dans les architectures dites «à passage de messages», les processeurs utilisent des

messages pour communiquer. Sur chacun des processeurs, une instance d'un système d'exploitation est exécutée et tous les processeurs disposent également de leur propre mémoire à laquelle eux seuls possèdent le droit d'accès en lecture/écriture.

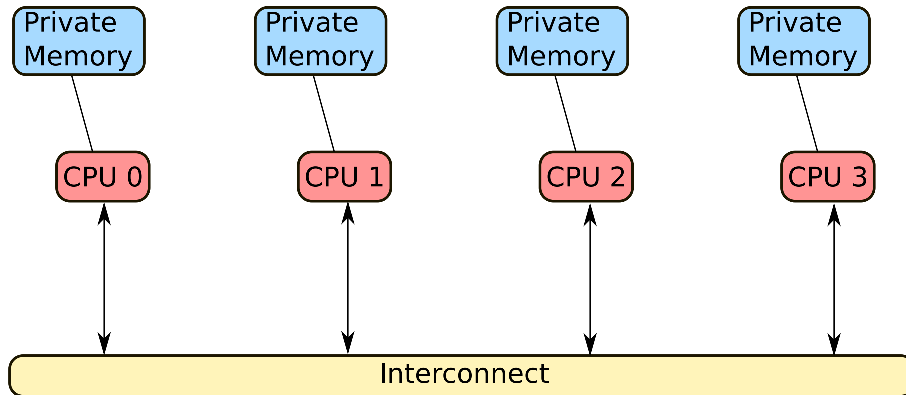


Figure 2.1 – Architecture à passage de messages

La figure 2.1 représente ce type d'architecture avec 4 CPUs et leur mémoire privée. Les CPUs communiquent entre eux à l'aide d'un bus de communication. Lorsque deux processus s'exécutant sur deux CPUs veulent communiquer, ils utilisent des messages de lecture ou d'écriture qui sont envoyés via le bus de communication. Lors de la réception d'un message, le CPU doit faire une suite particulière d'instructions permettant de l'écrire ou de le lire dans sa mémoire privée. Cette opération est faite à l'aide du système d'exploitation, ainsi ce type d'architecture nécessite un support de ce dernier et par conséquent s'appuie sur le logiciel.

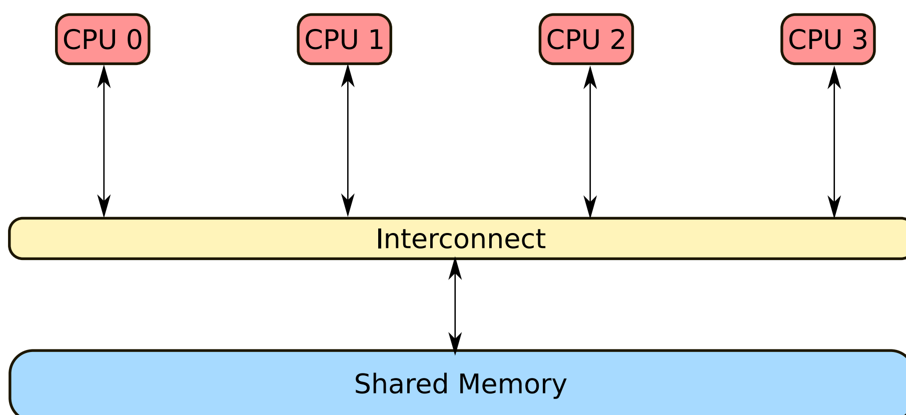


Figure 2.2 – Architecture à mémoire partagée

Dans les architectures à mémoire partagée, les processeurs ont accès à une mémoire en commun, comme le montre la figure 2.2. Cette fois, il existe une seule instance du système d'exploitation pour tous les processeurs et ils partagent la même mémoire. Pour s'échanger des données, les processeurs vont écrire et lire la mémoire partagée à l'aide de leurs instructions élémentaires d'accès à la mémoire. Ce type d'architecture s'appuie sur un support matériel pour l'accès à cette mémoire partagée.

Dans cette thèse, nous allons nous focaliser sur les machines *manycores* homogènes à mémoire partagée.

2.1.2 Hiérarchie mémoire

Les architectures *manycores* sont composées de plusieurs CPUs et de plusieurs mémoires afin de ranger les résultats des calculs. Les technologies de fabrication des mémoires ont des caractéristiques de taille, débit et latence très différentes, ce qui a historiquement déterminé la manière même de concevoir un ordinateur. Plus les mémoires sont proches des unités de calcul (c'est-à-dire des cœurs) plus elles sont petites et rapides. Au contraire, plus elles sont éloignées du processeur, plus elles sont lentes mais elles possèdent une capacité de stockage généralement plus grande.

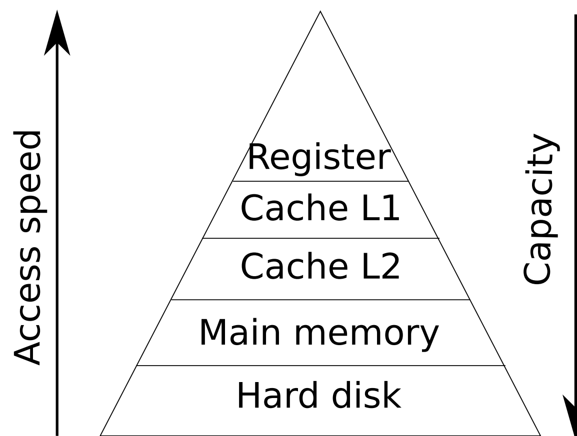


FIGURE 2.3 – Pyramide des différents types de mémoire

La figure 2.3 nous montre une pyramide qui représente une hiérarchie mémoire. À la base de la pyramide se trouvent les mémoires de type disque dur. Ces mémoires offrent une grande capacité (plusieurs tera-octets) mais elles sont particulièrement lentes. Actuellement, les *Solid-State Disk* (SSD) ont un débit de lecture qui peut atteindre 3 Go/s pour une latence de l'ordre de 0,1 ms. Les disques durs classiques ont une latence moyenne de 4,2 ms.

À l'étage supérieur se trouve la mémoire centrale ou principale. Cette mémoire permet de mémoriser une partie des données présentes dans le disque dur. Ces données sont copiées dans la mémoire principale lorsque le système d'exploitation souhaite y accéder. On utilise pour cela des mémoires de type *Dynamic Random Access Memory* (DRAM) qui ont une capacité de stockage de l'ordre de quelques giga-octets (aujourd'hui on trouve facilement des DRAM de 8 Go et bien plus) et elles ont un débit de lecture théorique jusqu'à 25 Go/s et une fréquence maximale de 3,2 GHz pour les DDR4 les plus rapides avec un contrôleur 64 bits.

Pour réduire la latence d'accès aux données, les processeurs utilisent des mémoires plus rapides que la mémoire principale que l'on appelle des caches [Smi82]. Les caches

possèdent une partie des données présentes dans la mémoire principale. Dans les architectures *manycores*, on retrouve plusieurs niveaux de cache. Plus le niveau est élevé, plus ce cache est loin des processeurs, mais dans ce cas il dispose d'une capacité plus grande qu'un cache proche d'un processeur. On accède aux données contenues dans les caches en quelques cycles, selon le niveau du cache. Pour un cœur, il existe le plus souvent deux caches de premier niveau *Level One* (L1), l'un pour les instructions et l'autre pour les données. Au-dessous, les caches de niveau 2 (L2) peuvent être :

- privés : seulement un cœur peut accéder à ce cache,
- partagés : plusieurs cœurs (mais pas forcément tous) peuvent accéder aux données de ce cache.

Enfin, au sommet de la pyramide, on retrouve les registres. Les unités de calcul utilisent les registres pour effectuer leurs calculs. Selon le type d'architecture les registres ont une taille de 32 ou 64 bits et ils sont accédés en un cycle du processeur.

Dans la pratique, lorsqu'un cœur souhaite lire une donnée, il envoie une requête de lecture à son cache. Si le cache possède la donnée, alors il y a *hit* dans le cache, sinon on parle de *miss*. Si le cache ne dispose pas de la donnée, il fait une requête au niveau supérieur de la hiérarchie mémoire jusqu'à ce que la requête arrive à un étage où se trouve la donnée. Lorsque la réponse est envoyée, elle traverse elle aussi les différents étages de la pyramide, et chaque étage peut mémoriser cette donnée. Par exemple, pour les caches, cela dépend de la politique d'inclusivité. Si un cache L2 est inclusif, alors toutes les données présentes dans les caches L1 le sont aussi dans le cache L2. Il faut donc dimensionner la taille du cache L2 afin qu'il puisse contenir les données mémorisées dans les caches L1. En effet, si le cache L2 est plein, alors il faut évincer une ligne de cache du L2, ce qui engendre l'éviction des données également dans les caches L1.

2.2 Caches et cohérence des caches dans les architectures *manycores*

Nous avons vu que les architectures étaient composées de plusieurs types de mémoires. Dans cette partie, nous allons nous focaliser sur les caches qui sont les plus petites mémoires en dehors des registres des processeurs. Au cours d'une exécution d'un programme, il existe un principe de localité temporelle : lorsqu'une donnée est accédée, il y a de fortes chances qu'elle soit de nouveau accédée dans un futur plus ou moins proche. Les caches permettent de ranger les données déjà accédées dans une mémoire proche des cœurs qui les utilisent, ce qui permet de réduire la latence d'accès.

2.2.1 Architecture des caches

Quel que soit le niveau d'un cache, ce dernier est organisé de la même manière. Il est structuré sous la forme d'agrégations de lignes de cache. Les lignes de cache ont une taille fixe et permettent de mémoriser des données présentes à des adresses mémoires consécutives. Une ligne de cache contient donc plusieurs mots mémoire. Le choix de mémoriser plusieurs mots en un seul bloc a été fait en s'appuyant sur le principe de localité spatiale. En effet, lorsque l'on exécute l'instruction **i**, il y a de fortes chances que

l'instruction suivante à exécuter soit l'instruction **i + 1**. Pour les données, si l'on est en train de lire une donnée comme un tableau à l'instant **t**, à l'instant **t + 1** il y a une forte probabilité que l'on accède à la suite du tableau qui est rangée en mémoire à l'adresse suivante. En augmentant la granularité lors des requêtes auprès de la mémoire principale, le nombre de messages échangés entre les caches et avec la mémoire principale diminue, en particulier lorsque l'on travaille sur des données proches.

Pour faire correspondre une adresse physique à un emplacement dans un cache, il existe plusieurs stratégies possibles :

les caches à correspondance directe : chaque adresse mémoire correspond à un seul emplacement dans le cache.

les caches totalement associatifs : les données peuvent être rangées n'importe où dans le cache.

les caches associatifs par ensemble : une adresse mémoire correspond à un ensemble d'emplacements et la donnée est rangée dans l'une des lignes réservées à cet ensemble.

Chacun de ces types possède des avantages et des inconvénients. Le calcul de l'entrée pour un cache à correspondance directe est facile à réaliser, il s'agit de l'adresse mémoire modulo la taille du cache. En revanche, si plusieurs données importantes pour le programme correspondent à la même ligne, elles vont se gêner en s'évinçant l'une l'autre à chaque accès, et ce même lorsque d'autres lignes du cache sont libres.

Les caches totalement associatifs n'ont pas ce problème, car une donnée peut être rangée dans n'importe quelle case. En revanche, il n'existe pas de fonction simple pour savoir à quel endroit est rangé une donnée. Il faut donc tester toutes les entrées pour trouver où se trouve la ligne de cache qui nous intéresse.

Pour remédier à ce problème, les caches associatifs par ensemble combinent les avantages de chacun de ces deux types. Le cache est divisé en plusieurs ensembles, eux-mêmes composés de plusieurs voies. Ainsi, une donnée peut être rangée dans l'une des voies d'un ensemble donné. Le calcul de l'entrée correspond donc à l'adresse modulo le nombre d'ensembles, puis il faut tester chacune des voies de l'ensemble afin de trouver la ligne de cache à laquelle on souhaite accéder.

2.2.2 Cohérence des caches

Dans le cadre de nos travaux, nous nous intéressons aux architectures *manycores* à mémoire partagée et cohérente. Lorsque l'on exécute un calcul sur plusieurs cœurs, ces derniers peuvent lire et écrire les mêmes données. Or lorsque qu'une machine utilise des caches, les cœurs font des requêtes seulement à leur cache et si ces derniers possèdent la donnée dans leur mémoire interne, ils répondent à la requête sans repasser par la mémoire principale.

La figure 2.4 montre un exemple avec quatre cœurs et leurs caches L1. Tous les caches L1 possèdent la donnée **D** qui vaut 10. Lorsque le cœur 1 change la valeur de **D**, son cache L1 est mis à jour. Sans communication entre les caches L1, la valeur de **D** est différente selon le cache. En effet, dans cet exemple, **D** vaut 0 dans le cache L1 1 et 10

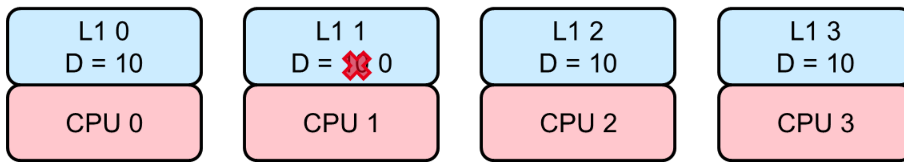


Figure 2.4 – Exemple avec 4 cœurs partageant des données

dans les autres caches L1. Il faut donc mettre en place un mécanisme de communication pour éviter qu'un tel problème se pose. Ce problème est connu sous le nom de cohérence de caches.

La gestion de la cohérence des caches peut être faite en logiciel ou en matériel. Dans le cas d'une gestion logicielle, le programmeur doit lui même invalider les lignes de cache avant qu'un cœur en modifie la valeur. Lorsque la cohérence des caches est gérée au niveau matériel, celle-ci nécessite du matériel supplémentaire mais l'écriture du logiciel n'est plus impactée. Dans le cadre de cette thèse, nous nous intéressons à la cohérence de caches effectuée de manière matérielle.

Il existe deux catégories de protocole de cohérence de caches matériel : ceux basés sur l'espionnage (*snoop*) et ceux basés sur un répertoire (*directory based*).

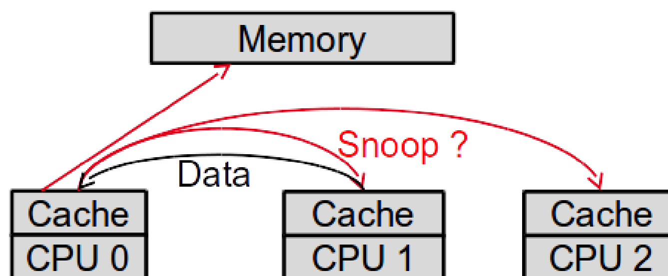


Figure 2.5 – Protocole de cohérence de caches basé sur l'espionnage

Dans le cas des protocoles de type espionnage, un cœur envoie un message de diffusion (*broadcast*) à tous les cœurs lorsqu'il veut accéder à une donnée, comme le montre la figure 2.5. Ce type de protocole passe difficilement à l'échelle car plus le nombre de cœurs, augmente plus le nombre de messages pour la cohérence sur le réseau augmente.

Le deuxième type de protocole de cohérence de caches utilise un répertoire comme sur la figure 2.6. Lorsqu'un cœur veut accéder à une donnée, il en fait la requête à un répertoire centralisé au niveau du cache de dernier niveau qui, lui, connaît la liste de tous les caches qui contiennent cette donnée. On appelle cette liste la liste des copies. La représentation la plus simple de la liste des copies est un champ de bits de la taille du nombre de cache L1, ce qui correspond aux nombres de cœurs lorsqu'il n'y a pas de séparation des caches d'instructions et de données. Quand un bit est à 1, cela signifie

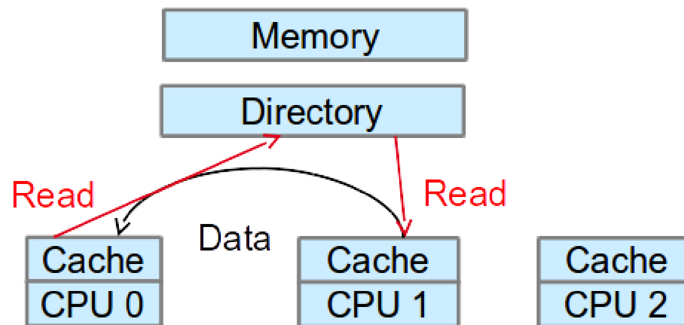


FIGURE 2.6 – Protocole de cohérence de caches basé sur un répertoire

que ce cache contient la donnée. Cette solution permet de générer seulement le nombre de messages nécessaires. En revanche, l'implémentation la plus triviale de ce type de protocole passe difficilement à l'échelle à cause du coût de stockage du répertoire qui peut atteindre la taille des données mémorisées.

Le répertoire doit aussi contenir des informations sur l'état de la ligne de cache et en particulier l'état dans lequel est la machine à états du protocole de haut niveau. L'un des protocoles de haut niveau le plus utilisé est le protocole *Modified*, *Exclusive*, *Shared*, *Invalid* (MESI), dont les états stables, par opposition aux états transitoires qui eux sont le reflet de l'implémentation, sont les suivants :

***Modified* :** la ligne a été modifiée et ce cache contient la donnée la plus à jour, par contre la mémoire principale n'est pas à jour ;

***Exclusive* :** la ligne de cache contient la donnée la plus à jour, tout comme la mémoire principale, en revanche aucun autre cache ne possède cette donnée ;

***Shared* :** la ligne de cache contient la donnée la plus à jour mais on ne sait pas si d'autres caches l'ont en mémoire. De plus, on ne sait pas si la mémoire principale est à jour ;

***Invalid* :** la ligne de cache n'est pas valide, c'est-à-dire qu'elle ne contient pas de données.

L'ensemble de ces états permet de savoir où se trouve la donnée la plus à jour. On peut ainsi savoir à l'éviction de la ligne de cache si les données doivent être mises à jour dans le niveau supérieur de la hiérarchie mémoire. Lorsque la ligne a été modifiée et qu'une demande d'écriture ou de lecture arrive d'un autre cache, il faut mettre à jour la mémoire principale en envoyant un message d'écriture différée (*write-back*).

La figure 2.7 montre les différentes transitions pour passer d'un état à un autre. Sur cette figure, on remarque qu'il existe quatre types de transitions :

***Local Read* :** accès en lecture du même cache ;

***Remote Read* :** accès en lecture d'un autre cache ;

***Local Write* :** accès en écriture du même cache ;

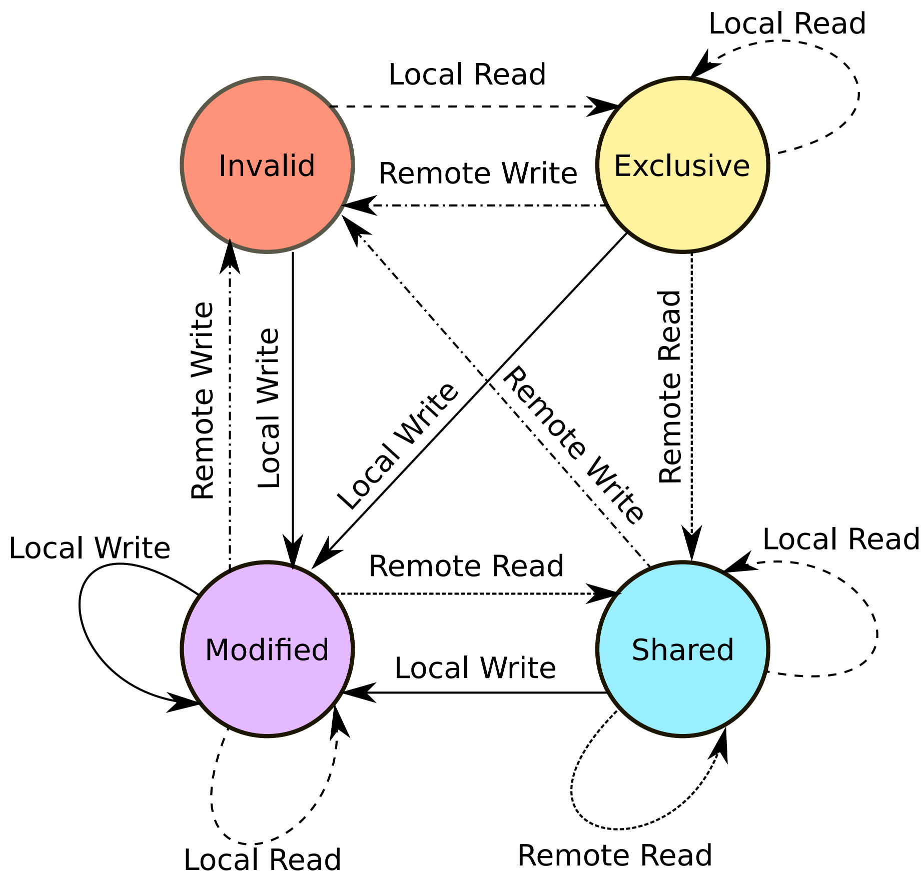


FIGURE 2.7 – Machine à états finis pour un protocole de type MESI

***Remote Write* :** accès en écriture d'un autre cache.

Le protocole *Modified*, *Owned*, *Exclusive*, *Shared*, *Invalid* (MOESI) est une amélioration du protocole MESI où l'état *Owned* est ajouté. Lorsque la ligne de cache est dans cet état, cette ligne contient la donnée la plus à jour mais d'autres caches peuvent aussi avoir cette donnée et la mémoire principale n'est pas à jour. Cet état permet d'ajouter un état entre l'état *shared* et *modified*, c'est-à-dire lorsque la mémoire principale doit être mise à jour. Ainsi, lorsqu'une donnée a été modifiée et qu'un autre cœur souhaite lire cette donnée, le cache qui a modifié cette donnée peut satisfaire la requête sans envoyer un message d'écriture différée à la mémoire principale. Ce nouvel état permet de limiter les messages à la mémoire principale.

2.3 Représentation de la liste des copies

Dans le cas des protocoles de cohérence de caches avec un répertoire, la taille de chaque entrée dans le répertoire dépend essentiellement de la représentation de la liste des copies. En effet, comme le montre la figure 2.8, chaque entrée du répertoire comporte plusieurs informations :

Tag : l'identifiant de l'adresse physique de la ligne de cache ;

State : l'état de la ligne de cache selon le protocole de haut niveau choisi (M, O, E, S ou I dans le cas du MOESI par exemple) ;

Sharers : la liste des caches qui possèdent cette ligne (c'est-à-dire la liste des copies).

Tag	State	Sharers

Figure 2.8 – Format d'entrée dans un répertoire avec la liste des copies

Parmi toutes ces informations, seule la représentation de la liste des copies évolue en fonction du nombre de cœurs dans le système. Ainsi, lorsque le nombre de cœurs augmente, le coût de la liste des copies augmente également, ce qui constitue un frein au passage à l'échelle des protocoles avec un répertoire. En effet, pour une ligne de cache de 64 octets (soit 16 mots de 32 bits), la taille du champ de bits nécessaire pour 1024 cœurs est deux fois plus importante que la taille nécessaire aux données elles-mêmes. Pour résoudre ce problème, on pourrait augmenter la taille des lignes de cache, mais cela augmente le nombre de données rapatriées dans le cache et potentiellement non utilisées, ainsi que le nombre de faux partages. On parle de faux partage quand plusieurs cœurs accèdent à des données différentes mais qui se trouvent dans la même ligne de cache. Par exemple, un cœur **c0** accède au deuxième mot de la ligne de cache et un autre cœur **c1** accède au sixième mot de la ligne de cache, cette ligne sera donc partagée entre **c0** et **c1** alors qu'ils n'utilisent pas les mêmes données. Plus on augmente la taille de la ligne de cache, plus on augmente le nombre de faux partages. Ce n'est donc pas une solution et c'est pourquoi plusieurs travaux de recherche s'intéressent à réduire la taille de la représentation de la liste des copies.

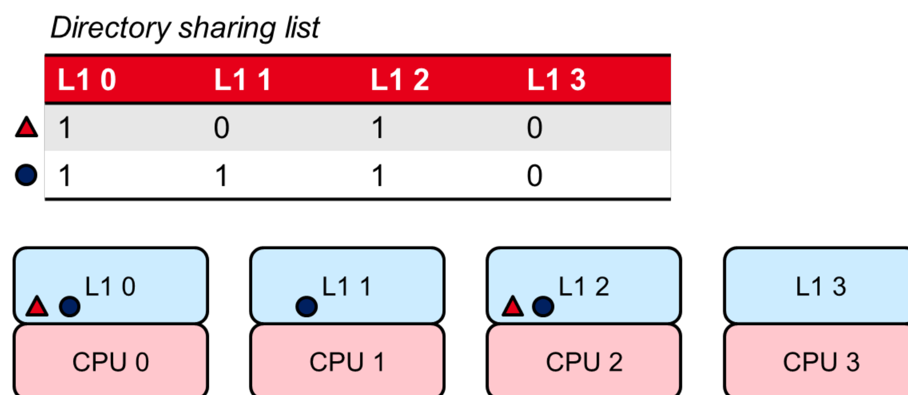


Figure 2.9 – Exemple de liste des copies représentée avec un champ de bits complet

La figure 2.9 montre un exemple d'une liste des copies représentée par un champ de bits. Dans cet exemple, la donnée représentée par un cercle bleu est présente dans les caches L1 des cœurs 0, 1 et 2. Les bits 0, 1 et 2 du champ de bits sont donc mis à 1 alors que le bit 3 vaut 0. La donnée représentée par un triangle rouge est présente dans les caches 0 et 2. On peut lire cette même information dans la liste des copies.

Dans la cadre de cette thèse, nous allons tenter de répondre à la question suivante : **Quelles sont les représentations de la liste des copies qui permettent de réduire la taille du répertoire des protocoles de cohérence de caches ?**

2.4 Caractéristiques des applications parallèles

Dans les architectures à mémoire partagée, les tâches des applications parallèles communiquent entre elles à l'aide de données partagées même si la plupart des données sont privées comme le montre la figure 2.10.

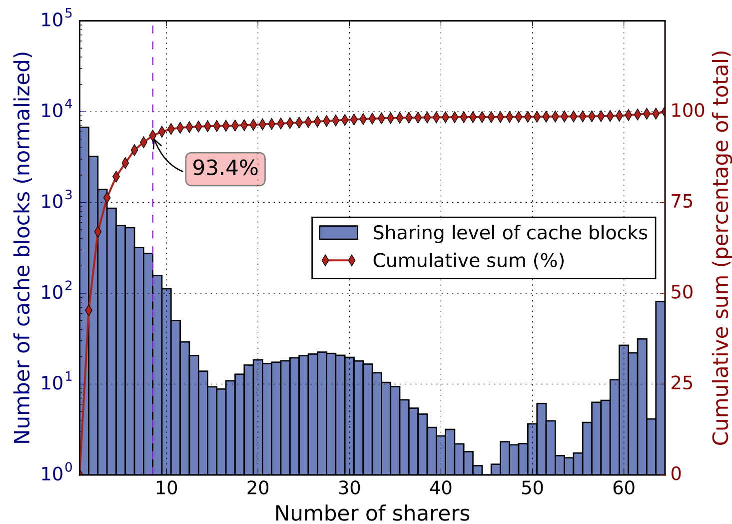


Figure 2.10 – Nombre de lignes de cache partagées par n cœurs

Sur cette figure, l'histogramme représente le nombre de lignes de cache partagées par n cœurs, pour n compris entre 1 et 64. Cette figure correspond à la moyenne du nombre de lignes de cache partagées pour 13 applications provenant de SPLASH 2 [WOT+95] et PARSEC [BKSL08], applications parallèles de référence pour l'évaluation des architectures *manycores*. On remarque que 93,4% des lignes de cache sont partagées par au maximum 8 cœurs pour une architecture de 64 cœurs. On note également que pratiquement la moitié des lignes de cache sont privées et ce pour la durée de vie de l'application. Cette figure a été obtenue en analysant le nombre de cœurs différents ayant accédé à chaque donnée tout au long de la durée de vie de l'application. Ce choix permet d'obtenir une borne supérieure du nombre de copies et ainsi ne pas être dépendant de la taille des caches. En effet, au cours de la durée de vie de l'application, le nombre de copies peut fortement évoluer.

En partant de ces observations, une question légitime se pose : **Peut-on exploiter les caractéristiques dynamiques des applications pour trouver une représentation plus compacte de la liste des copies ?**

2.5 Évaluation des représentations de la liste des copies

Nous venons de voir que les protocoles de cohérence de caches ont des difficultés à passer à l'échelle pour les architectures *manycores*. Au cours du développement des nouveaux protocoles de cohérence de caches, les architectes peuvent utiliser différentes stratégies pour leur permettre d'évaluer les performances de leur protocole, allant des approches analytiques à gros grain [JF96] à la simulation.

Dans cette thèse, nous nous intéressons aux techniques de simulation pour des architectures *manycores*. Afin d'évaluer les performances des protocoles, les architectes peuvent utiliser des simulateurs plus ou moins précis. Plus le simulateur est précis, plus la description du protocole doit l'être, et donc le temps de développement d'un protocole augmente. De même, plus la description du protocole est détaillée, plus on augmente le temps de simulation.

Il existe un nombre important de simulateurs et gem5 [BBB+11], qui permet de simuler des architectures multiprocesseurs au niveau système, est l'un des plus utilisés par les architectes. Dans le cas du simulateur gem5, le temps de simulation nécessaire au boot d'un Linux sur 64 cœurs Alpha est de 20 minutes.

Si l'on ajoute de la précision en utilisant par exemple SystemCASS [Sys], un simulateur basé sur SystemC, il faut 14 minutes pour démarrer linux avec un seul cœur, plusieurs heures pour une plateforme avec 4 cœurs et plus de 3 jours avec 16 cœurs. Le temps de simulation augmente plus rapidement que linéairement avec le nombre de cœurs, ce qui rend la simulation d'une architecture *manycore* trop lente pour être exécutée dans un temps raisonnable.

Une simulation du système complet décrit au niveau *Register Transaction Level* (RTL) n'est pas possible pour une architecture *manycore* car cela est encore plus lent qu'avec les simulations basées sur SystemC. Par contre, si l'on dispose d'un modèle RTL et d'un émulateur, l'émulation est une solution viable car rapide. Avec un émulateur, le RTL doit être synthétisé puis le design est chargé sur l'émulateur. Pour l'exécution, l'émulateur dispose de matériel dédié (principalement de la logique reconfigurable) qui permet de reproduire le comportement du RTL. En revanche, pour de l'exploration architecturale, il faut développer chacun des modèles au niveau RTL, ce qui n'est pas négligeable en temps de développement.

Les architectes doivent donc trouver le bon compromis entre précision et temps (développement et exécution). Pour cela, nous pouvons par exemple ne modéliser qu'une partie de l'architecture et nous concentrer sur une modélisation fine des caches, du protocole de cohérence, et du réseau. Une autre solution est de modéliser une partie du système avec un niveau d'abstraction plus élevé et ainsi modéliser seulement les états stables. Dans ce cas, on ne pourra pas prouver que la proposition est stable, sans *deadlock*. Par contre, cette solution permet de fixer la borne maximale des performances atteignables. Ainsi, l'évaluation du protocole ne correspond pas à l'évalution d'une implémentation matérielle mais bien des capacités du protocole en supposant que l'on peut atteindre les conditions optimales.

Dans cette thèse, nous allons essayer de répondre à la question suivante : **Comment peut-on évaluer un protocole de cohérence de caches et plus particulièrement**

la représentation de la liste des copies pour une architecture *manycore*?

2.6 Conclusion

Tout au long de ce chapitre nous avons soulevé plusieurs problèmes autour des protocoles de cohérence de caches et de leur simulation. Nous avons également défini un certains nombre d'hypothèses sur le type des architectures utilisées. Ces hypothèses sont les suivantes :

— nous considérons les architectures *manycores* à mémoire partagée ;

— nous nous intéressons seulement aux protocoles de cohérence de caches matériels.

Dans notre étude nous avons considéré que :

— les caches ont une politique d'écriture différée ;

— le protocole de cache de haut niveau est de type MOESI.

Ce sont des propriétés nécessaires pour la simulation, mais nos travaux peuvent être appliqués à des caches dont l'écriture se fait après chaque modification (*write-through*) ou à un autre type de protocole de cache de haut niveau.

En partant de ces hypothèses et de nos considérations, les questions auxquelles nous allons tenter de répondre sont les suivantes :

— **Quelles sont les représentations de la liste des copies qui permettent de réduire la taille du répertoire des protocoles de cohérence de caches?**

— **Comment peut-on évaluer un protocole de cohérence de caches et plus particulièrement la représentation de la liste des copies pour une architecture *manycore*?**

— **Peut-on exploiter les caractéristiques dynamiques des applications pour trouver une représentation plus compacte de la liste des copies?**

Les prochains chapitres de ce document ont pour but de répondre aux différentes questions énoncées ci-dessus.

Chapitre 3: Cohérence des caches dans une architecture *manycore* à mémoire partagée

Sommaire

3.1 Protocoles basés sur l'espionnage **15**
3.1.1 *Flexible snooping* 16
3.1.2 *In-Network Coherence Filtering* 17
3.2 Protocoles avec répertoire limité **18**
3.2.1 Ackwise 18
3.2.2 *Coarse bit-vector directory* 19
3.2.3 Virtualisation des CIDs 19
3.3 Protocoles de cohérence de caches dynamiques **20**
3.3.1 Liste chaînée 20
3.3.2 Champ de bits limité 21
3.3.3 *Sponge directory* 22
3.3.4 SPACE 23
3.4 Autres propositions **24**
3.4.1 *Scalable Interface Coherence* 24
3.4.2 *In-Network Cache Coherence* 24
3.4.3 Classifications des données 25
3.4.4 Fenêtres temporelles 26
3.5 Conclusion **27**

Nous avons vu dans la problématique (chapitre 2) qu'il existe deux classes de protocoles de cohérence de caches : ceux basés sur l'espionnage et ceux utilisant un répertoire. Seuls ces derniers possèdent une liste des copies. Néanmoins, ces deux classes tendent à se rejoindre avec d'un côté l'ajout de filtres pour limiter la délivrance de messages à tous les cœurs dans le cas des protocoles basés sur l'espionnage et de l'autre côté une représentation moins précise de la liste des copies pour les protocoles utilisant un répertoire.

3.1 Protocoles basés sur l'espionnage

Parmi les protocoles basés sur l'espionnage, on observe que des filtres sont de plus en plus utilisés afin de réduire le nombre de messages. Ces filtres permettent également de diminuer la latence car les requêtes ne doivent plus attendre les réponses de l'ensemble des cœurs.

3.1.1 *Flexible snooping*

Dans *flexible snooping* [SST06] les auteurs proposent d'ajouter des prédicteurs afin de se rapprocher de la solution optimale du point de vue de la latence. Dans leur exemple, le réseau a une topologie en anneau. La figure 3.1 montre trois types de communication. Dans le cas *lazy*, à chaque nœud le cache est interrogé, et lorsqu'il ne possède pas la donnée, la requête est transférée au cache suivant. Lorsque la donnée est présente, la réponse est envoyée au demandeur sans interroger les autres caches. La latence dépend donc de la distance sur l'anneau entre le demandeur et le premier cache possédant la donnée. Néanmoins, la topologie du réseau étant un anneau, la réponse doit traverser l'ensemble des nœuds suivants avant de revenir au demandeur mais ces nœuds ne sont pas interrogés, ils transfèrent simplement la requête au nœud suivant. Dans l'exemple *eager*, lorsqu'un nœud reçoit une requête, il la transfère au nœud suivant puis il interroge son cache. Tous les caches qui disposent de la donnée répondent à la requête. Cette solution permet de réduire la latence, car la requête est transférée directement. Néanmoins, tous les caches sont interrogés pour chaque requête et le nombre de messages générés est plus élevé. Enfin, le troisième type de communication *oracle* montre la solution idéale que les auteurs souhaitent atteindre. Dans ce cas, la requête est transférée directement au nœud dont le cache possède la donnée et seulement ce cache est interrogé. Cette solution permet de minimiser la latence d'accès aux données. Dans *flexible snooping*, le choix du type de traitement à effectuer à chaque nœud est fait à l'aide d'un prédicteur. Ce dernier permet de choisir si la requête est traitée avant d'être transférée, si elle est transférée puis traitée ou si elle est simplement transférée.

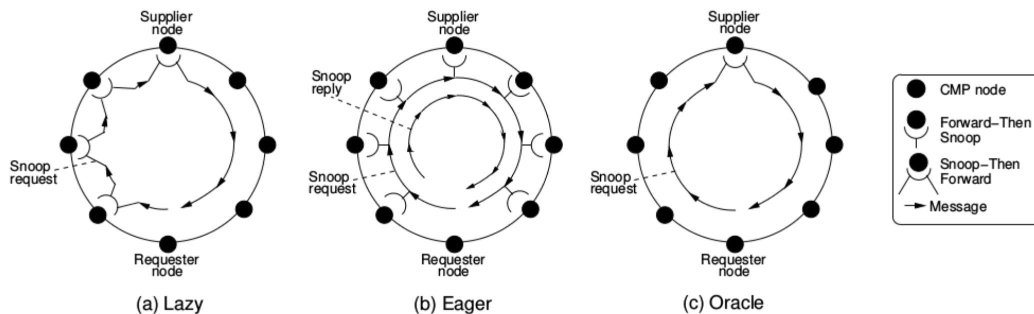


Figure 3.1 – Trois types de communication dans un réseau en anneau

Cette proposition est basée sur une topologie en anneau, et lorsqu'on utilise cette topologie, un nœud communique seulement avec ses deux voisins. Lorsqu'un nœud reçoit une requête, il la transfère seulement à son autre voisin. Ainsi, la requête passe par tous les nœuds avant de revenir à l'émetteur. Lorsqu'on utilise un réseau de type grille 2D, la requête ne passe pas nécessairement par tous les nœuds. Lorsque l'on souhaite envoyer un message à tous les autres nœuds (messages de diffusion), la requête peut être dupliquée afin de parcourir l'ensemble des nœuds plus rapidement. Les messages de diffusion doivent être limités lorsqu'on utilise un réseau de type grille car ils saturent rapidement le réseau. Ainsi, la proposition *flexible snooping* ne peut pas être appliquée à ce type de topologie qui est largement répandue dans les architectures *manycores*.

3.1.2 *In-Network Coherence Filtering*

Les auteurs de *In-Network Coherence Filtering* [APJ09a] proposent d'ajouter des filtres au niveau des routeurs. Pour cela, les filtres maintiennent une table pour savoir si la donnée est partagée par au minimum un nœud dans chacune des directions. Ainsi lorsqu'un message est reçu, le routeur sait vers quelle direction il peut diffuser le message. Afin de limiter la taille des filtres, les auteurs proposent de maintenir les informations de cohérence au niveau d'une région, c'est-à-dire de plusieurs lignes de cache consécutives.

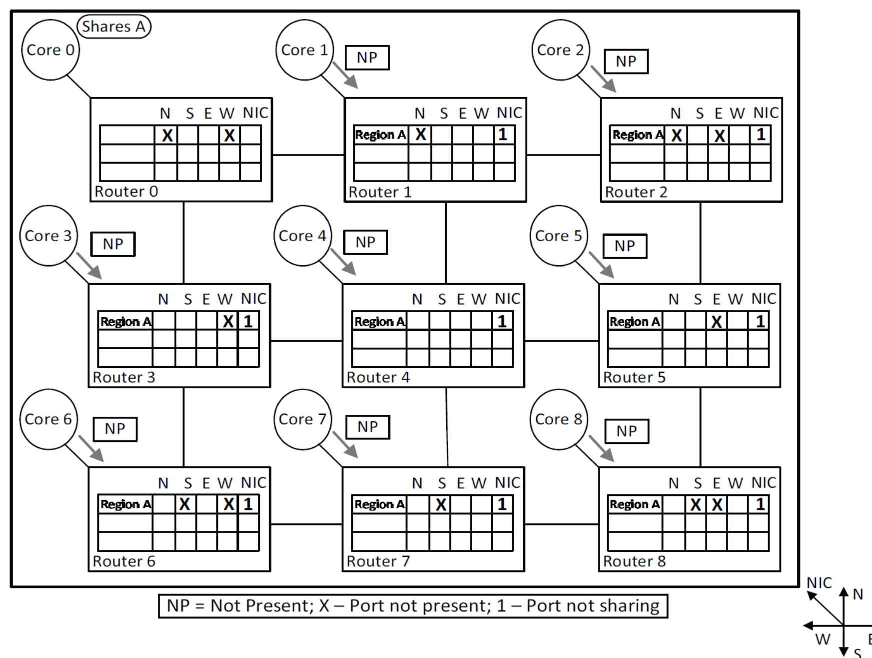


FIGURE 3.2 – INCF : exemple d'architecture avec des filtres au niveau des routeurs

La figure 3.2 montre un exemple avec 9 cœurs où le cœur 0 partage la région A. On remarque qu'il existe une entrée pour la région dans chaque filtre. De plus, on remarque que seuls les liens menant vers le cœur 0 avec un algorithme de routage XY sont activés. Cette solution permet de limiter les messages diffusés sur le réseau, mais les expérimentations ont été faites avec seulement 16 cœurs. Lorsqu'on augmente le nombre de nœuds dans le réseau, il faut augmenter la taille des tables au niveau de chaque routeur. De plus, lorsqu'on augmente la taille du réseau, les liens actifs pour chaque région augmentent et ainsi le nombre de messages diffusés augmente. Ces deux raisons font que ce protocole passe difficilement à l'échelle des *manycores*.

En l'état actuel des choses, et malgré l'ajout de filtres, les protocoles de cohérence de caches basés sur l'espionnage ne permettent pas de passer à l'échelle des *manycores* [SHW11].

3.2 Protocoles avec répertoire limité

Pour les protocoles de cohérence de caches utilisant un répertoire, la taille de ce dernier est un frein au passage à l'échelle à cause de la représentation de la liste des copies. Une solution pour limiter cette taille est d'utiliser un répertoire imprécis, tout en garantissant *a minima* que tous les caches concernés par un message le recevront.

3.2.1 Ackwise

Le protocole Ackwise proposé par Kurian *et al.* [KMP+10] limite le nombre de copies (c'est-à-dire le nombre de caches L1 qui possèdent la donnée). Pour cela, le protocole possède deux modes : le premier dans lequel le répertoire contient la liste des copies exacte et le deuxième où les messages de cohérence sont envoyés en diffusion. Lorsque le nombre de copies est inférieur ou égal à k, toutes les copies peuvent être mémorisées dans le répertoire.

State	G	*Keeper*	$Sharer_1$	...	$Sharer_{k-1}$

Figure 3.3 – Répertoire du protocole Ackwise

Le répertoire utilisé par Ackwise est présenté dans la figure 3.3 et contient les informations suivantes :

State : ces premiers bits sont nécessaires pour mémoriser l'état de la machine à états du protocole MOESI ;

G (*Global*) : ce bit est égal à 1 lorsque le mode diffusion est activé et 0 sinon, ce qui correspond au mode précis ;

***Keeper* :** ce champ est utilisé pour mémoriser le *Core ID* (CID) dont le cache est responsable de la cohérence ;

Sharers : ces champs servent à mémoriser les identifiants des caches contenant les copies. Chaque champ peut contenir un seul CID.

Le *keeper* est le cache responsable de la cohérence, c'est lui qui répond aux requêtes de lecture. Au-dessus de k copies, il n'y a pas plus de place dans le répertoire, on utilise alors des messages de diffusion. Pour éviter de saturer le réseau avec ce type de messages, le champ $Sharer_k$ est utilisé pour mémoriser un compteur du nombre de copies. Ainsi, seuls les caches L1 qui disposent de la donnée répondent à la requête, et le répertoire attend seulement ce nombre de réponses. Le nombre de champs pour mémoriser les copies dans le répertoire est fixé lors de l'implémentation et correspond au seuil k.

Dans leur article, les auteurs évaluent les performances d'Ackwise pour plusieurs valeurs de k comprises entre 2 et 64. Plus le seuil est élevé, plus les performances sont proches du protocole avec un répertoire exact mais le coût matériel est élevé. En effet, le coût matériel nécessaire pour mémoriser la liste des copies correspond au nombre de bits pour encoder le CID multiplié par le seuil k. Ainsi, pour réduire la taille du répertoire, il faut trouver un compromis entre performance et quantité de ressources. Pour cela les auteurs proposent de fixer un seuil bas, de l'ordre de quelques copies

(inférieur à 8).

La taille de chaque entrée du répertoire dépend de deux facteurs : la valeur du seuil k et le nombre de bits nécessaires pour encoder chaque CID. Pour une architecture avec $k = 6$ et composée de 64 cœurs, 6 bits sont nécessaires pour chaque CID, soit un total de $6 \times k$. Lorsqu'on s'intéresse à une architecture composée 1024 cœurs, il faut 10 bits pour chaque CID. Plus généralement, si on considère une architecture de n cœurs, il faut $\lceil log_2(n) \rceil \times k$ bits pour mémoriser la liste des copies. Le nombre de cœurs a donc une influence sur la taille de la représentation de la liste des copies, ce qui est un frein au passage à l'échelle des architectures *manycores*.

3.2.2 *Coarse bit-vector directory*

Une première représentation dynamique avec un répertoire limité a été proposé par Gupta *et al.* [GWM90]. Cette proposition se nomme *coarse bit-vector directory*. Les auteurs proposent de mémoriser un nombre réduit de CIDs dans le répertoire. Quand la limite est atteinte, le répertoire est utilisé pour mémoriser un champ de bits où chaque bit représente un groupe de plusieurs cœurs. Le nombre de copies ainsi que la taille des groupes de cœurs sont déterminés lors de l'implémentation et ne varient pas au cours de l'exécution. Dans leurs expérimentations, seulement trois CIDs peuvent être mémorisés dans le répertoire. Les auteurs ont montré que ce mécanisme permet de réduire le nombre de messages d'invalidation par rapport à l'utilisation de messages de diffusion lorsque le seuil est atteint. De plus, ce nombre de messages d'invalidation est à mi-chemin entre l'utilisation de messages de diffusion et la représentation de la liste des copies avec un champ de bits complet.

Le problème majeur de cette solution est son manque de flexibilité. En effet, les groupes de cœurs utilisés lorsque la liste des copies est pleine sont fixés lors de l'implémentation matérielle et ne sont pas connus du système d'exploitation qui répartit les tâches sur les différents cœurs. Ainsi deux cœurs placés à proximité l'un de l'autre peuvent être placés dans un groupe différent alors qu'ils travaillent sur les mêmes données. Afin d'améliorer ce protocole, une solution serait d'apporter un support logiciel afin que ce dernier ait connaissance des groupes utilisés pour la cohérence avec comme inconvénient principal de contraindre fortement le placement des tâches.

3.2.3 Virtualisation des CIDs

Fu *et al.*[FNW15] proposent d'utiliser un champ de bits sur un nombre restreint de cœurs. Ils nomment leur proposition *coherence domain restriction*. Pour cela ils proposent de virtualiser les CIDs : chaque page mémoire possède sa propre conversion entre cœur virtuel et cœur physique. Ainsi, si nous considérons un champ de bits de taille k, pour chaque page mémoire il existe une table de correspondance entre les bits 0 à $k-1$ du champ de bits et les cœurs physiques. Comme pour les autres protocoles proposant un répertoire limité, lorsque la table de correspondance est pleine et qu'il faut ajouter un nouveau cœur, le protocole passe dans un mode dégradé. Dans ce cas, les messages sont envoyés en diffusion.

Cette proposition comporte plusieurs problèmes pour le passage à l'échelle. En effet, lorsque le nombre de cœurs dans l'architecture augmente, la taille des tables de correspondance augmente également. De plus, cette solution est gérée au niveau des pages mémoires, et lorsqu'on augmente le nombre de cœurs, le taux de faux partage augmente. Sur une machine disposant de ressources mémoires largement supérieures à celles nécessaires à une application, un support logiciel permettrait de répartir les données utilisées sur l'ensemble des pages de manière à limiter les faux partages.

3.3 Protocoles de cohérence de caches dynamiques

Une autre possibilité pour représenter la liste des copies à l'intérieur du répertoire est d'utiliser des structures dynamiques. Ces structures sont rangées dans un espace partagé entre les caches d'un niveau donné. Chacune de ces structures a une taille fixe. Par nature les protocoles utilisant un répertoire dynamique sont également des répertoires limités. Les limites de ces répertoires sont fixées par la taille de l'espace partagé. Lorsque cet espace est plein, il faut libérer de l'espace et donc on perd des informations. Néanmoins, les protocoles dynamiques ajoutent de la flexibilité par rapport aux répertoires purement limités.

3.3.1 Liste chaînée

Une première représentation de la liste des copies utilisant des structures dynamiques est proposé par Thapar *et al.* dans [TDF93]. Dans cet article, les auteurs utilisent des listes chaînées de CIDs (*Linked list* en anglais). Cette liste chaînée est mémorisée dans un tas partagé et, lorsque le tas est plein, les messages de cohérence sont envoyés en diffusion.

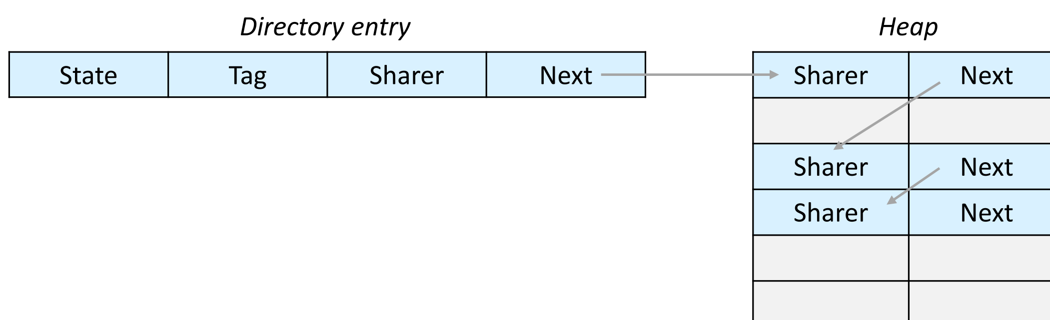


FIGURE 3.4 – Représentation de la liste des copies basée sur une liste chaînée

La figure 3.4 montre une entrée type du répertoire avec l'identifiant de la première copie et un pointeur vers un élément du tas. La structure de chaque élément du tas est composée d'un CID et d'un pointeur vers l'élément suivant. Cette représentation a donné lieu à plusieurs variantes dont *Released Write-Through* (RWT) proposé par Liu *et al.* [LDG+15]. Dans RWT les caches peuvent être *write-through* ou *write-back* selon si la ligne est respectivement dans l'état cohérent ou non-cohérent. Lorsque la ligne est dans l'état cohérent, la liste des copies est mémorisée à l'aide d'une liste chaînée limitée à un seuil k. Lorsque le nombre de copies dépasse le seuil, seul le nombre de copies

est mémorisé et les messages de cohérences sont envoyés en diffusion. Cette solution permet de limiter les ressources nécessaires pour mémoriser les listes chaînées.

Les protocoles basés sur une liste chaînée nécessitent une exploration architecturale afin de déterminer la valeur du seuil pour passer dans le mode dégradé ainsi que la taille du tas partagé car la performance de cette représentation de la liste des copies dépend de ces deux paramètres. De plus, le parcours de la liste ajoute de la latence au traitement des requêtes et cette latence augmente avec le nombre de copies.

3.3.2 Champ de bits limité

Les auteurs de *segment directory* [CP99] proposent d'utiliser des segments pour encoder un sous-ensemble de la liste des copies sous forme de champs de bits. Dans cette proposition, le répertoire est composé d'un champ de bits limité ainsi qu'un identifiant de début de segment, comme on peut le voir sur la figure 3.5. Ces champs sont respectivement nommés *segment vector* et *segment pointer* sur cette figure.

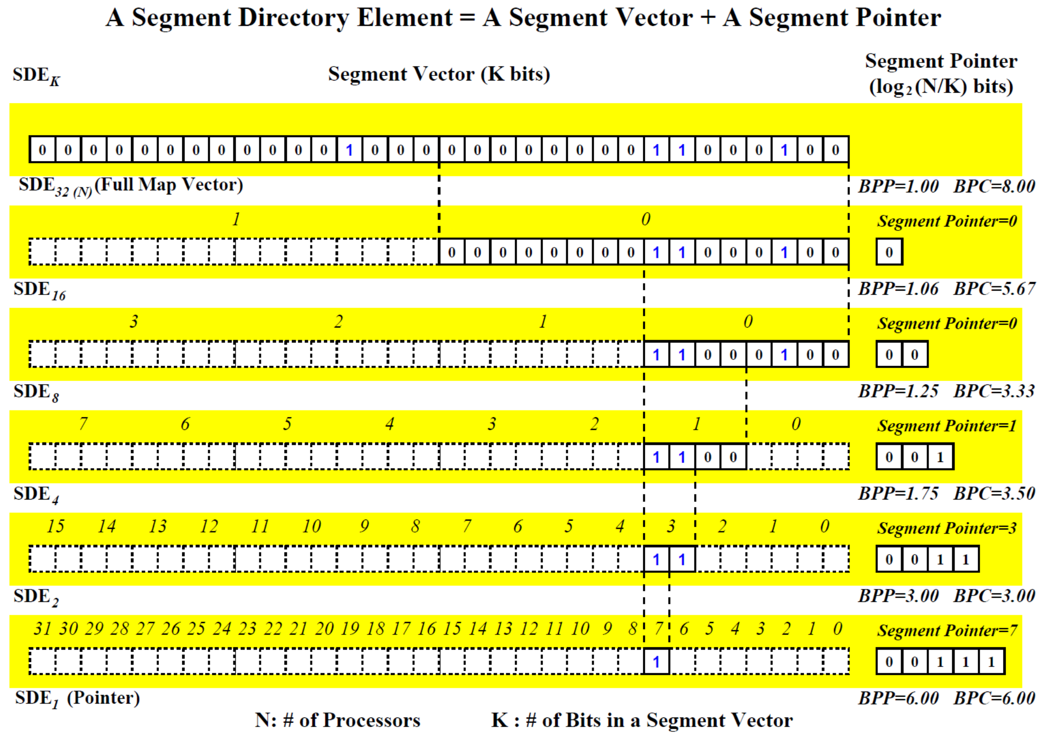


FIGURE 3.5 – Structure du répertoire utilisant des segments

Sur cette figure on observe le champ de bits (en trait plein) ainsi que son identifiant pour des segments contenant 64, 32, 16, 8, 4, 2 et 1 copies. Plus le segment est petit, plus la taille de l'identifiant augmente.

Cette proposition a été étendue par Shukla *et al.* [SC15] qui proposent *pool directory*. Ce dernier utilise deux représentations, l'une avec un répertoire limité contenant des

CIDs comme dans Ackwise, et l'autre avec des segments de taille fixe. Pour accéder à ces structures qui sont rangées sur un tas partagé, un pointeur vers la première structure est mémorisé dans le répertoire. Les lignes privées n'utilisent pas ce tas, leur CID est directement accessible dans le répertoire.

Le problème de ces deux propositions, *segment directory* et *pool directory*, est que les segments sont alignés sur un nombre fixe de bits. Ainsi, lorsque les données sont partagées par deux cœurs proches l'un de l'autre, rien ne garantit que la taille du segment sera minimale. Ces deux propositions pourraient être améliorées en permettant des segments non alignés sur des puissances de deux.

3.3.3 *Sponge directory*

Sponge directory [ZSS+14] est un répertoire proposé par Zhang *et al.* qui utilise également deux représentations de la liste des copies : soit avec des CIDs, soit avec un champ de bits complet. L'originalité de cette solution réside dans l'organisation du répertoire. Ce dernier est organisé comme un cache associatif par ensemble composé de plusieurs voies. Chaque voie est divisée en plusieurs niveaux qui contiennent un objet dont le type est dynamique. Lorsque la liste des copies est rangée sous la forme d'un champ de bits complet, tous les niveaux d'une voie sont utilisés pour une seule ligne de cache. En revanche, lorsque les copies sont mémorisées avec des CIDs, les niveaux de chaque voie sont répartis entre plusieurs lignes de cache. Ainsi, si nous considérons une voie w composée de 5 niveaux l, les copies de la ligne de cache A peuvent être rangées dans les niveaux $l1$ et $l4$, celles de la ligne B dans le niveau $l2$ et les niveaux $l3$ et $l5$ sont libres.

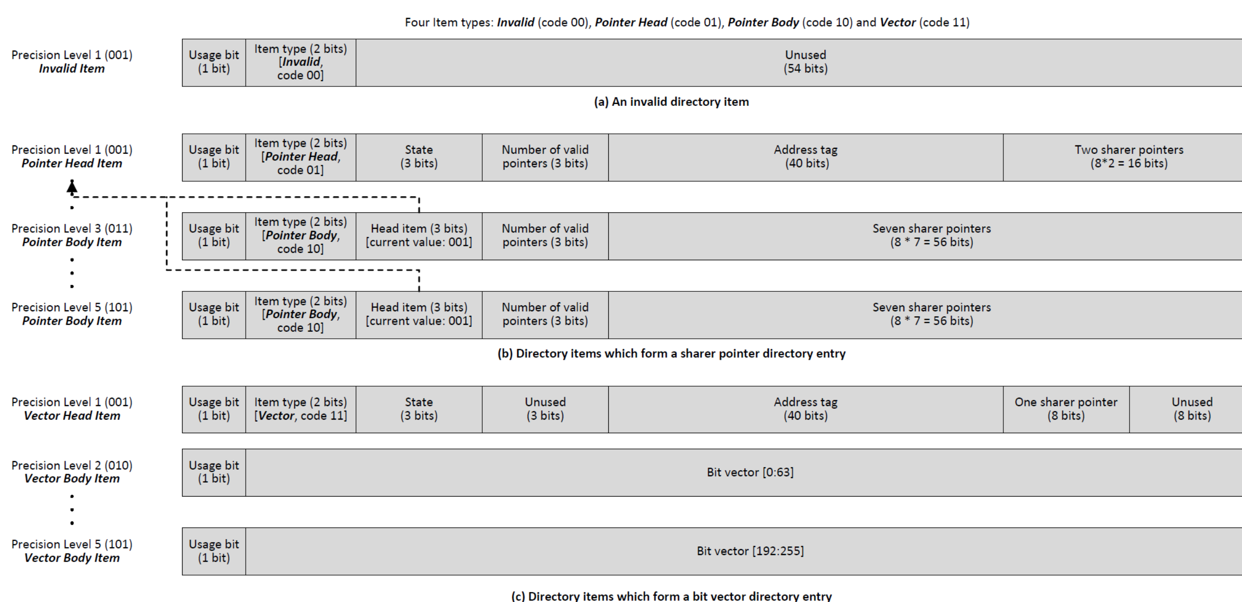


FIGURE 3.6 – Formats des entrées de *Sponge directory*

La figure 3.6 montre les formats possibles pour une entrée de *sponge directory*. Sur cette figure, on observe que le deuxième champ indique le type d'objet dont les

possibilités sont les suivantes :

Invalid : cette entrée du répertoire n'est pas utilisée ;

Pointer Head : il s'agit de la tête d'un objet qui contient des CIDs ;

Pointer Body : cet objet contient une partie des copies sous forme de CID et un pointeur vers la tête ;

Vector Head : il s'agit de la tête d'un objet qui contient un champ de bits complet ;

Vector Body : cet objet représente une partie du champ de bits complet.

Lorsque le mode de représentation est un champ de bits complet, l'ensemble des niveaux est utilisé pour mémoriser le champ de bits. La tête est donc toujours au premier niveau. La figure 3.6 nous montre également les trois types de format pour le répertoire. Le format (a) correspond à une entrée invalide, le format (b) correspond à une représentation avec des CIDs alors que le format (c) correspond à un champ de bit complet.

3.3.4 SPACE

Une autre proposition faite par Zhao *et al.* nommée SPACE [ZSD10] consiste à analyser des schémas de partage et à encoder les plus courants. D'après leurs analyses, pour une architecture de 16 cœurs, 200 schémas de partage permettent de représenter 80% des accès mémoire.

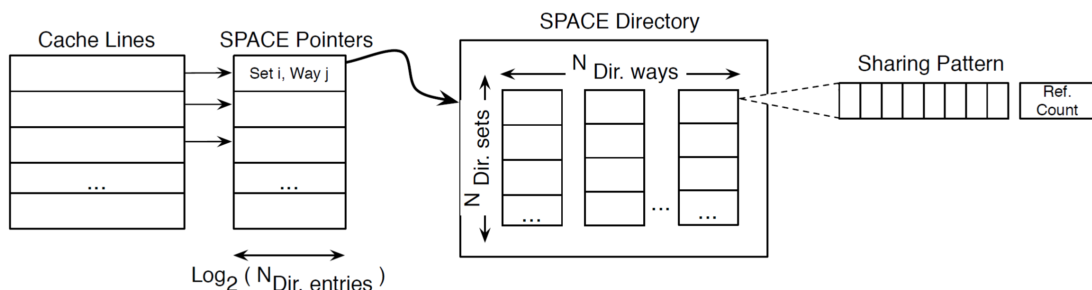


Figure 3.7 – Répertoire SPACE

La figure 3.7 montre les différents éléments qui composent le répertoire SPACE. Pour chaque ligne de cache, le répertoire contient un pointeur vers une entrée dans un tas partagé (nommé *SPACE Directory* sur la figure 3.7) qui contient la liste des copies sous la forme d'un champ de bits complet. Plusieurs entrées du répertoire pointent vers la même entrée du tas partagé. L'ajout d'une copie revient donc à changer le pointeur dans le répertoire pour qu'il pointe vers une entrée qui contient la nouvelle liste des copies. Lorsqu'aucune entrée ne correspond au schéma de partage, une nouvelle entrée est créée. Lorsque le tas est plein, la solution retenue par les auteurs est de fusionner deux schémas de la liste des copies qui sont proches l'une de l'autre, ce calcul étant réalisé à l'aide de la distance de Hamming. Dans ce cas, le schéma de partage contient des copies qui ne possèdent pas la donnée, ce qui va générer des messages d'invalidation qui ne sont pas nécessaires, mais cette solution reste cependant moins coûteuse qu'un message envoyé en diffusion.

Lorsque l'on augmente le nombre de cœurs dans l'architecture, le nombre de schémas de partage augmente également. De plus, la taille des champs de bits complet augmente également. Ainsi, le tas nécessaire à la mémorisation des schémas de partage augmente en largeur et en profondeur, ce qui est un frein pour le passage à l'échelle des *manycores*.

3.4 Autres propositions

Il existe d'autres représentations de la liste des copies bien connues dans la littérature. Nous avons sélectionné plusieurs protocoles de cohérence de caches basés sur des répertoires qui ne sont pas des répertoires limités ou dynamiques comme par exemple *Scalable Interface Coherence* et *In-Network Cache Coherence*. Nous nous sommes également intéressés aux protocoles utilisant une classification des données ou encore ceux utilisant des fenêtres temporelles.

3.4.1 *Scalable Interface Coherence*

Une première proposition faite en 1990 [JLGS90] relie les caches L1 entre eux en ajoutant à chaque entrée du répertoire un champ pour ranger un pointeur vers le prochain cache L1. Cette proposition qui se nomme *Scalable Interface Coherence* a été utilisée et évaluée dans différents travaux [LC96, TSS+97, FPRA16]. Le coût de la représentation de la liste des copies dépend essentiellement de la taille du pointeur, et ce dernier croît en logarithme du nombre de cœurs. De plus, dans cette proposition la liste des copies n'est pas ordonnée, cela engendre une latence plus grande car la recherche n'est pas optimale. Cette solution a un coût limité ce qui est une bonne chose pour le passage à l'échelle des *manycores* mais la recherche dans la liste est trop coûteuse en particulier lorsque la liste des copies est longue. De plus, la circulation des messages le long de la chaîne des caches L1 est source d'interblocages dans le réseau car cette liste se substitue alors à la fonction de routage du réseau.

3.4.2 *In-Network Cache Coherence*

Eisley *et al.* proposent de réduire la taille du répertoire en distribuant la liste des copies sur le réseau. On appelle ce protocole *In-Network Cache Coherence* (INCC) [EPS06]. Dans ce cas, des arbres virtuels sont créés et les informations concernant l'arbre se trouvent au niveau de chaque routeur.

Sur la figure 3.8, on voit que le nœud R1 souhaite lire une donnée. La requête est dirigée vers le nœud responsable de cette ligne (noté *home* sur la figure) selon l'algorithme de routage XY. Un répertoire est présent au niveau de chaque routeur afin de savoir si le nœud appartient à l'arbre ou non. Lorsqu'une requête arrive à un nœud, le routeur est interrogé afin de voir s'il existe une entrée dans le répertoire. Si aucune entrée ne correspond, alors la requête est transférée au nœud suivant toujours selon l'algorithme de routage XY. Lorsqu'une entrée du répertoire correspond à la requête, alors le nœud appartient à l'arbre et la requête peut être satisfaite. La réponse crée alors un arbre vers le demandeur. Sur cette même figure on observe également le nœud R2 qui souhaite lire la même donnée, l'arbre existant est alors étendu pour que R2 soit dans l'arbre de

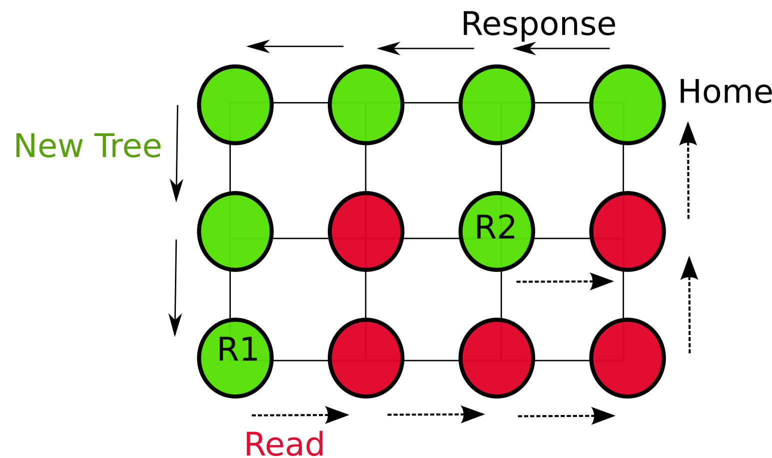


Figure 3.8 – Création de l'arbre de cohérence du protocole INCC

cohérence.

N	S	E	W	Root	Busy	Req	Valid

Figure 3.9 – Répertoire du protocole INCC

La figure 3.9 montre une entrée du répertoire de ce protocole. On remarque que les quatre premiers bits servent à encoder les directions où s'étend l'arbre. Le champ *Root* indique le lien qui permet d'obtenir la base de l'arbre de cohérence. Les bits suivants permettent de savoir l'état dans lequel se trouve la ligne de cache et l'entrée du répertoire. Cette proposition a l'avantage de réduire la taille des entrées du répertoire et de rendre cette taille indépendante du nombre de cœurs. En revanche, la profondeur du répertoire à chaque routeur croît en fonction du nombre de cœurs. De plus, l'utilisation des entrées du répertoire n'est pas la même à chaque nœud du réseau. Cette solution passe difficilement à l'échelle car l'ajout de cœurs augmente le nombre de requêtes à traiter à chaque nœud du réseau. Une implémentation matérielle de ce protocole a été réalisée dans [BNGD13]. Dans cet article, les auteurs ont réalisé des simulations avec des générateurs de trafic et non pas avec un trafic réel, ce qui ne permet pas d'évaluer le comportement de ce protocole pour une application réelle. Ses auteurs remarquent également que, comme dans la représentation précédente, la circulation des messages le long de l'arbre est source d'interblocages et nécessite de nombreux mécanismes de prévention et de recouvrement d'interblocages, ce qui complique l'utilisation de ce genre de protocoles.

3.4.3 Classifications des données

Afin de réduire le coût matériel de la liste des copies, plusieurs travaux proposent de classer les données en deux catégories : privées ou partagées par plusieurs cœurs. Lorsque les données sont privées, la cohérence n'est pas nécessaire. De plus, d'après Cuesta *et al.* [CRG+11], 75% des lignes de caches sont privées. Ils proposent de désactiver la cohérence pour ces lignes de cache, c'est-à-dire de ne pas ajouter les informations

de cohérence dans le répertoire. Pour cela, un support du système d'exploitation est nécessaire pour marquer et détecter les lignes de caches privées. Afin de réduire les ressources matérielles, ils proposent d'effectuer la classification à plus haut niveau, c'est-à-dire au niveau des pages mémoires. Par défaut, les pages sont privées et elles deviennent partagées lorsque le système d'exploitation détecte qu'un deuxième cœur souhaite accéder à cette même page. Lors de leurs expérimentations, 57% des lignes sont détectées comme privées, et 18% des lignes privées ne sont pas identifiées correctement. Malgré cet écart, la cohérence est désactivée pour la moitié des lignes de caches, ce qui permet de réduire le nombre d'entrées nécessaires pour le répertoire. Néanmoins, l'ensemble de ces résultats n'est valable que pour l'architecture qu'ils ont simulée, soit 8 tuiles contenant chacune 2 cœurs.

Les auteurs de *Spatiotemporal Coherence Tracking* [Ali12] proposent de réduire le nombre d'entrées dans le répertoire en utilisant une entrée pour une région (un ensemble de plusieurs lignes de cache consécutives) lorsque la donnée est temporairement privée. En effet, au cours de l'exécution, les lignes de cache contiguës peuvent être accédées par un seul cache. Dans ce cas, l'allocation d'une seule entrée dans le répertoire pour l'ensemble de ces lignes permet de gagner en espace de stockage. Le répertoire utilise deux formats d'entrées : un pour les lignes de cache et l'autre pour les régions. La taille de ces deux formats est la même, ce qui permet d'utiliser une structure partagée. Lorsque la région est considérée comme temporairement privée, une entrée pour la région est allouée dans le répertoire. Cette entrée contient le CID du cache ainsi que le nombre de lignes appartenant à cette page cachées par ce dernier. Lorsque la ligne est détectée comme temporairement partagée, il existe une entrée du répertoire correspondant à la région ainsi qu'une entrée par ligne de cache. Les expérimentations ont été réalisées avec seulement 16 cœurs, ce qui ne nous permet pas de confirmer ou d'infirmer que ce protocole de cohérence de caches passe à l'échelle des *manycores*.

3.4.4 Fenêtres temporelles

D'autres auteurs proposent de résoudre le problème du passage à l'échelle des protocoles de cohérence de caches en utilisant des fenêtres temporelles comme par exemple *Tardis* [YD15]. Lorsqu'un cache souhaite modifier une donnée, il obtient le droit de lecture ou d'écriture pendant une fenêtre de temps. Pour déterminer cette fenêtre de temps, chaque cache dispose d'un compteur correspondant au temps utilisé lors de sa dernière requête. Ainsi, lorsqu'un cache envoie une requête, il envoie également la valeur de ce compteur, correspondant à la durée de réservation qu'il souhaite. Si aucune réservation n'est déjà faite par un autre cache, le cache demandeur reçoit un message provenant du répertoire qui lui confirme sa réservation. Si la réservation expire et que le cache n'a pas terminé avec la donnée, il peut demander une prolongation. Lorsque le cache a terminé de lire ou d'écriture la donnée, il envoie la durée réelle de sa réservation et le répertoire rend de nouveau disponible la donnée pour une nouvelle réservation.

Cette proposition possède un avantage majeure pour le passage à l'échelle : le nombre de bits nécessaires pour chaque entrée du répertoire ne dépend pas du nombre de cœurs. En revanche, la difficulté majeure qu'induit la mise en place de ce protocole est le dimensionnement des fenêtres temporelles et donc de la taille des différents compteurs.

3.5 Conclusion

Nous nous sommes intéressés à plusieurs représentations de la liste des copies dont certaines ne sont pas adaptées aux architectures *manycores* comme celles utilisées par les protocoles basés sur l'espionnage à cause du nombre de messages de diffusion qu'ils génèrent.

Dans le cas des protocoles de cohérence de caches avec répertoire, la représentation de la liste des copies avec un champ de bits complet pose également un problème pour le passage à l'échelle à cause du coût de stockage. Pour cela, plusieurs travaux proposent de réduire la représentation de la liste des copies. Nous avons vu qu'il existe plusieurs représentations de la liste des copies limitées comme par exemple Ackwise, *coarse bit-vector directory* ou encore *coherence domain restriction*. Le problème de ces représentations de la liste des copies est que le coût de leur répertoire augmente avec le nombre de cœurs et que ces solutions manquent de flexibilité. Par exemple, la taille et la composition des groupes de cœurs utilisés dans le mode dégradé de *coarse bit-vector directory* est défini lors de l'implémentation. Afin d'améliorer certaines de ces propositions, un support logiciel peut être apporté afin de se ramener à une configuration proche de l'idéal pour chacun de ces protocoles. Néanmoins, un support logiciel demande un effort de développement supplémentaire pour chaque nouvelle architecture.

D'autres propositions vont plus loin en proposant des répertoires limités associés à un comportement dynamique. C'est le cas de la liste chaînée, *segment directory*, *pool directory*, *sponge directory* ou encore *SPACE*. Toutes ces représentations ont l'avantage de diminuer le coût matériel de la liste des copies mais plusieurs paramètres doivent être fixés lors de l'implémentation, ce qui nécessite une exploration architecturale afin d'obtenir des résultats performants. Ces solutions manquent de souplesse pour s'adapter aux différentes applications, en particulier *segment directory* et *pool directory* qui utilisent des champs de bits partiels mais obligatoirement alignés sur des puissances de deux.

Enfin, plusieurs travaux ont essayé de représenter différemment la liste des copies comme par exemple *Scalable Interface Coherence*, INCC, *Spatiotemporal Coherence Tracking* ainsi que les répertoires utilisant des fenêtres temporelles. *Scalable Interface Coherence* et INCC ne permettent pas de passer à l'échelle des *manycores* pour des raisons différentes. Pour *Scalable Interface Coherence*, la liste des copies n'est pas ordonnée ce qui ne permet pas une communication efficace. Pour INCC, la multiplication du nombre de cœurs engendre un trafic plus important au milieu du réseau que sur les bords, avec le risque de saturer le réseau. *Spatiotemporal Coherence Tracking*, qui classifie les données, et *Tardis*, qui utilisent des fenêtres temporelles, n'ont pas fait l'objet d'expérimentations avec des architectures *manycores*, ce qui ne nous permet pas de juger de leur efficacité pour ce type d'architecture.

Malgré les nombreuses propositions faites pour le passage à l'échelle de la représentation de la liste des copies, aucune solution ne semble se dégager des autres. De plus, certaines combinaisons de propriétés n'ont pas encore été testées comme par exemple, des groupes de cœurs dont la forme n'est pas fixée lors de l'implémentation matérielle.

CHAPITRE 4: REPRÉSENTATION DYNAMIQUE DE LA LISTE DES COPIES

Sommaire

4.1 Présentation du concept du protocole DCC . . . 29
4.2 Représentation du répertoire du protocole DCC . . . 31
4.2.1 Entrée du répertoire . . . 31
4.2.2 Liste chaînée . . . 32
4.2.3 Exemple d'entrée du répertoire DCC . . . 32
4.2.4 Coût matériel du répertoire . . . 33
4.3 Algorithmes de placement du rectangle cohérent . . . 33
4.3.1 Algorithme **idéal** . . . 34
4.3.2 Algorithme **premier touché** . . . 36
4.3.3 Algorithme **combinatoire** . . . 38
4.4 Détails de l'algorithme combinatoire . . . 39
4.4.1 Comportement de l'algorithme **combinatoire** . . . 40
4.4.2 Évaluation théorique du coût matériel . . . 43
4.5 Conclusion . . . 45

Nous avons vu que les protocoles de cohérence de caches passent difficilement à l'échelle des *manycores* et nous souhaitons en conséquence réduire le coût de la liste des copies. Dans ce chapitre, nous proposons une représentation dynamique de la liste des copies que nous nommons DCC pour *Dynamic Coherent Cluster*.

4.1 Présentation du concept du protocole DCC

Les protocoles de cohérence de caches avec répertoire peuvent utiliser un champ de bits complet pour mémoriser la liste des copies. Or, ce champ de bits est majoritairement creux. En effet, nous avons observé que le taux de partage de chaque ligne de cache est assez bas, 93% des lignes de cache étant partagées par au plus 8 cœurs. De plus, les auteurs de [FNW15] ont montré à l'aide d'analyses basées sur la loi d'Amdahl que le taux de partage n'augmente pas avec le nombre de cœurs. Nous avons également observé que les systèmes d'exploitation placent les tâches communicantes à proximité les unes des autres [DCA+16]. En s'appuyant sur ces deux observations, nous proposons une nouvelle représentation de la liste des copies. Pour ce faire, nous introduisons les structures suivantes :

— un champ de bits de taille C pour représenter les copies à l'intérieur d'un rectangle de cohérence contenant C cœurs,
— un identifiant de taille k pour l'origine (x, y) du rectangle de cohérence où k est une fonction de N, et N est le nombre de cœurs,
— un identifiant de taille I pour encoder la forme du rectangle, où I est une fonction de C,
— un tas partagé composé de H entrées pour mémoriser des listes chaînées de CID,
— un pointeur vers une entrée du tas pour chaque entrée du répertoire.

Le principe du protocole est basé sur une partition de la liste des copies en deux sous-ensembles. Le premier correspond à un rectangle (au sens du produit du nombre de cœurs en x par le nombre de cœurs en y), d'aire maximale C, englobant une partie des caches copiant la ligne, l'idée étant que la forme et l'origine de ce rectangle permettent de recouvrir le maximum d'éléments de la liste des copies. Les copies ne pouvant être placées dans ce rectangle sont mémorisées sous forme de liste chaînée. Lorsque le nombre d'éléments dans la liste chaînée dépasse un seuil T, la ligne de cache est passée en mode diffusion, de sorte à ce que les messages de cohérence soient envoyés par précaution à tous les cœurs. Ce mode est également utilisé lorsque le tas est saturé.

La figure 4.1 donne un exemple de cette représentation pour un réseau de type grille 2D de taille 8×8. Sur cette figure les copies sont représentées par des étoiles. Dans cet exemple, cinq copies sont dans le rectangle de cohérence et deux copies sont placées dans la liste chaînée. Les copies sont identifiées dans la liste chaînée à l'aide de leur CID. Les copies placées dans le rectangle de cohérence possèdent des coordonnées relatives à l'origine de ce nouveau repère dans la grille, c'est-à-dire qu'elles sont translatées par rapport à l'origine du rectangle. Ces coordonnées sont ensuite encodées dans un champ de bits. Ainsi le cœur ayant les coordonnées $(1, 1)$ correspond au bit 0 du champ et celui dont les coordonnées sont $(4, 3)$ correspond au onzième bit du champ.

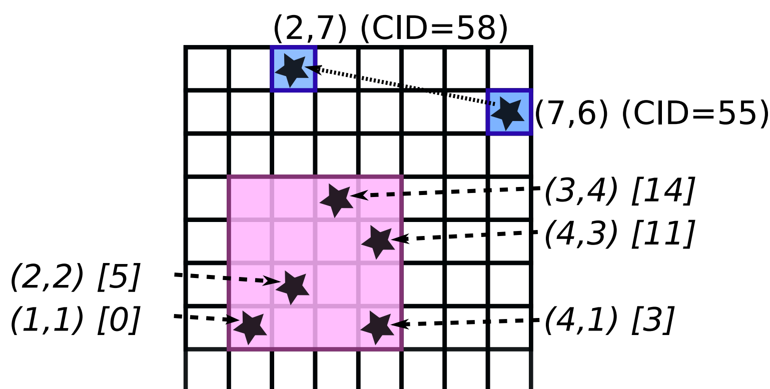


Figure 4.1 – Principe de DCC : les étoiles représentent les copies, les coordonnées (x, y) sont dans un plan 2D dont l'origine est le coin en bas à gauche de la grille, et les valeurs entre crochets représentent la position du cœur dans le rectangle

DCC permet de placer dans un champ de bits de taille réduite une partie significative des copies et la liste chaînée permet de mémoriser les copies isolées, comme celles liées à l'initialisation d'une donnée. La liste chaînée permet d'ajouter de la flexibilité

lorsque les copies ne sont pas proches. Cette situation se produit par exemple lors de la migration des tâches effectuée par le système d'exploitation. Afin de pouvoir s'adapter au déplacement des données qui pourrait avoir lieu lors de l'exécution, la position et la forme du rectangle évoluent dans le temps. Ainsi, à chaque moment, le rectangle contient le maximum de copies et la liste chaînée permet de mémoriser les copies qui ne sont pas comprises dans cette zone de cohérence, ce qui permet de cibler uniquement les cœurs concernés grâce à un *multicast*. En revanche, pour les données très partagées, notre protocole n'enregistre pas toutes les copies, mais seulement le nombre de copies. Les messages de cohérence sont alors envoyés en diffusion et le protocole est dans le mode imprécis que l'on nomme le mode diffusion.

4.2 Représentation du répertoire du protocole DCC

Nous avons vu que notre protocole possède deux modes : le premier où l'on dispose de la liste des copies exacte et le deuxième, le mode diffusion, où seul le nombre de copies est disponible. Quel que soit le mode, DCC utilise une entrée du répertoire dont le format d'entrée est montré sur la figure 4.2. Sur cette figure, seules les métadonnées spécifiques à notre protocole apparaissent. Pour être complet il faudrait ajouter les bits nécessaires pour le *tag* ou encore ceux indiquant l'état de la ligne de cache qui dépend du protocole de cache à haut niveau (MOESI, MESI, ...).

4.2.1 Entrée du répertoire

field :	broadcast	origin	shape	bit vector	pointer
size :	1	$\log_2 N$	l	C	$\log_2 H$

Figure 4.2 – Entrée du répertoire DCC

Les différents champs du répertoire sont les suivants :

Broadcast : ce bit permet de savoir si le mode diffusion est activé ou non ;

***Origin* :** ce champ représente l'origine du rectangle de cohérence ;

***Shape* :** ce champ encode la forme du rectangle ;

***Bit vector* :** ces bits sont utilisés pour mémoriser la liste des copies présentes dans le rectangle de cohérence sous la forme d'un champ de bits ;

***Pointer* :** ce champ est un pointeur vers le tas utilisé pour mémoriser la liste chaînée.

Sur la figure 4.2 apparaît également la taille de chaque champ. L'origine du rectangle nécessite $\log_2 N$ bits, où N est le nombre de cœurs. Pour encoder la forme du rectangle de manière compacte, le nombre de bits nécessaires dépend essentiellement de l'aire du rectangle de cohérence ainsi que du nombre de cœurs. Pour simplifier, nous ferons l'hypothèse que l'aire C est une puissance entière de 2, c'est-à-dire $C = 2^d$. Si l'aire maximale autorisée pour le rectangle est de 16, nous dénombrons sept formes de rectangles possibles : 1×16, 2×8, 3×5, 4×4, 5×3, 8×2, 16×1. Pour une architecture de 64 cœurs soit une grille de 8×8, les rectangles 1×16 et 16×1 n'existent pas car ils sortent de la grille. Néanmoins, pour une aire de 16, le nombre de bits nécessaires pour encoder

sept ou cinq formes reste le même. Le champ de bits a une taille correspondant à l'aire maximale du rectangle. Enfin, la taille du pointeur dépend du nombre d'entrées du tas, soit $\log_2 H$ bits, où H est le nombre d'entrées du tas.

Lorsque l'on est dans le mode diffusion, le champ *origin* est utilisé pour mémoriser le CID d'une copie que l'on nomme *keeper*. Le *keeper* devient responsable de la cohérence, il répond aux requêtes de lecture. Les champs utilisés pour le champ de bits ainsi que le pointeur vers le tas sont réutilisés pour mémoriser le nombre de copies.

4.2.2 Liste chaînée

Le tas utilisé par la liste chaînée contient H entrées elles-mêmes composées de deux éléments. Le premier élément est le CID de la copie et le second élément est un pointeur vers la prochaine entrée du tas. Le CID de la copie a une taille de $\log_2 N$ bits, et le pointeur a une taille de $\log_2 H$ bits.

Nous avons choisi de limiter de nombre d'éléments dans la liste chaînée par un seuil T. La valeur du seuil de la liste chaînée est importante pour plusieurs raisons :

- une liste avec une borne maximale sur sa taille permet de fixer une limite pour le temps de parcours de la liste,
- le seuil permet de ne pas polluer le tas avec les copies partagées par un nombre important de cœurs, en particulier les copies partagées par tous les cœurs,
- le nombre de copies dans la liste chaînée influence l'algorithme de placement du rectangle de cohérence. En effet lorsque le nombre de copies dans la liste chaînée est important, l'algorithme de placement va calculer plus de rectangles de cohérence avant de passer en mode diffusion. De plus, lorsque la taille de la liste augmente, le nombre de rectangles calculés augmente également ce qui améliore la chance d'obtenir le meilleur rectangle.

4.2.3 Exemple d'entrée du répertoire DCC

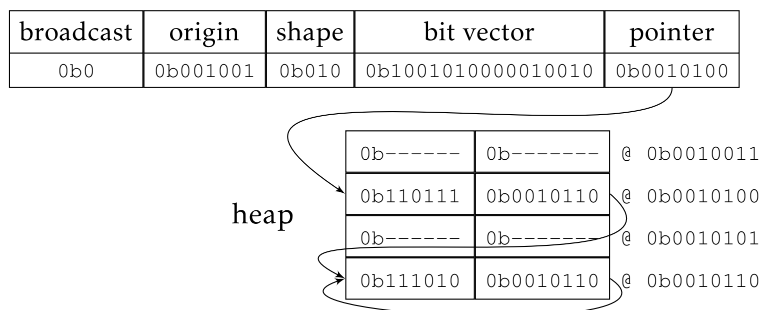


FIGURE 4.3 – Exemple d'entrée du répertoire DCC avec le tas

La figure 4.3 reprend l'exemple de la figure 4.1. Sur cette figure apparaissent les deux structures utilisées par DCC : l'entrée du répertoire et une partie du tas (les entrées 0b10011 à 0b10110). L'entrée du répertoire nous indique que cette ligne n'est pas en

mode diffusion, et le champ *origin* contient le CID du cœur 9, c'est-à-dire celui aux coordonnées $(1,1)$. Il s'agit de l'origine du rectangle cohérent de taille 4×4. Le champ de bits encode les copies présentes à l'intérieur du rectangle en calculant leur indice relatif à l'origine du rectangle. Ainsi le cœur aux coordonnées $(4,1)$ correspond au bit 3 avec une représentation des bits de poids fort en premier. L'entrée du répertoire contient également un pointeur vers une entrée du tas contenant le premier élément de la liste chaînée rangée à l'adresse 0b10100 du tas. Le premier élément de cette entrée du tas contient le CID de la première copie qui ne rentre pas dans le rectangle de cohérence. Dans notre exemple, il s'agit du cœur dont le CID est 55. Le deuxième élément de cette entrée du tas contient un pointeur vers la prochaine entrée utilisée par la liste chaînée. Cette deuxième entrée correspond à la copie dont le CID est 58 et le pointeur pointe sur cette même case pour indiquer la fin de la liste chaînée.

4.2.4 Coût matériel du répertoire

Dans notre exemple, l'architecture est composée de 64 cœurs, le rectangle de cohérence permet d'englober jusqu'à 16 cœurs et le tas est composé de 128 entrées. Dans ce cas, le coût d'une entrée du répertoire est de 33 bits. Pour avoir une estimation du coût de la cohérence par entrée du répertoire, il faut y ajouter le coût du tas. Le tas est composé de H entrées dont la taille du premier élément (utilisé pour mémoriser le CID de la copie) est de $\log_2 N$ bits et le pointeur vers l'élément suivant a une taille de $\log_2 H$ bits, soit dans notre cas un total de 1664 bits. Pour un cache L2 contenant par exemple 4096 lignes de 16 mots de 32 bits (soit un cache de 256 kB), le coût du tas est ramené à $\frac{1664}{4096} \approx 0.41$ bit par ligne, ce qui est très inférieur aux 512 bits de cette ligne de cache.

4.3 Algorithmes de placement du rectangle cohérent

Après avoir vu la représentation de la liste des copies dans le répertoire DCC, nous nous intéressons à l'algorithme de placement du rectangle cohérent. Nous avons vu que la position et la forme du rectangle peuvent évoluer au cours de l'exécution. Pour cela nous avons imaginé plusieurs algorithmes :

Idéal : cette version calcule à chaque accès le rectangle optimal en essayant tous les rectangles possibles.

Premier touché : dans cette version, après le premier accès, le rectangle englobe seulement le premier cœur (rectangle de taille 1×1). Par la suite, le rectangle peut seulement croître dans toutes les directions.

Combinatoire : cette version tend à se rapprocher de la version **idéale** en limitant les ressources. Cet algorithme cherche le rectangle englobant le maximum de copies et dont l'aire est inférieure ou égale à C à l'aide d'un bloc combinatoire avec un nombre limité d'entrées correspondantes aux copies. Lorsque le nombre de copies est supérieur au nombre d'entrées du bloc, nous avons mis au point une version **sous-optimale** qui tente d'approcher la solution idéale sans surcoût important par rapport au cas **optimal**.

Les trois versions sont décrites dans les sections suivantes.

4.3.1 Algorithme idéal

La version **idéale** correspond à la borne maximale en termes de performance. À chaque accès mémoire (ajout ou suppression d'une copie dans la liste), l'algorithme de recherche du meilleur rectangle est appliqué. Pour chaque origine, nous essayons tous les rectangles dont l'aire est inférieure ou égale à C et nous comptons le nombre de copies à l'intérieur de ce rectangle. L'un des rectangles qui permet d'englober le maximum de copies est sélectionné. Afin de limiter le nombre d'essais à effectuer, nous avons choisi de ne sélectionner comme candidats que les rectangles qui ne sont pas inclus dans un rectangle de taille supérieure. La figure 4.4 montre un exemple de rectangle englobant. Le rectangle d'origine $(0,0)$ de taille 2×7, en bleu sur la figure, est compris dans celui de même origine et de taille 2×8 représenté sur rouge sur cette même figure.

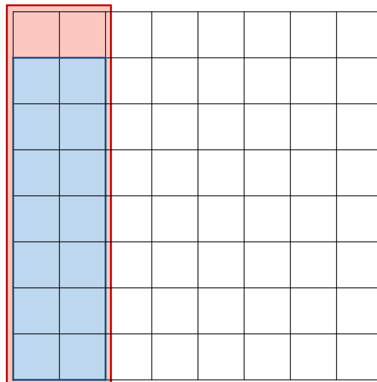


Figure 4.4 – Exemple de rectangle englobant un rectangle de taille inférieure

DCC idéal avec un tas illimité et sans seuil

Afin d'obtenir la borne maximale des performances de notre protocole, nous pouvons utiliser cet algorithme avec un tas illimité et sans seuil pour passer en mode diffusion. Dans ce cas, le protocole n'est pas limité par les ressources matérielles ni par une valeur du seuil mal choisie qui fait basculer dans le mode imprécis trop rapidement. Dans ce cas, les copies sont soit dans le rectangle, soit dans la liste chaînée. On a donc une représentation précise de la liste des copies et aucun message de type diffusion n'est généré. En revanche, une implémentation matérielle de cette solution est difficile pour deux raisons. Premièrement, la taille du tas équivalent à un tas infini correspond au produit du nombre de cœurs par le nombre de lignes de cache, soit 262 144 entrées pour 64 cœurs et 4096 lignes de cache. Ce nombre peut être réduit à 196 608 car la liste chaînée ne peut pas avoir une taille égale au nombre de cœurs : lorsque qu'une ligne est partagée par tous les cœurs, 16 cœurs sont déjà placés dans le rectangle de cohérence. La deuxième raison qui rend difficile l'implémentation matérielle de cet algorithme est liée à la complexité trop élevée de la fonction combinatoire implémentant cet algorithme en un temps raisonnable. En effet, dans le cas d'une architecture de 64 cœurs, il existe 7 origines possibles pour la forme 2×8 et 7 origines pour la forme symétrique 8×2. Pour la forme 3×5, il y a 24 origines possibles et autant pour la forme 5×3. Enfin, il existe 25 origines possibles pour la forme 4×4. Il faut donc tester 87 cas différents pour

déterminer le meilleur rectangle.

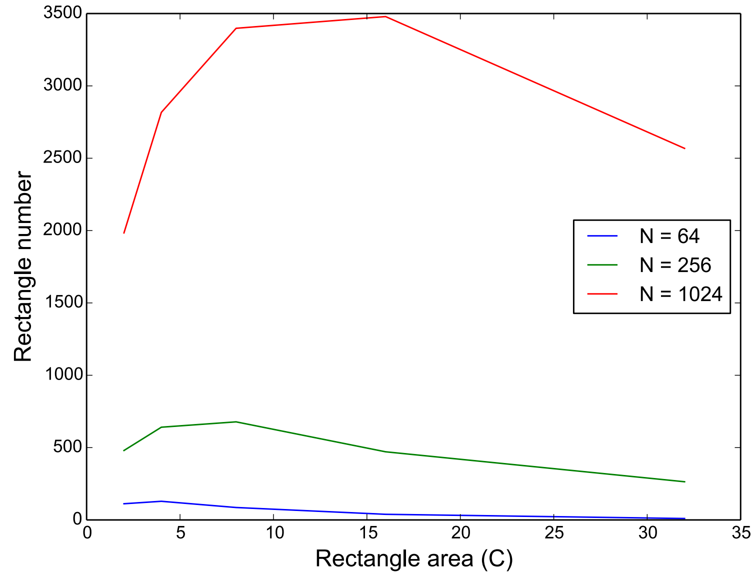


Figure 4.5 – Nombre de rectangles d'origine et de forme différentes en fonction de l'aire du rectangle pour 64, 256 et 1024 cœurs

En généralisant, pour une grille 2D carrée de taille $\sqrt{N} \times \sqrt{N}$, on peut compter le nombre d'origines différentes pour chacun des rectangles maximaux d'aire égale à C. En notant w la largeur et h la hauteur d'un tel rectangle, chaque rectangle peut être placé à $(\sqrt{N} - w + 1) \times (\sqrt{N} - h + 1)$ origines différentes dans la grille. La figure 4.5 montre le nombre de rectangles d'aire strictement égale à C en fonction de la valeur de C pour des architectures composées de 64, 256 et 1024 cœurs. Nous avons considéré seulement les rectangles dont l'aire est une puissance de 2 comprise entre 2 et 32. Le nombre de rectangles correspond à une borne inférieure car nous n'avons pas compté les rectangles dont l'aire n'est pas strictement égale à C. On remarque malgré tout que pour une architecture de 64 cœurs, le nombre de formes à essayer reste raisonnable, alors que pour des architectures plus grandes, ce nombre est trop important. Sur cette figure, on note également que les trois courbes ont la même allure, une partie croissante avec un maximum correspondant à $C = \frac{\sqrt{N}}{2}$. En conclusion, le nombre de cas à traiter devient trop grand lorsque le nombre de cœurs augmente pour permettre une implémentation matérielle dont le coût est raisonnable.

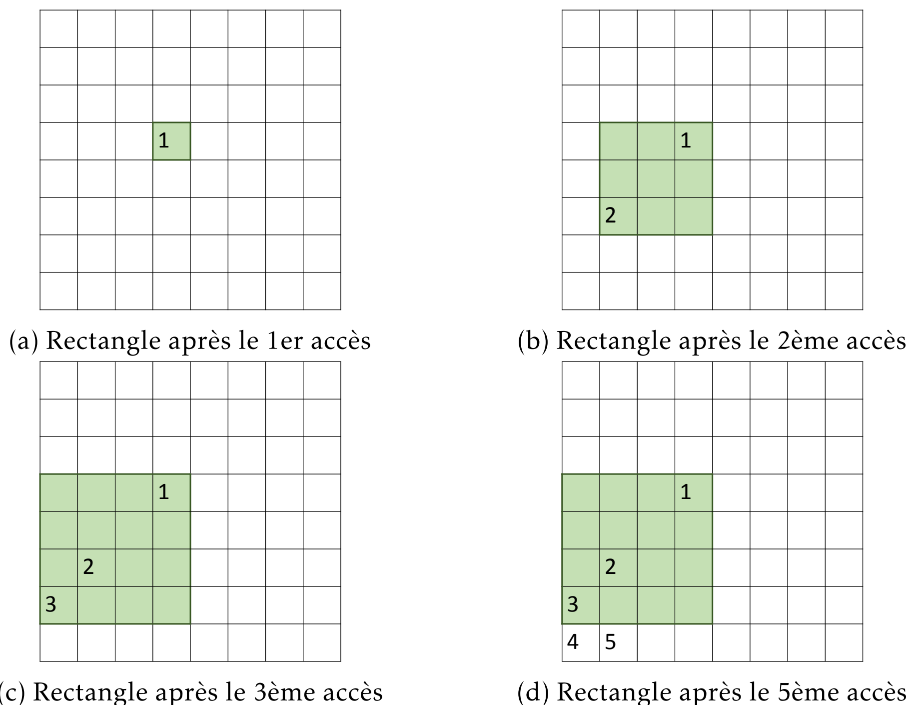
DCC idéal dégradé

Nous avons également défini une version dégradée de cette version **idéale**. Dans ce cas, un seuil T est fixé pour passer en mode diffusion et le tas a une taille finie. Par exemple, avec un seuil $T = 3$, lorsqu'un cœur ne peut pas être placé dans un rectangle de cohérence et que la liste chaînée contient déjà trois éléments, alors cette ligne doit passer en mode diffusion. Dans ce cas, toutes les entrées de la liste chaînée sont libérées

et l'entrée du répertoire DCC est mise à jour pour ne plus utiliser le rectangle de cohérence et son champ de bits associé. De plus, l'une des copies appartenant au rectangle est choisie pour devenir le *keeper* et le cache correspondant devient responsable de la cohérence pour cette ligne. Cette variante peut être implantée en matériel, mais elle reste trop coûteuse à cause du nombre de niveaux de logique nécessaires au calcul de la couverture d'une ligne pour tous les rectangles possibles. Nous avons donc développé d'autres algorithmes de placement du rectangle de cohérence.

4.3.2 Algorithme premier touché

Une première version d'algorithme de placement du rectangle cohérent dont les ressources matérielles sont raisonnables est l'algorithme **premier touché**. La copie ayant effectué le premier accès est placée dans le rectangle et les rectangles suivants devront toujours englober cette copie. Ensuite, à chaque nouvel accès, l'algorithme essaye de changer la taille du rectangle pour ajouter la nouvelle copie. Lorsque cela n'est pas possible, il essaye de placer la copie dans la liste chaînée. Si la liste chaînée a une taille égale au seuil T ou que le tas est plein, alors l'ajout dans la liste n'est pas possible et la ligne de cache passe en mode diffusion. Dans ce cas, toutes les copies comprises dans le rectangle et dans la liste chaînée sont enlevées et le répertoire est mis à jour. Comme dans la version précédente, l'une des copies appartenant au rectangle de cohérence est désignée pour devenir responsable de la cohérence (le *keeper*).



(a) Rectangle après le 1er accès (b) Rectangle après le 2ème accès

(c) Rectangle après le 3ème accès (d) Rectangle après le 5ème accès

Figure 4.6 – Évolution du rectangle de cohérence après chaque accès avec l'algorithme **premier touché**

L'ensemble des figures 4.6 représente l'évolution du rectangle de cohérence avec

l'algorithme **premier touché**, avec $C = 16$. Dans la suite de ce chapitre, nous considérons que le rectangle de cohérence a une aire maximale de 16. On remarque qu'après le cinquième accès il existe un rectangle englobant les copies 2, 3, 4 et 5 mais ce dernier ne contient pas la copie 1, il ne peut donc pas être choisi par cet algorithme. L'algorithme **premier touché** est un algorithme conservateur : lorsqu'une copie a été admise dans le rectangle de cohérence, cette copie sera toujours comprise dans le rectangle. Ainsi, lors des insertions, la taille du rectangle de cohérence peut seulement croître.

Cette solution est peu coûteuse et facile à réaliser en matériel. En effet, le rectangle de cohérence défini initialement sera agrandi pour tenter d'englober la nouvelle copie. Si l'aire du nouveau rectangle est inférieure ou égale à la taille du champ de bits, alors le rectangle est mis à jour. Sinon la copie est ajoutée à la liste chainée.

Le problème majeur de cet algorithme est dû à son comportement conservateur et en particulier vis-à-vis du premier cœur accédant à la donnée. En effet, dans le cas des applications utilisant des *threads*, un cœur unique initialise généralement l'ensemble des données nécessaires à l'exécution de chacun des *threads*, l'exécution étant réalisée par un ou plusieurs autres cœurs. Avec la variante **premier touché**, le rectangle contiendra le cœur qui a réalisé l'initialisation mais pas nécessairement les cœurs réalisant le calcul. Dans cet exemple, le placement du rectangle **idéal** ne doit pas nécessairement englober le cœur ayant initialisé les données. Ce cœur peut donc être placé dans le rectangle ou dans la liste chaînée.

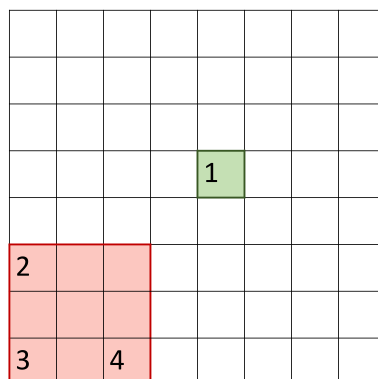


FIGURE 4.7 – Exemple de choix de rectangle **premier touché** en vert, **idéal** en rouge

La figure 4.7 illustre ce problème. Le cœur ayant fait le premier accès (noté 1 sur la figure) pour initialiser la donnée permet de placer le rectangle vert. Les cœurs effectuant les calculs sur cette donnée (noté 2, 3 et 4) ne peuvent pas être placés dans un rectangle englobant aussi le cœur 1. Lorsque l'algorithme **premier touché** est appliqué, ces copies se retrouvent dans la liste chaînée et si le seuil est fixé à 2 alors cette ligne passe en mode diffusion après la requête de lecture provenant du cœur 4. L'algorithme **idéal** permet quant à lui de changer de rectangle et de choisir le rectangle vert qui englobe 3 copies sur les 4.

La version **premier touché** est un algorithme de placement de rectangle simple à

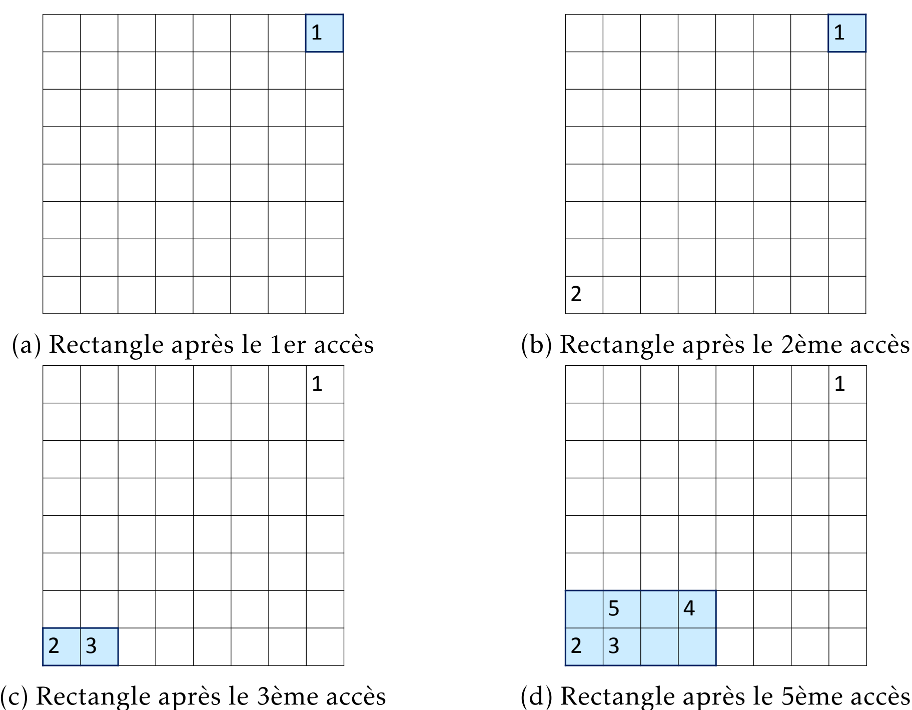
réaliser et peu coûteux. En revanche, cette version n'est pas satisfaisante par exemple lorsque l'initialisation des données n'est pas faite par le cœur qui effectue le calcul. De manière plus générale, cet algorithme manque de flexibilité pour le placement du rectangle. Cette flexibilité est nécessaire lorsque la migration des tâches est utilisée par le système d'exploitation.

4.3.3 Algorithme combinatoire

Afin de trouver un compromis entre ressources nécessaires et efficacité de la recherche du meilleur rectangle englobant, nous avons défini un algorithme **combinatoire**, qui peut s'exécuter en un seul cycle d'horloge. Cet algorithme est basé sur un bloc combinatoire que nous nommons *tiling*. Ce bloc prend en entrée les coordonnées de n copies, l'une de ces coordonnées pouvant être celle du rectangle, avec laquelle il faut donner un poids, c'est-à-dire le nombre de copies qu'il contient. Si le nombre total de copies dans le rectangle et dans la liste chaînée est inférieur ou égal à n, alors on est dans le cas **optimal** : il y a une entrée par copie. L'algorithme recherche le rectangle qui englobe le plus de copies et dont l'aire ne dépasse pas C. Dans ce cas, l'algorithme **combinatoire** a le même comportement et est équivalent à la version **idéale**.

Lorsque le nombre de copies est supérieur à n, alors on est dans le mode **sous-optimal**. Dans ce cas, la première entrée correspond aux coordonnées actuelles du rectangle avec son poids et les autres entrées représentent les éléments de la liste chaînée ainsi que le cœur ayant fait la requête de lecture. Pour que le mode **sous-optimal** soit utilisable, il faut que le nombre d'entrées n soit au minimum égal au seuil de la liste chaînée T plus deux : une entrée pour le rectangle et l'autre pour la nouvelle copie. Quel que soit le mode utilisé, il faut ensuite sélectionner le rectangle qui englobe le maximum de copies. Dans la phase de sélection, le poids est utilisé pour représenter le nombre de copies à l'intérieur du rectangle lorsque celui-ci est sélectionné pour faire partie du nouveau rectangle de cohérence. La liste des copies à placer dans le rectangle est donnée à la sortie du bloc. Les autres copies sont placées dans la liste chaînée ; si cette dernière est pleine ou que le tas est plein, alors la ligne passe en mode diffusion. Sinon, la liste chaînée est mise à jour ainsi que le répertoire. Le comportement de cet algorithme est détaillée dans la section 4.4.

La figure 4.8 montre l'évolution du rectangle de cohérence avec l'algorithme **combinatoire**. On remarque qu'au troisième accès le rectangle de cohérence change de place afin d'englober un nombre plus important de copies. Pour ce même exemple, on observe que les rectangles de cohérence choisis par les trois algorithmes ne sont pas les mêmes. La figure 4.9 illustre ce cas. Lorsque le nombre d'entrées du bloc combinatoire, 6 dans cet exemple, est inférieur au nombre de copies alors on observe des divergences entre l'algorithme **idéal** et **combinatoire**. Sur cet exemple, l'algorithme **premier touché** englobe une seule copie (en vert sur la figure). L'algorithme **combinatoire** englobe 5 copies (en bleu sur la figure). L'algorithme **idéal** permet d'en mémoriser 6, soit une copie supplémentaire.



(a) Rectangle après le 1er accès

(b) Rectangle après le 2ème accès

(c) Rectangle après le 3ème accès

(d) Rectangle après le 5ème accès

Figure 4.8 – Évolution du rectangle de cohérence après chaque accès avec l'algorithme **combinatoire**

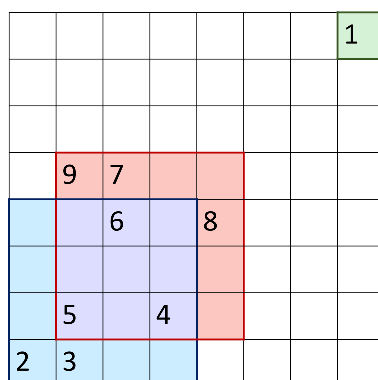


Figure 4.9 – Exemple de choix de rectangle **premier touché** en vert, **idéal** en rouge et **combinatoire** en bleu

4.4 Détails de l'algorithme combinatoire

L'algorithme **combinatoire** est le deuxième algorithme implémentable en matériel car il permet un bon compromis entre performances et ressources matérielles nécessaires. L'architecture du bloc *tilling* est représentée sur la figure 4.10 et son comportement est décrit dans l'algorithme 1.

Comme nous l'avons vu dans la présentation de l'algorithme **combinatoire**, dans le cas du mode **sous-optimal**, la première entrée contient les coordonnées du rectangle

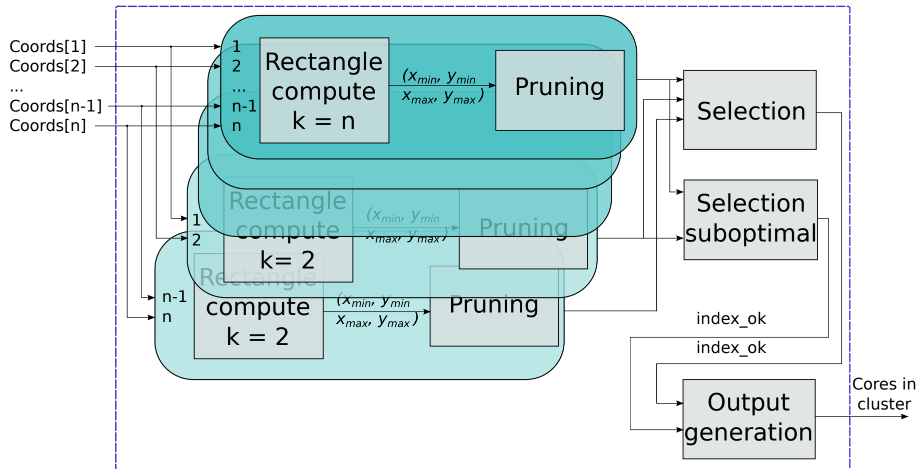


Figure 4.10 – Bloc *tiling* pour la recherche du rectangle de cohérence

sous la forme d'un quadruplet $(x_{min}, y_{min}, x_{max}, y_{max})$. Afin de simplifier le design de notre bloc, l'ensemble des entrées sont de la forme $(x_{min}, y_{min}, x_{max}, y_{max})$. Ainsi, la première entrée du bloc peut être un point avec $x_{min} = x_{max}$ et $y_{min} = y_{max}$ ou un rectangle. Le mode **sous-optimal** est utilisé lorsque :

$$n \geq T + 1(\text{nouvelle copie}) + 1(\text{rectangle})$$

où n est le nombre d'entrées du bloc *tiling* et T la taille maximale de la liste chainée, c'est-à-dire le seuil. Lorsque l'on est dans le mode **optimal**, il faut calculer les coordonnées des copies présentes dans le rectangle à l'aide du champ de bits, de l'origine et de la taille du rectangle. Ensuite, quel que soit le mode, il faut calculer les coordonnées des copies présentes dans la liste chaînée ainsi que celles de la nouvelle copie.

4.4.1 Comportement de l'algorithme combinatoire

L'algorithme **combinatoire** est réalisé à partir du bloc *tiling*, lui-même composé des blocs suivants :

Calcul des rectangles : ce bloc calcule les dimensions des rectangles englobants k copies parmi les n entrées. Il y a $\binom{n}{k}$ blocs de calcul des rectangles pour toutes les valeurs de k comprises entre 2 et n, soit $\sum_{k=2}^{n} \binom{n}{k}$;

Élagage : pour chaque bloc de calcul des rectangles, ce bloc teste si l'aire du rectangle englobant est inférieure ou égale à C, ce qui correspond à la taille du champ de bits de DCC ;

Sélection : ce bloc permet de sélectionner parmi tous les rectangles dont l'aire respecte la condition, celui englobant le plus de copies ;

Génération : ce bloc génère la liste des CID compris dans le rectangle de cohérence.

Après l'exécution du bloc *tiling*, les copies comprises dans le rectangle sont encodées sous la forme d'un champ de bits. De plus, l'origine et la taille du rectangle sont mises à jour dans le répertoire. Cette opération peut être réalisée au prochain cycle correspondant à un autre état de la machine à états.

Algorithme 1: Algorithme **combinatoire** pour la recherche du rectangle englobant

Données en entrée : s : nombre de copies ;
S : ensemble des coordonnées des copies ;
c : coordonnées du rectangle ;
w : poids du rectangle
Résultat : Ensemble des copies dans le rectangle

```
1  comb = collection de tous les sous-ensembles de S de taille k pour k ∈ [2,n] avec
     n = |S|;
2  for i = 1 to |comb| do
       /* Calcul des rectangles */
3      x_min ← min(e.x_min ∀e ∈ comb_i);
4      x_max ← max(e.x_max ∀e ∈ comb_i);
5      y_min ← min(e.y_min ∀e ∈ comb_i);
6      y_max ← max(e.y_max ∀e ∈ comb_i);
       /* Élagage */
7      Δx ← x_max − x_min;
8      Δy ← y_max − y_min;
9      if Δx × Δy ≤ C then          /* C correspond à la taille du champ de bits */
10         comb_ok ← comb_ok ∪ {comb_i};
   /* Sélection */
11 Trouver comb_best tel que |comb_best| ≥ |comb_i| ∀comb_i ∈ comb_ok;
12 if s + 2 > n then                                        /* Cas sous-optimal */
13     Trouver comb_best_c tel que
         |comb_best_c| ≥ |comb_i| ∀comb_i ∈ comb_ok such that c ∈ comb_i;
14     if |comb_best| ≤ |comb_best_c| + w − 1 then
15         comb_best ← comb_best_c;
16 if comb_best = ∅ then
17     if s + 2 > n then                                    /* Cas sous-optimal */
18         comb_best ← c;
19     else
20         comb_best est choisi parmi les éléments de S;
```

Bloc de calcul des rectangles

Le bloc de calcul des rectangles calcule les dimensions du rectangle englobant pour chaque ensemble k de coordonnées données en entrée, ce qui correspond aux lignes 3 à 6 de l'algorithme 1. Le résultat de ce bloc est donné sous la forme d'un quadruplet $(x_{min}, y_{min}, x_{max}, y_{max})$. L'implémentation matérielle peut être réalisée à l'aide d'un arbre de comparaisons. Ainsi la complexité de cette opération est de $k-1$ comparateurs pour chaque élément du quadruplet, soit $(4 \times (k-1))$ par bloc de calcul des rectangles. La figure 4.11 montre un exemple d'implémentation pour une combinaison de deux éléments. Le nombre d'entrées des blocs min et max dépendent du nombre d'éléments dans la combinaison.

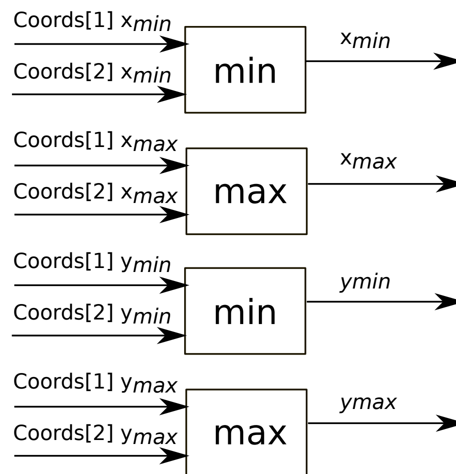


Figure 4.11 – Bloc de calcul des rectangles pour la combinaison [1, 2]

Bloc d'élagage

Le bloc d'élagage prend en entrée les sorties des blocs de calcul des rectangles sous la forme de quadruplets $(x_{min}, y_{min}, x_{max}, y_{max})$ comme le montre la figure 4.12. Pour chacune de ses entrées, il teste si l'aire du rectangle englobant satisfait la condition suivante : l'aire doit être inférieure ou égale à C, où C est la taille du champ de bits du répertoire. La première étape consiste à calculer la longueur et la largeur du rectangle à l'aide d'une soustraction comme l'indiquent les deux premiers blocs de la figure 4.12, cette étape correspond aux lignes 7-8 de l'algorithme 1. La deuxième étape peut être réalisée de deux façons. La première utilise une multiplication pour trouver l'aire du rectangle puis compare le résultat à la constante C, c'est cette solution que nous avons choisi de représenter sur la figure 4.12. Une autre solution est basée sur une table de valeurs précalculées (LUT). Dans notre cas, la LUT est symétrique et peut être optimisée en triant la longueur et la largeur du rectangle afin d'avoir toujours le plus petit dans la première entrée. Cette opération est réalisée avec une comparaison et deux multiplexeurs. Les deux solutions seront comparées dans la suite de cette thèse. Dans l'algorithme 1, nous avons choisi de faire apparaître la solution à base de multiplication car cette dernière est plus explicite.

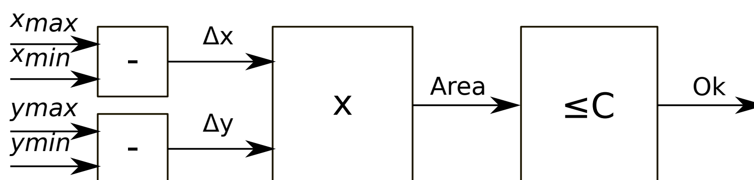


Figure 4.12 – Bloc d'élagage pour une combinaison de copies

Bloc de sélection

Le bloc de sélection correspond à l'arbre de décision final, il choisit le meilleur rectangle englobant satisfaisant la condition sur l'aire. Le meilleur rectangle est celui

qui contient le plus de copies. Dans l'algorithme 1, la sélection est faite à la ligne 13. Ce bloc est réalisé avec un encodeur de priorité de $\sum_{k=2}^{n} \binom{n}{k}$ entrées, comme le montre la figure 4.13. Lorsque le mode **sous-optimal** est activé, un deuxième bloc de sélection est exécuté en parallèle. Ce deuxième bloc de sélection, que l'on retrouve lignes 14-18 dans l'algorithme 1, prend en entrée un sous-ensemble de rectangles englobants. Ce sous-ensemble correspond à tous les rectangles qui respectent la condition sur l'aire et qui englobent le rectangle d'origine, c'est-à-dire celui en entrée du bloc *tiling*. Le poids, qui correspond au nombre de copies dans le rectangle, est ajouté afin de connaître le nombre exact de copies dans le rectangle englobant. Après l'exécution de ces deux blocs de sélection, le rectangle qui contient le plus grand nombre de copies est choisi. Si aucune combinaison ne satisfait la condition sur l'aire, c'est-à-dire que le meilleur rectangle englobe une seule entrée, alors la première entrée du bloc *tiling* est choisie pour devenir le rectangle. Dans le mode **optimal**, il s'agit d'une seule copie alors que dans le mode **sous-optimal**, cela correspond au rectangle d'origine. Cette opération correspond aux lignes 20-26 de l'algorithme 1.

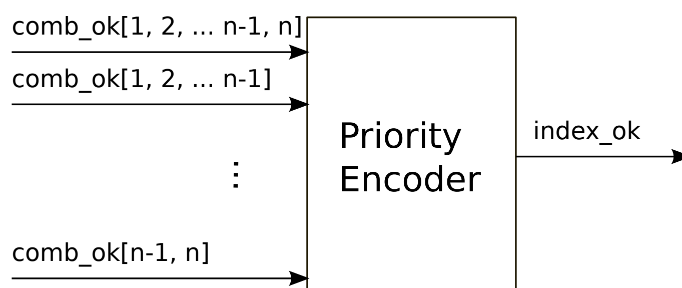


Figure 4.13 – Bloc de sélection implémenté avec un encodeur de priorité

Bloc de génération de la sortie

Le bloc de génération transforme l'index de la combinaison correspondant au rectangle choisi en une liste de copies. Dans notre cas, cette liste est un champ de bits où le bit 0 correspond à l'origine du rectangle. La sortie du bloc *tiling* est donc une liste de k copies pouvant être placées dans le rectangle et donc dans le champ de bits du répertoire. Les $n-k$ copies restantes doivent être placées dans la liste chaînée si cette dernière n'est pas pleine.

4.4.2 Évaluation théorique du coût matériel

Afin d'évaluer le coût de notre solution, nous proposons de nous intéresser au coût matériel du bloc *tiling*, qui correspond au cœur de l'algorithme **combinatoire**. Ce coût matériel dépend du nombre d'entrées n. Il s'agit donc d'une fonction de n qui peut être estimée en fonction du nombre de comparateurs, soustractions et LUT ou multiplications.

Nombre de comparateurs

Le bloc de calcul des rectangles correspond à un arbre de $k-1$ comparateurs pour chacun des éléments du quadruplet $(x_{min}, y_{min}, x_{max}, y_{max})$. Ainsi le nombre de compara-

teurs pour les blocs de calcul des rectangles est la somme des $4 \times (k-1)$ comparateurs. Le bloc d'élagage nécessite un comparateur soit pour trier les dimensions dans le cas de la LUT, soit pour comparer le résultat de la multiplication à la constance C. Le nombre de blocs de calcul des rectangles correspond aux combinaisons de 2 à n éléments, soit $\sum_{k=2}^{n}\binom{n}{k}$. Le nombre total de comparaisons est donc de :

$$\sum_{k=2}^{n}\binom{n}{k} \times (4 \times (k-1) + 1).$$

Cette formule peut être réécrite de la façon suivante :

$$\begin{aligned}
&\sum_{k=1}^{n}\binom{n}{k} \times (4 \times (k-1) + 1) - \binom{n}{1} \\
&= 4\sum_{k=1}^{n}\binom{n}{k} \times k - 3\sum_{k=1}^{n}\binom{n}{k} - n \\
&= 4\sum_{k=0}^{n}\binom{n}{k} \times k - 3\sum_{k=0}^{n}\binom{n}{k} + 3 \times \binom{n}{0} - n \\
&= 4\sum_{k=0}^{n}\binom{n}{k} \times k - 3\sum_{k=0}^{n}\binom{n}{k} + 3 - n.
\end{aligned}$$

En utilisant la formule du binôme de Newton [1], cette formule peut être réduite à :

$$\begin{aligned}
&4 \times n \times 2^{n-1} - 3 \times 2^{n} - n + 3 \\
&= (2 \times n - 3) \times 2^{n} - n + 3.
\end{aligned}$$

On remarque que la complexité du point de vue du nombre de comparateurs croît de manière exponentielle en fonction du nombre d'entrées n. Il faut donc minimiser le nombre d'entrées du bloc *tiling*.

Nombre de soustractions

Le bloc *selection* est le seul à réaliser des soustractions, il y en a deux par bloc soit :

$$\sum_{k=2}^{n}\binom{n}{k} \times 2.$$

1. $(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x^{n-k}y^k$ où $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Cette formule peut également être développée puis réduite de la façon suivante :

$$\begin{aligned}
&2\sum_{k=0}^{n}\binom{n}{k} - 2\times\binom{n}{1} - 2\times\binom{n}{0} \\
&= 2\sum_{k=0}^{n}\binom{n}{k} - 2\times n - 2 \\
&= 2\times 2^{n} - 2\times n - 2 \\
&= 2^{n+1} - 2\times(n+1).
\end{aligned}$$

Nombre de multiplications ou de LUT

Le dernier paramètre à évaluer pour déterminer le coût matériel du bloc combinatoire dépend du choix entre la multiplication ou la LUT dans le bloc d'élagage. Quel que soit le choix d'implémentation, il y a soit une multiplication soit une LUT par bloc d'élagage soit un total de :

$$\sum_{k=2}^{n}\binom{n}{k}.$$

Cette formule est proche de celle du nombre de soustractions et peut également être réduite de la façon suivante :

$$\begin{aligned}
&\sum_{k=0}^{n}\binom{n}{k} - \binom{n}{1} - \binom{n}{0} \\
&= \sum_{k=0}^{n}\binom{n}{k} - n - 1 \\
&= 2^{n} - n - 1.
\end{aligned}$$

4.5 Conclusion

Dans ce chapitre, nous avons proposé une nouvelle représentation de la liste des copies appelée DCC. Cette représentation est composée d'un rectangle correspondant à un sous-ensemble de copies et une liste chaînée pour les copies ne pouvant être placées dans le rectangle. La position et la forme du rectangle évoluent au cours de l'exécution, il s'agit donc d'un protocole dynamique. Chaque entrée du répertoire permet de mémoriser :

— la position et la forme du rectangle,
— la liste exacte des copies comprises dans le rectangle grâce à un champ de bits,
— les copies n'appartenant pas au rectangle à l'aide d'un pointeur vers une entrée du tas contenant le premier élément de la liste chaînée des copies.

Afin de limiter les ressources nécessaires au stockage de la liste chaînée, un seuil T définit le nombre maximum d'éléments de cette liste. Le nombre d'entrées dans le tas est lui aussi borné, il contient au plus H éléments.

Le cœur de cette proposition réside dans l'algorithme de détermination de l'origine et de la forme du rectangle cohérent appliqué à chaque nouvel accès mémoire. Nous avons proposé trois algorithmes : **idéal**, **premier touché** et **combinatoire**. L'algorithme **idéal** permet d'obtenir la borne supérieure des performances du protocole DCC. Cet algorithme est très coûteux en ressources et/ou en temps d'exécution. L'algorithme **premier touché** est un algorithme conservateur. L'origine du rectangle de forme 1×1 est déterminée par les coordonnées de la première copie ayant effectué une requête de lecture. Puis, à chaque nouvel accès, le rectangle est agrandi pour englober la nouvelle copie. L'aire de ce rectangle ne doit pas dépasser la taille du champ de bits. Enfin, nous avons proposé un algorithme **combinatoire** dont les ressources matérielles sont limitées. Cet algorithme possède deux modes :

Optimal : dans ce mode le placement du rectangle est identique à l'algorithme **idéal** ;

Sous-optimal : les copies du rectangle de cohérence sont un seul et même ensemble dont le poids correspond au nombre de copies du rectangle, les autres entrées du bloc sont utilisées par les copies de la liste chaînée ainsi que par la nouvelle copie.

Le choix entre les deux modes dépend du nombre de copies et du nombre d'entrées du bloc combinatoire. La qualité des résultats obtenus grâce à ces trois algorithmes est évaluée dans la suite de cette thèse.

Chapitre 5: Méthode d'évaluation de la représentation de la liste des copies

Sommaire

5.1 Méthodes de simulation existantes **47**
5.1.1 Simulations par interprétation d'instructions 48
5.1.2 Simulations basées sur l'injection de traces 50
5.2 Flot de simulation **51**
5.3 Extraction des requêtes d'accès mémoire **52**
5.3.1 Choix des paramètres de gem5 52
5.3.2 Modifications apportées à gem5 53
5.3.3 Choix des applications 53
5.3.4 Simulation et extraction du trafic 54
5.4 PyCCExplorer : un simulateur haut niveau pour la cohérence de caches **55**
5.4.1 Architecture logicielle 56
5.5 Choix des métriques et instrumentations du simulateur **59**
5.5.1 Latence d'accès aux données 59
5.5.2 Trafic induit par la cohérence de caches 60
5.5.3 Évaluation du nombre de messages de diffusion 60
5.6 Conclusion **60**

Dans ce chapitre, nous allons nous intéresser aux différentes méthodes d'évaluation de la liste des copies des protocoles de cohérence de caches afin de comparer les différentes solutions. Dans un premier temps, nous présenterons les méthodes de simulation utilisées pour l'exploration architecturale. Nous ne présentons pas les techniques analytiques qui ne sont pas comprises dans le spectre de nos travaux. Dans un second temps, nous présenterons notre méthode d'évaluation de la représentation de la liste des copies.

5.1 Méthodes de simulation existantes

Il existe plusieurs méthodes de simulation pour l'exploration architecturale. Dans ce chapitre, nous allons nous focaliser sur les simulations par interprétation d'instructions et les simulations basées sur l'injection de traces.

5.1.1 Simulations par interprétation d'instructions

On appelle simulation par interprétation d'instructions toutes les techniques de simulation qui exécutent une à une les instructions que le processeur doit effectuer. Ces simulations peuvent être qualifiées de simulation par instruction (ou *instruction accurate*). Nous allons nous intéresser aux limites de ce type de simulation lors d'exploration architecturale pour des architectures *manycores*.

Le simulateur Wisconsin Wind Tunnel [Wis] a été utilisé dans les années 1990 pour l'exploration architecturale. Mukherjee *et al.* [MH94] utilisent ce simulateur pour l'évaluation des protocoles de cohérence de caches avec répertoire pour une architecture de 128 cœurs. Afin d'évaluer les protocoles de cohérence, seulement trois applications sont utilisées : `Ocean` et `Barnes` provenant de la suite Splash 2 [WOT+95], ainsi qu' `Appbt` provenant de la suite NAS [BBB+91]. Les auteurs précisent qu'ils n'ont utilisé que trois applications à cause du temps de simulation. Dans le cas d'une exploration architecturale, plusieurs simulations d'une même application mais avec des différences au niveau de l'architecture sont nécessaires : par exemple avec un protocole de cohérence de caches différent ou encore avec des paramètres du protocole de cohérence de caches différents comme une variation de la taille de la liste chaînée pour la représentation basée sur ces dernières. Le temps de simulation est donc une limite importante pour ce simulateur.

Les auteurs de [GdMP08] utilisent leur propre plateforme composée de plusieurs éléments de la bibliothèque SocLib [SoC], ces composants sont décrits avec une précision *Cycle Approximate*, *Bit Accurate* (CABA). Dans le cadre de leur travaux, les auteurs effectuent une comparaison des politiques d'écriture des caches pour une architecture *manycore* composée de 64 cœurs utilisant un réseau sur puce ou en anglais *Network on Chip* (NoC). Dans leur papier seulement deux applications, `Ocean` et `Water` de la suite Splash 2 [WOT+95], sont exécutées. Dans sa thèse [GM09], l'auteur principal précise qu'il a utilisé seulement 6 applications, entre autres, pour que le temps d'exécution sur leur plateforme reste abordable. Là encore le temps de simulation est une des limites pour cette technique de simulation.

D'autres études architecturales ont été menées en utilisant des plateformes décrites en SystemC comme par exemple le simulateur Noxim [FPP08, CMM+16] ou encore celui utilisé dans [LPB06]. Il existe plusieurs niveaux de description d'architecture avec SystemC. Les travaux de Loghi *et al.* utilisent SystemC précis au niveau signal, à part pour le simulateur de jeu d'instructions qui lui l'est au niveau cycle d'horloge. Toutes les simulations faites avec SystemC présentent l'inconvénient d'être lentes. L'une des explications est que les modélisations faites avec SystemC prennent en compte tous les états transitoires ce qui augmente considérablement le temps de simulation. Ces méthodes ne sont donc pas adaptées à l'exploration architecturale rapide.

Il existe des simulateurs complets d'architectures qui modélisent l'ensemble d'une architecture. L'un des simulateurs les plus connus est gem5 [BBB+11] qui possède plusieurs modes de simulations dont un complet qui permet de simuler l'ensemble du système, c'est-à-dire le processeur, les mémoires, le réseau ainsi que les périphériques. Les différents modes de gem5 et plus particulièrement leur précision ont été étudiées

par Butko *et al.* [BGOS12]. La figure 5.1 nous montre le compromis à réaliser entre la vitesse de simulation et la précision des différents modes de gem5. On remarque qu'il existe plusieurs modèles de CPU, deux modes de simulation et plusieurs modèles pour les mémoires. Les deux modes de simulation sont : *SE* pour *System Emulation* et *FS* pour *Full System*. Ici, nous nous intéressons aux simulations de systèmes complets et donc au mode *Full System*. De plus, il existe plusieurs modèles pour la hiérarchie mémoire : le système classique, le mode ruby qui permet une modélisation précise des différentes mémoires et, enfin, le mode ruby avec garnet [AKPJ09] qui permet une simulation précise du réseau sur puce. Pour évaluer la représentation de la liste des copies et de manière plus générale les protocoles de cohérence de caches avec gem5, il faut utiliser au minimum le mode ruby. Ce dernier correspond à une description plus détaillée des mémoires et donc du protocole de cohérence. Le problème avec le mode ruby est que le temps de simulation augmente considérablement. De plus, gem5 n'est pas exécuté de manière parallèle : la simulation de n cœurs se fait sur un seul cœur physique ce qui ralentit d'autant la simulation pour les architectures *manycores*. Ainsi, il devient difficile d'utiliser gem5 pour l'exploration architecturale des solutions liées au passage à l'échelle des protocoles de cohérence de caches.

Processor		Memory System		
CPU Model	System Mode	Classic	Ruby	
			Simple	Garnet
Atomic Simple	SE	Speed		
	FS			
Timing Simple	SE			
	FS			
InOrder	SE			
	FS			
O3	SE			
	FS			Accuracy

Figure 5.1 – Compromis entre temps de simulation et précision pour les différents modes de gem5 extrait de [BGOS12]

Il existe d'autres simulateurs d'architectures présentant les mêmes limites que gem5 comme par exemple Simics avec le plugin Flexus [Fle] ou bien Graphite [MKK+10]. Flexus n'a pas été mis à jour depuis 2012, il est difficile d'en trouver la documentation, et peu de travaux publiés l'utilisent. Nous n'avons donc pas retenu ce simulateur. Graphite est un simulateur d'architectures multicoeurs proche de gem5 développé au MIT par une partie des auteurs qui ont développé le protocole de cohérence Ackwise [KMP+10]. Ces simulateurs présentent les même caractéristiques que gem5 : le temps de simulation est important lorsque l'on simule une architecture *manycore* à cause de la précision des

simulations.

Les simulations par interprétation d'instructions sont des simulations décrites avec un niveau élevé de précision. Toutes les simulations décrites précédemment présentent un inconvénient majeur : le temps de simulation est trop important pour permettre une exploration architecturale dans de bonnes conditions.

5.1.2 Simulations basées sur l'injection de traces

Pour gagner en temps de simulation, une solution est de décrire de manière moins précise l'architecture tout en permettant l'évaluation des différentes architectures proposées. Dans le cas des simulations basées sur l'injection de traces, seuls les composants concernés directement par l'exploration architecturale sont modélisés. Dans le cadre de nos travaux, les simulations doivent décrire au minimum le protocole de cohérence de caches et en particulier la représentation de la liste des copies. Pour cela, nous pouvons modéliser seulement les caches et le réseau. Dans ce cas, les requêtes normalement envoyées par le processeur sont émises par un générateur de trafic. Ces simulations sont connues sous le nom de simulation basée sur des traces ou *trace-driven simulations* [UM97].

Générateurs de trafic aléatoire

Les générateurs de trafic peuvent injecter différents types de trafics. La génération de trafic aléatoire est facile à réaliser, mais le trafic généré est éloigné de la réalité. En effet, la distribution des messages de cohérence n'est pas uniforme. C'est pourquoi, les générateurs de trafic peuvent également produire du trafic qui suit une autre distribution qui sera, elle, caractérisée à l'aide des statistiques de distribution sur ce genre de messages comme c'est le cas dans l'article [SWP06] ou encore [BJ14]. Dans l'article [SWP06], le modèle utilisé comporte seulement quatre cœurs qui communiquent à l'aide d'un bus alors que les architectures *manycores* actuelles utilisent un NoC. Le deuxième article [BJ14] simule des architectures plus modernes basées sur des NoC mais dans leur travaux les auteurs ont utilisé seulement 16 cœurs. De plus, les générateurs de trafic caractérisés par des statistiques ne sont pas adaptés aux simulations pour la cohérence de caches car les accès mémoires évoluent au cours de la durée de vie d'une application. Il est donc difficile de générer un trafic proche d'un trafic provenant d'une simulation exécutant une application.

Injection de traces provenant d'une simulation précise

Afin de reproduire un comportement proche de celui d'une application, une autre proposition est d'utiliser un trafic provenant d'une simulation précise [WJ90]. Dans ce cas, le générateur de trafic ne génère pas un trafic mais injecte les traces provenant de la simulation précise. Ces simulations permettent donc d'injecter un trafic réel même si certaines propriétés sont propres à chaque exécution. En effet, pour effectuer l'opération $c = a + b$, il faut d'abord lire les données a et b, réaliser l'addition et écrire le résultat dans c. Or, si lors de la simulation précise, la donnée a était présente dans un cache alors que dans la simulation basée sur l'injection des traces, cette donnée se trouve uniquement dans la mémoire principale alors la donnée a mettra plus de temps pour être disponible

et l'écriture du résultat dans c peut avoir lieu avant même que la valeur de a soit connu, ce qui n'est pas possible lors d'une exécution réelle. Néanmoins, lorsque l'on ne s'intéresse pas aux résultats de l'application mais seulement aux messages générés, cette variation de comportement n'est pas un problème et les simulations avec injection de traces provenant d'une simulation précise restent une bonne approximation pour un trafic réel.

Les simulations avec injection de trafic réel nécessitent une simulation précise mais lente afin d'extraire le trafic qui sera utilisé pour toutes les simulations suivantes. Nous n'avons pas trouvé de travaux utilisant le trafic provenant d'une simulation précise pour l'injecter dans un modèle haut niveau de cache pour permettre l'évaluation des protocoles de cache et plus particulièrement de la représentation de la liste des copies. Nous pensons que cette méthodologie de simulation permet d'obtenir rapidement les performances d'une représentation de la liste des copies par rapport à d'autres représentations. Dans cette thèse, nous avons choisi d'implémenter un simulateur basé sur l'injection de traces afin de classer différentes représentations de la liste des copies.

5.2 Flot de simulation

Nous proposons une méthode de simulation basée sur l'injection de traces en trois étapes, comme décrit dans la figure 5.2 :

- étape 1 : extraction de traces provenant d'une simulation précise,
- étape 2 : simulation à l'aide d'un modèle haut niveau de cache,
- étape 3 : analyse des résultats.

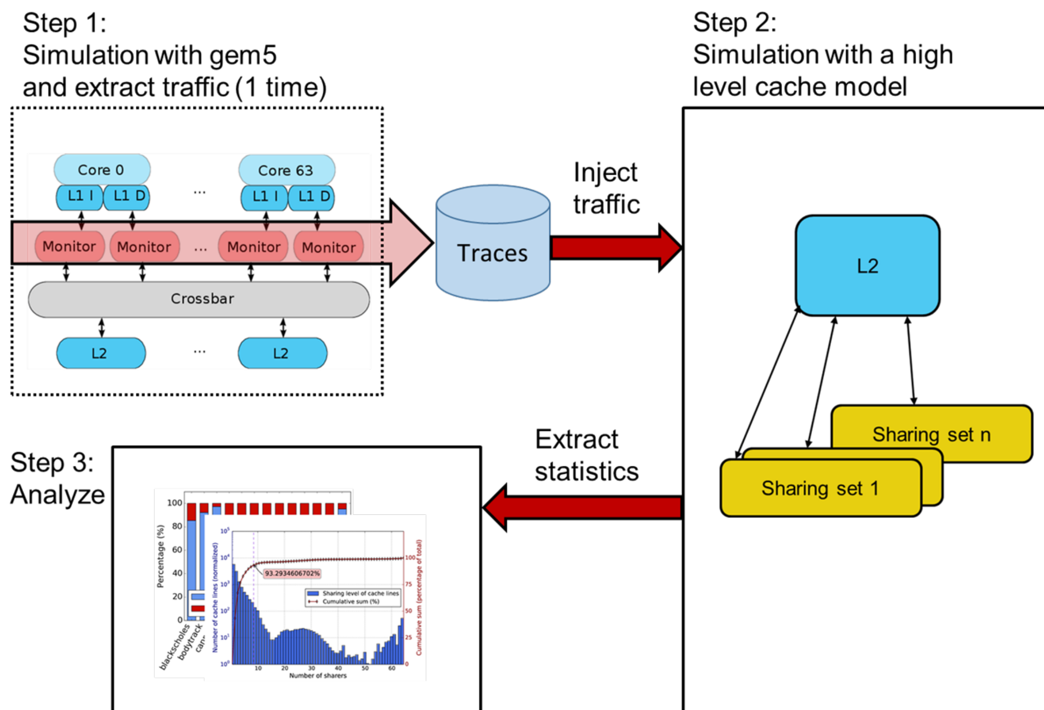


Figure 5.2 – Flot de simulation en trois étapes

La première étape est réalisée une seule fois et permet d'obtenir le trafic provenant d'une simulation réelle. Cette étape est décrite dans la section 5.3.

La deuxième étape constitue l'une des contributions majeures de cette thèse. Il s'agit du simulateur de cache à haut niveau qui est décrit dans la section 5.4.

Enfin, la dernière étape concerne l'analyse des résultats faite à l'aide des métriques obtenues lors des simulations à l'aide des instrumentations réalisées dans le simulateur. Ces dernières sont détaillées dans la section 5.5.

5.3 Extraction des requêtes d'accès mémoire

Dans le cadre de cette thèse, nous avons choisi d'extraire les requêtes d'accès à la mémoire à l'aide du simulateur gem5. Nous avons choisi ce simulateur, car il s'agit du simulateur de référence en architecture. Il est utilisé par exemple par les auteurs de [YWG+15, ASP17]. Ce simulateur est la fusion du simulateur m5 [BDH+06] et du modèle mémoire provenant de GEMS [MSB+05]. Ce dernier est majoritairement utilisé pour l'évaluation des protocoles de cohérence de caches comme dans les articles suivants : [RAG10, CRG+11, APJ09a, APJ09b, DCS+14, PG08, KR13, RK12, BMW06, HDH11, HDH09].

5.3.1 Choix des paramètres de gem5

Le simulateur gem5 dispose de plusieurs modes comme nous l'avons vu dans la section 5.1.1. Pour l'extraction des traces, nous utilisons gem5 avec le modèle mémoire classique car l'une des principales différences avec le mode ruby (un modèle plus précis de la hiérarchie mémoire) est que le temps pour satisfaire les requêtes de cohérence n'est pas modélisé. De plus, cette latence dépend du protocole de cohérence de caches et nous allons utiliser ces traces pour les injecter dans un modèle à haut niveau qui simule plusieurs protocoles de cohérence de caches. Les résultats de latence que l'on pourrait obtenir avec le mode ruby de gem5 ne seront corrects que pour le protocole de cohérence simulé sur gem5. Notre simulateur permet d'obtenir la latence pour chaque représentation de la liste des copies sans biais induit par la modélisation de la latence dans la simulation précise d'où sont extraites les traces.

Table 5.1 – Paramètres du simulateur gem5

CPU	64 cores Alpha 2 GHz
L1 instructions	32 kB, 2 way set associative
L1 data	64 kB, 2 way set associative
Shared L2	256 kB × 64, 8 way set associative
Block size	64 B
Network	Full crossbar

Les paramètres de gem5 que nous avons choisis se trouvent dans le tableau 5.1. Nous avons limité nos simulations à 64 cœurs car il s'agit d'une limite actuelle de gem5. Nous avons utilisé le modèle de CPU *timing*. Nous avons effectué des simulations avec des cœurs de type Alpha, ARM et x86. Nous avons finalement réalisé nos analyses

avec des cœurs Alpha car ces derniers sont les plus stables, ce qui est confirmé dans la documentation de gem5.

5.3.2 Modifications apportées à gem5

Nous avons apporté deux modifications au simulateur gem5 :
- l'ajout de requêtes de type *clean eviction*,
- l'ajout de moniteurs entre les caches L1 et le réseau.

Les requêtes de type *clean eviction* sont envoyées par les caches L1 lorsqu'ils évincent une ligne de cache. En effet, les caches L2 de gem5 n'étant pas inclusifs, ils n'ont pas besoin d'être notifiés lorsqu'un cache L1 évince une ligne. Lorsqu'on utilise un protocole de cohérence de caches basé sur un répertoire, le cache L2 est inclusif. Dans ce cas toutes les lignes de caches présentes dans un cache L1 sont également présentes dans le cache L2. En effet, la liste des copies est maintenue au niveau du cache L2, il faut donc que ce dernier sache quelles sont les lignes présentes dans les caches de niveau inférieur. Cependant, dans notre modélisation, pour des raisons de performances, comme nous le verrons plus tard, les caches L1 sont modélisés de manière très abstraite. Le fait de savoir quelles sont les données dans chaque cache est le moyen, pour ce modèle abstrait, de savoir quels caches L1 peuvent répondre aux requêtes de cohérence.

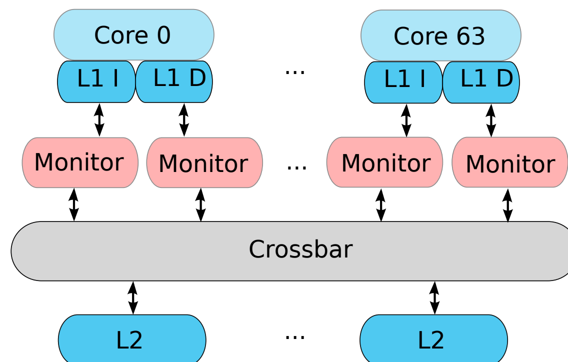


FIGURE 5.3 – Architecture simulée sur gem5 avec l'ajout des moniteurs

La deuxième modification concerne l'ajout de moniteurs entre les caches L1 et le réseau pour enregistrer les messages de cohérence. La figure 5.3 représente l'architecture simulée avec les moniteurs entre les caches L1 et le réseau. On peut remarquer qu'il existe un moniteur par cache L1, soit un total de 128 moniteurs, car chaque cœur dispose d'un cache L1 pour les instructions et un autre pour les données.

5.3.3 Choix des applications

Afin d'évaluer l'efficacité de la représentation de la liste des copies des protocoles de cohérence de caches, nous avons exécuté plusieurs applications provenant des suites de *benchmark* Splash 2 [WOT+95] et PARSEC [BKSL08]. Ces suites de *benchmark* contiennent des applications dont le calcul et l'accès aux données sont parallèles.

Ils sont largement utilisés par la communauté pour l'évaluation des architectures *many-cores*. Nous avons sélectionnés 13 applications qui s'exécutent sur Alpha dont la liste se trouve dans le tableau 5.2.

Table 5.2 – Liste des applications

Application	Taille du problème
PARSEC	
blackscholes	65 536 options
bodytrack	4 images, 4 000 particules
canneal	200 000 éléments
dedup	31 MB
ferret	64 requêtes, 13 787 images
fluidanimate	100 000 particules, 5 images
freqmine	500 000 transactions
streamcluster	8 192 points
vips	2 336 x 2 336 pixels
x264	32 images, 640 x 360 pixels
Splash2	
fft	4 194 304 complex doubles
ocean (contiguous)	256 x 256 ocean
ocean (non contiguous)	256 x 256 ocean

De plus, nous avons choisi de nous focaliser sur la *Region Of Interest* (ROI), c'est-à-dire la section du code où les calculs parallèles sont effectués en omettant l'initialisation. Des informations supplémentaires sur la ROI comme le temps passé avant, pendant et après la ROI sont données dans l'article [SR16]. Afin d'isoler la ROI, la suite d'applications PARSEC permet l'ajout d'instructions pour le simulateur gem5 au début et à la fin de la ROI. Lors des simulations avec gem5, un point de sauvegarde est créé au début de la ROI et le simulateur est quitté à la fin de la ROI. Ainsi, nous pouvons extraire les messages de cohérence seulement à l'intérieur de la ROI.

5.3.4 Simulation et extraction du trafic

La simulation de l'ensemble des applications sur le simulateur gem5 a été réalisée en deux étapes. La première simulation est faite sans les moniteurs et permet de créer le point de sauvegarde au début de la ROI. La deuxième simulation est réalisée à partir du point de sauvegarde et avec les moniteurs, elle permet d'obtenir le trafic sur le réseau entre les caches L1 et L2. Lorsque le simulateur gem5 restaure son état à partir d'un point de sauvegarde, les caches redémarrent à froid, c'est-à-dire qu'ils sont vides. Afin d'obtenir le même résultat, nous avons choisi de commencer nos simulation avec des caches également vide. Dans ce cas, les caches de notre simulateur seront mis à jour à l'aide des messages de cohérence extraits de la simulation avec gem5.

Le temps nécessaire pour effectuer l'ensemble de ces simulations avec gem5 est indiqué dans le tableau 5.3. On peut remarquer que les applications présentent une grande

Table 5.3 – Temps d'exécution de certaines applications PARSEC et Splash2 sur gem5

Applications	**Initialisation et ROI**	**ROI avec moniteurs**
blackscholes	1h09	0h46
bodytrack	2h28	1h32
canneal	3h08	0h49
dedup	13h37	14h55
ferret	2h36	2h24
fluidanimate	2h05	1h44
streamcluster	16h01	17h59
vips	5h37	5h17
x264	2h16	2h08
fft	1h45	0h54
ocean (contiguous)	1h33	1h30
ocean (non contiguous)	1h31	1h28

différence de temps d'exécution. On peut également noter que l'application `canneal` passe plus de temps à initialiser les données nécessaires au calcul que pour le calcul en lui-même, ce qui est confirmé dans l'article [SR16]. Pour l'ensemble des applications, il faut 105 heures pour réaliser la première étape de simulation afin de créer point de sauvegarde au début de la ROI ainsi que la deuxième étape avec les moniteurs pour extraire le trafic. Ce temps de simulation peut paraître non négligeable mais cette étape est réalisée une seule fois.

Les moniteurs nous permettent d'obtenir les informations nécessaires pour rejouer l'exécution d'un protocole de cohérence de caches au niveau L2. Pour chaque requête, les informations mémorisées sont les suivantes :

Tick : temps auquel la requête a été envoyée ;

Source : identifiant du cache L1 qui a envoyé la requête ;

Address : adresse mémoire de la donnée concernée par la requête ;

Request : type de requête de cohérence.

À l'issue des simulations avec gem5, ces informations sont enregistrées dans une base de données de 58 Go, cette dernière se trouvant sur un disque dur. Cette base de données est utilisée par notre simulateur pour la suite de notre approche.

Les informations pourraient être extraites d'un autre simulateur que gem5. Pour cela, il faudra mettre en forme les données et en particulier le type de requête afin que notre simulateur reconnaisse les requêtes.

5.4 PyCCExplorer : un simulateur haut niveau pour la cohérence de caches

Le but de notre simulateur est de modéliser la latence des requêtes de cohérence et de comparer les différentes représentations de la liste des copies. Une fois que l'on a

extrait le trafic avec un simulateur précis, gem5 dans notre cas, ce trafic est injecté dans un simulateur de cache à haut niveau que nous avons conçu et développé dans le cadre de cette thèse.

5.4.1 Architecture logicielle

La figure 5.4 représente l'architecture logicielle de notre simulateur de cache que nous avons nommé PyCCExplorer pour *Python Cache Coherence Explorer*. Ce dernier a été développé en python qui, grâce aux nombreuses bibliothèques disponibles, permet de réaliser des programmes complexes en un nombre relativement limité de lignes de code. Le nombre de lignes de code de chaque module développé est noté *loc* (pour *lines of code*) dans la figure.

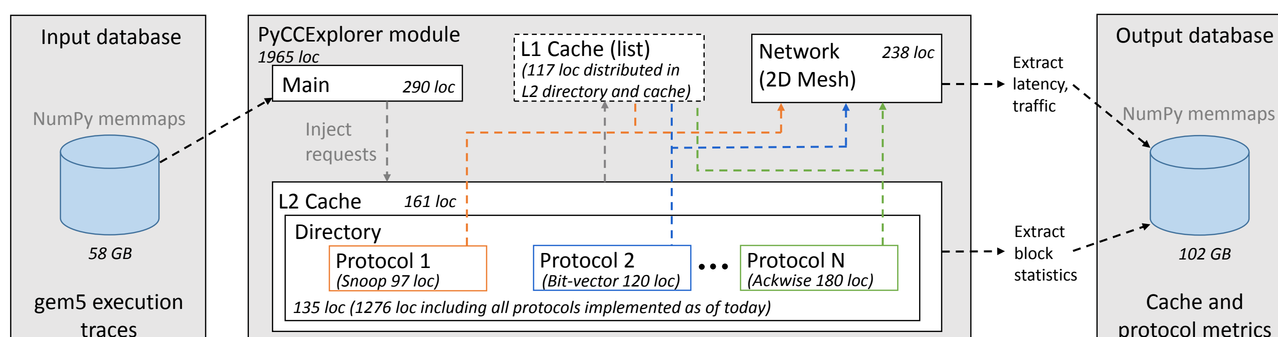


FIGURE 5.4 – Architecture logicielle de PyCCExplorer

PyCCExplorer est composé des modules suivants :
- un ou plusieurs caches L2 ;
- un réseau ;
- plusieurs caches L1.

Le module de cache L2, le cœur de ce simulateur, contient un répertoire et lors des simulations, plusieurs représentations de la liste des copies peuvent être simulées.

Les interactions entre les différents modules sont les suivantes :

1. La base de données contenant le trafic obtenu à l'aide de gem5 est lue par le programme principal ;
2. Ensuite, les messages de cohérence, c'est-à-dire les traces, sont injectés sur le réseau ;
3. Les messages sont reçus par le cache L2 qui consulte son répertoire pour la ligne de cache concernée par la requête ;
4. Puis, selon le protocole de cohérence, un ou plusieurs messages sont envoyés sur le réseau à destination d'un ou plusieurs caches L1 ;
5. Lorsque les caches L1 reçoivent une requête, ceux dont la donnée est présente répondent à la requête ;
6. Et finalement, le demandeur reçoit la donnée. Au passage, le cache L2 et la liste des copies du répertoire ont été mis à jour.

Les modules sont décrits de manière plus précise dans les sous-sections suivantes.

Programme principal (main)

Ce module a deux fonctionnalités principales : la première est de configurer l'architecture simulée et la deuxième est de lire la base de données contenant les traces pour les injecter dans le simulateur.

Les configurations possibles concernent le nombre de cœurs simulés, le type de réseau, les représentations de la liste des copies, la taille et le nombre de caches L2 ainsi qu'un certain nombre de paramètres de ce cache comme le niveau d'associativité, la taille des lignes de caches, ou la politique de remplacement des lignes de cache. Dans notre cas, le cache L2 a les mêmes caractéristiques que celui simulé dans gem5, hormis le fait que le nôtre est inclusif. Cette différence n'est pas critique car la taille du cache L2 permet d'inclure toutes les lignes présentes dans les caches L1. Nous avons modélisé un cache L2 inclusif car cela est nécessaire lorsque le protocole de cohérence de caches utilise un répertoire. En effet, il existe une entrée du répertoire par ligne présente dans le cache L2.

En dehors de la configuration de l'architecture, la fonction principale de ce module lit la base de données et transmet les requêtes de cohérence au bon destinataire, c'est-à-dire l'un des caches L2. Dans gem5, les caches L2 sont entrelacés, c'est-à-dire que plusieurs lignes de caches qui se suivent ne sont pas dans le même banc. Pour savoir à quel cache correspond une donnée à partir de son adresse, on calcule l'adresse modulo le nombre de cache L2 pour obtenir l'indice du cache L2. Lorsque l'on utilise un NoC, la latence de lecture correspond au nombre de nœuds traversés et donc cette latence n'est pas la même pour chaque donnée. Dans ce cas, on peut s'interroger sur l'effet de l'entrelacement des adresses dans les caches L2. Une des solutions souvent utilisée est de mettre la donnée dans le cache le plus proche du demandeur. Dans ce cas, la fonction pour déterminer où se trouve la donnée est plus compliquée et nécessite l'ajout de matériel supplémentaire. Lors du développement de PyCCExplorer, nous avons choisi d'implémenter ces deux stratégies, l'entrelacement (*interleaving*) et le premier accès (*first touch*). Dans la suite de cette thèse, lorsqu'aucune précision n'est apportée, nous considérons que la fonction utilisée est celle qui entrelace les adresses comme dans gem5.

Cache L2

Le module cache L2 est composé de plusieurs ensembles (*set*) eux-mêmes composés de plusieurs voies (*way*). Les fonctions principales de ce module sont l'ajout et la suppression d'une ligne dans le cache. Dans ces deux cas, le cache doit calculer à partir de l'adresse de la donnée l'ensemble dans lequel se trouve cette donnée, puis les voies sont testées.

Dans notre simulateur, nous effectuons les simulations à haut niveau. Pour cela, nous avons réalisé plusieurs simplifications. L'une d'elles concerne les données contenues dans le cache ou encore la machine à états de haut niveau dont nous parlerons dans la sous-section suivante. Pour nos simulations, nous nous intéressons seulement aux messages de cohérence et à l'utilisation du cache : les données ne sont pas importantes, il faut seulement savoir quelles sont les lignes de cache occupées. Nous modélisons donc uniquement le comportement du cache relatif aux adresses, c'est-à-dire à quel endroit

doit être rangé et recherché une donnée.

Tous les caches L2 possèdent également au moins un répertoire, chacun avec sa représentation de la liste des copies. Cette dernière est mise à jour à chaque ajout ou suppression d'une copie, ainsi que lorsqu'une ligne de cache est évincée.

Répertoire et représentation de la liste des copies

Nous avons défini une interface à implémenter pour chaque représentation de la liste des copies. Les fonctions à implémenter sont les suivantes :

Read : demande de lecture d'un cache L1 ;

ReqEx : demande de lecture exclusive d'un cache L1 ;

Delete : suppression d'une ligne d'un cache L1 ;

Writeback : suppression d'une donnée modifiée par un cache L1, le niveau supérieur de la hiérarchie mémoire doit être mis à jour ;

Clean : suppression d'une copie dans la liste des copies.

Ces différentes fonctions sont appelées selon les requêtes lues. Dans la suite de ce paragraphe, nous indiquons l'association entre les fonctions et les requêtes provenant du simulateur gem5. Lorsque le simulateur injecte une requête de lecture (*ReadReq*), la fonction **Read** est utilisé. La fonction **ReadEx** est utilisé lorsque le simulateur reçoit une requête d'écriture (*WriteReq*), une requête de mise à jour afin de passer la ligne en mode exclusif (*UpgradeReq*), une requête de mise à jour pour la ligne après un *store conditional* (*SCUpgradeReq*), et finalement une requête de lecture exclusive (*ReadExReq*). La fonction **Writeback** est utilisée seulement avec les requêtes demandant la mise à jour du niveau supérieur de la hiérarchie mémoire (*Writeback*). La requête *Clean* que nous avons ajoutée dans gem5 pour indiquer l'éviction la ligne du L1 sans mise à jour de la mémoire correspond à la fonction **Clean** de notre simulateur. La fonction **Delete** est utilisée par notre simulateur lorsqu'il faut évincer une ligne du cache L2.

L'implémentation de ces fonctions est propre à chaque représentation de la liste des copies, ces détails sont donnés dans le chapitre 6. Quelle que soit la représentation de la liste des copies simulée, la machine à états de haut niveau (MOESI dans le cadre de cette thèse) est implémentée dans ce module. Nous avons choisi de ne modéliser que les états stables du protocoles car nous souhaitons réaliser une simulation à haut niveau et les états transitoires ne sont pas nécessaires. Cette simplification nous permet d'effectuer des simulations rapides en réduisant le nombre d'états possibles à 5.

Réseau

Dans PyCCExplorer, nous avons choisi de modéliser un réseau de type 2D maillé implémentant l'algorithme de routage *X-first* (X en premier). Nous avons fait ce choix car c'est l'un des réseaux le plus utilisé dans les architectures actuelles. Malgré ce choix, une autre topologie de réseau est possible. Il suffit pour cela de créer une nouvelle classe qui hérite de **network** et qui implémente l'ensemble des fonctions.

Dans le réseau que nous avons implémenté, nous avons modélisé deux canaux : l'un pour les requêtes et l'autre pour les réponses. Nous avons défini quatre types de messages pouvant circuler sur ce réseau : les requêtes, les réponses, les messages de diffusion qui attendent une réponse de tous les cœurs et ceux qui n'attendent qu'un nombre donné de réponses. Le type des messages de diffusion dépend uniquement du protocole de cohérence de caches. De plus, nous avons ajouté une modélisation qui correspond à un support matériel aux messages de diffusion : un message de diffusion est envoyé et le réseau se charge de répliquer ce message pour le transmettre à tous les caches. Ce support influence donc le trafic et la latence.

Nous n'avons pas modélisé la contention sur le réseau : nous montrons par la suite que notre taux d'injection des messages dans le réseau nous permet de considérer que nous disposons d'un réseau parfait. Néanmoins, lorsqu'un message est envoyé en diffusion, le cache demandeur ne peut traiter qu'une seule réponse à chaque cycle. On ajoute donc un cycle à la latence de la requête pour chaque réponse supplémentaire.

Cache L1

Les caches L1 ne sont pas réellement modélisés dans notre simulateur car le trafic extrait de gem5 est pris en sortie des caches L1. La modélisation de leur comportement n'est donc pas nécessaire. En revanche, le trafic représente seulement les requêtes de cohérence, les caches L1 envoyant également les réponses aux requêtes. Ces réponses dépendent du protocole de cohérence de caches. Les extraire avec gem5 pour les introduire dans le trafic n'a donc pas de sens. Pour savoir quel cache peut satisfaire les requêtes de cohérence, pour chacune des données, les caches L1 sont représentés sous forme d'une liste de caches L1 contenant cette donnée. Cette liste contient les mêmes informations que la liste des copies lorsqu'elle est complète et précise. Ainsi, cette liste permet de générer les réponses des caches L1 quand cela est nécessaire.

5.5 Choix des métriques et instrumentations du simulateur

Pour permettre l'évaluation de la liste des copies, nous avons retenu plusieurs métriques dont la latence, le trafic et le nombre de message de diffusion qui sont évalués à l'aide d'instrumentations dans PyCCExplorer. Les instrumentations nécessaires sont réalisées à partir de compteurs et sont décrites dans les sous-sections suivantes.

Notre simulateur peut également fournir des informations qui ne sont pas liées à la représentation de la liste des copies comme par exemple le taux de partage, la durée de vie d'une ligne de cache, le taux de remplissage du cache L2. L'ensemble de ces métriques n'est pas nécessaire pour notre étude, mais il nous semble important de mentionner que notre simulateur peut être utilisé plus largement pour l'exploration architecturale des caches.

5.5.1 Latence d'accès aux données

Une des métriques importantes pour choisir un protocole de cohérence de caches et en particulier sa représentation de la liste de copies est la latence entre le moment

où un cache L1 envoie une requête et le moment où il reçoit la réponse. Pour mesurer la latence, nous considérons le temps mis par la requête pour accéder au répertoire du cache L2, puis la latence qui dépend du protocole et de la représentation de la liste des copies. Dans tous les cas, la latence entre deux nœuds correspond au nombre de routeurs du NoC traversés par le message. De plus, nous avons considéré qu'un seul message par émetteur peut être envoyé à chaque cycle. Par exemple, lorsqu'une ligne partagée par trois caches L1 est évincée du cache L2, il faut trois cycles pour envoyer les trois requêtes aux caches L1 pour qu'ils évincent cette ligne. Les caches L1 doivent évincer cette ligne car notre cache L2 est inclusif.

5.5.2 Trafic induit par la cohérence de caches

Nous avons également choisi d'évaluer le nombre de messages de cohérence envoyés sur le réseau. En effet, on souhaite que le réseau ne sature pas à cause des requêtes de cohérence. Pour évaluer le trafic induit par la cohérence de caches, nous avons ajouté des compteurs sur chaque canal qui comptent au niveau de chaque routeur le nombre de messages les traversant. Pour avoir une image de l'état du réseau au cours de la durée de vie de l'application, cette métrique est évaluée sur des fenêtres de 10 000 cycles.

5.5.3 Évaluation du nombre de messages de diffusion

Une autre métrique qui nous semble importante pour choisir une représentation de la liste des copies est liée à la consommation énergétique. Dans notre modélisation, nous nous concentrons sur les messages nécessaires à la cohérence sans modéliser l'énergie nécessaire à l'envoi de messages. Néanmoins, le trafic généré nous permet d'avoir une idée de la consommation énergétique. De plus, le nombre de messages envoyés en diffusion influence largement cette consommation. En effet, les messages de diffusion augmentent la consommation énergétique et peuvent saturer le réseau. C'est pourquoi nous avons ajouté un compteur du nombre de messages de diffusion envoyés, pour pouvoir évaluer les représentations de la liste des copies qui limitent ce type de messages très gourmant en énergie.

5.6 Conclusion

Dans ce chapitre, nous avons présenté un état de l'art sur les simulations par interprétation d'instructions ainsi que sur celles basées sur l'injection de traces. Dans cette thèse, nous avons proposé une méthodologie basée sur l'injection de traces afin d'évaluer la représentation de la liste des copies dans les protocoles de cohérence de caches. Cette méthode est réalisée est trois étapes :

- étape 1 : extraction de traces provenant d'une simulation précise ;
- étape 2 : simulation à l'aide d'un modèle haut niveau de cache ;
- étape 3 : analyse des résultats.

Dans cette thèse, l'extraction du trafic provient du simulateur gem5 sur lequel nous avons exécuté 13 applications de référence provenant de PARSEC et Splash2. Cette étape, qui prend environ 5 jours de temps de calcul et produit 58 giga-octets de traces, est réalisée une seule fois. La deuxième étape est réalisée à l'aide de PyCCExplorer, le simulateur de cache défini et développé dans le cadre de cette thèse. Ce simulateur

rejoue le comportement d'un cache L2 en modélisant seulement le cache L2 et son répertoire, le réseau et le protocole de cohérence de caches. Afin de réduire le temps de simulation, nous avons modélisé ces différents composants à haut niveau. De plus, nous avons modélisé seulement les états stable du protocole de cohérence de caches MOESI. Enfin, la dernière étape permet de traiter et d'interpréter les résultats obtenus à l'aide d'instrumentations dans PyCCExplorer. Notre simulateur permet d'obtenir entre autres la latence, le trafic, le nombre de messages envoyés en diffusion ou encore le taux de succès d'accès aux caches (*hit*).

Chapitre 6: Évaluation de l'influence de la représentation de la liste des copies

Sommaire

6.1 Expérimentations avec le simulateur PyCCExplorer 63
- 6.1.1 Implémentation dans le simulateur 64
- 6.1.2 Exploration architecturale avec PyCCExplorer 66
- 6.1.3 Classement des représentations de la liste des copies 70
- 6.1.4 Validation du simulateur PyCCExplorer 76

6.2 Évaluation du protocole DCC avec PyCCExplorer et implémentation matérielle 79
- 6.2.1 Simulation avec PyCCExplorer 79
- 6.2.2 Implémentation matérielle du protocole DCC 89

6.3 Conclusion 91

Dans ce chapitre, nous nous intéressons aux performances de la représentation de la liste des copies des protocoles de cohérence de caches. Dans un premier temps, nous présenterons la validation de notre méthodologie de simulation basée sur l'injection de traces. Dans un second temps, nous nous focaliserons sur la validation du protocole DCC et nous le comparerons à d'autres représentations de la liste des copies. Nous nous intéresserons aux performances de l'implémentation matérielle du protocole DCC.

6.1 Expérimentations avec le simulateur PyCCExplorer

Afin de valider notre simulateur, nous avons implémenté plusieurs représentations de la liste des copies :
- espionnage (*snoop*) ;
- champ de bits complet (*full bit-vector*) ;
- Ackwise ;
- liste chaînée.

Nous avons utilisé PyCCExplorer pour classer les représentations de la liste des copies selon plusieurs critères comme la latence, le trafic généré ou encore le nombre de diffusion. Notre simulateur peut également être utilisé pour l'exploration architecturale des paramètres d'un protocole, par exemple le seuil d'Ackwise pour passer dans le mode dégradé.

6.1.1 Implémentation dans le simulateur

Pour chaque représentation de la liste des copies, nous avons choisi un comportement qui est le reflet d'une implémentation matérielle. Les sections suivantes détaillent le comportement des trois représentations de la liste des copies qui nous servent de référence dans le simulateur afin de classer d'autres représentations de la liste des copies.

6.1.1.1 Protocole basé sur l'espionnage (snoop)

Nous avons choisi d'inclure dans notre simulateur le protocole basé sur l'espionnage que nous avons présenté dans la section 3.1 du chapitre 3. Avec ce protocole, lorsqu'un cache L1 souhaite lire une donnée, il envoie une requête de lecture au cache L2. Ce dernier envoie la requête en diffusion à tous les caches L1. Le cache L2 attend les réponses de l'ensemble des caches L1 puis la réponse est transférée au cache L1 ayant effectué la requête. Dans ce cas, la latence entre la requête et la réception de la réponse correspond à la latence du cache L1 le plus proche du cache L2. Lorsque la seule copie valide est dans le cache L2, ce dernier doit quand même attendre les réponses de tous les caches L1. Dans ce cas, la latence correspond à la latence maximale. Enfin, si la donnée n'est pas présente dans la hiérarchie mémoire, alors le cache L2 envoie une requête à la mémoire principale. Dans ce dernier cas, la latence correspond à la latence d'accès à la mémoire principale, nous avons choisi de fixer arbitrairement cette latence à 100 cycles.

Lors d'une demande de lecture exclusive, le cache L2 doit attendre la réponse de tous les caches L1. La latence correspond donc à la latence de la réponse du dernier cache L1. Lorsque la donnée est présente seulement dans la mémoire principale, la latence est la même que pour les lectures, il faut attendre la réponse de tous les caches L1 et transférer la requête à la mémoire principale, ce qui correspond à 100 cycles dans notre modélisation.

6.1.1.2 Répertoire avec champ de bits complet (bit-vector)

La représentation de la liste des copies utilisant un champ de bits complet correspond à la représentation la plus précise. Cette représentation a été présentée dans la problématique (chapitre 2 section 2.3). Un répertoire au niveau du cache L2 permet de mémoriser cette liste des copies. Lorsqu'un cache L1 souhaite lire une donnée, il en fait la requête auprès du cache L2. Ce dernier consulte son répertoire afin de savoir quels caches L1 possèdent cette donnée. Le cache L1 qui possède la donnée et qui permet de minimiser la latence entre lui, le répertoire et le cache L1 demandeur est sélectionné. Puis, la requête de lecture est transmise par le cache L2 et le cache L1 lui renvoie la réponse. La réponse contenant la donnée est ensuite transférée au cache L1 demandeur. Lorsqu'aucun cache L1 ne possède la donnée mais qu'elle est présente dans le cache L2 alors la latence correspond à la latence entre le cache L1 et le cache L2. Enfin, lorsque la donnée n'est présente ni dans le cache L2 ni dans un cache L1, une requête est envoyée à la mémoire principale. Dans ce cas, la latence correspond à la latence d'accès à la mémoire, 100 cycles dans notre cas.

Lorsque la requête est une demande de lecture exclusive, le cache L2 consulte la liste des copies présente dans le répertoire et un message d'invalidation est envoyé à chacune des copies. Un message d'acquittement est alors envoyé par chaque copie (c'est-à-dire chaque cache L1) au répertoire. Dans ce cas, la latence correspond à la latence du dernier message d'acquittement reçu par le cache L2.

6.1.1.3 Répertoire Ackwise

Nous avons avons également sélectionné un protocole de cohérence de caches avec un répertoire limité. Nous avons choisi le protocole Ackwise, que nous avons présenté dans le chapitre 3 dans la section 3.2. Le répertoire Ackwise est un répertoire limité à k copies. Le seuil (k) est l'un des paramètres du protocole qu'il faut fixer lors de l'implémentation. Lorsque le nombre de copies dépasse le seuil, on passe dans un mode imprécis en mettant le bit G (pour *global*) à 1. Dans ce cas, seulement le nombre de copies est mémorisé dans le répertoire et les messages de cohérence sont envoyés en diffusion. Lorsqu'un cache L1 souhaite lire une donnée, il envoie une requête au cache L2 et ce dernier consulte son répertoire pour savoir quel cache L1 doit satisfaire la requête. Dans le protocole Ackwise, un des caches est sélectionné pour être le *keeper* et c'est ce dernier qui doit satisfaire les requêtes de lecture. La latence correspond à la latence entre le cache demandeur, le cache L2, le *keeper* et le cache demandeur. En effet, dans ce protocole la réponse ne repasse pas par le répertoire, seul un message d'acquittement est envoyé au répertoire. Lorsque le message d'acquittement est reçu par le répertoire, ce dernier doit être mis à jour. Si le bit G vaut 1, alors le compteur du nombre de copies est incrémenté. Si le bit G vaut 0 et qu'il reste un emplacement vide dans la liste des copies, alors le CID du cache demandeur est ajouté à cette liste. Lorsque le bit G vaut 0 mais qu'il ne reste plus d'emplacement vide dans la liste des copies, alors on passe dans le mode imprécis. Dans ce cas, le bit G est mis à 1 et les champs *sharers* sont réinitialisés. De plus, le champ $sharer_k$ est utilisé comme compteur de copies. Enfin, lorsque la donnée n'est présente dans aucun cache L1, le comportement est le même que celui du protocole basé sur un répertoire avec un champ de bits complet : si la donnée est dans le cache L2, ce dernier satisfait la requête, sinon elle est transférée à la mémoire principale avec une latence de 100 cycles.

Lors d'une demande de lecture exclusive, le comportement est différent selon le mode utilisé. Lorsque le bit G vaut 0, un message d'invalidation est envoyé à chacune des copies présentes dans la liste des copies. En revanche, lorsque le bit G vaut 1, le message d'invalidation est envoyé en diffusion à tous les caches L1. Dans ce cas, le répertoire attend un nombre d'acquittements égal au nombre de copies.

6.1.1.4 Répertoire utilisant une liste chaînée

Nous avons également implémenté un répertoire utilisant une liste chaînée qui est un protocole de cohérence de caches dynamique dont le comportement a été décrit dans le chapitre 3 section 3.3. La représentation de la liste des copies basée sur une liste chaînée possède deux paramètres : la taille maximale pour la liste chaînée et le nombre d'entrées du tas utilisées pour mémoriser les listes chaînées. Lorsqu'un cache L1 souhaite lire une donnée, il envoie une requête au cache L2 qui consulte son répertoire.

Ce répertoire permet de savoir si cette ligne est en mode diffusion ou non. Lorsque cette ligne est en mode précis, le répertoire contient le CID de la première copie, ainsi qu'un pointeur vers une entrée du tas pour les éléments suivants de la liste chaînée. Ainsi lors d'une lecture, la requête est transférée à cette première copie qui renvoie la réponse au cache L2 puis la réponse est transférée au cache demandeur. La latence correspond donc à la latence entre le cache demandeur, le cache L2, le cache L1 correspondant à la première copie, le cache L2 et enfin le cache L1 demandeur. Le répertoire est mis à jour avec la nouvelle copie qu'il faut ajouter dans la liste chaînée. Si cette dernière a déjà la taille maximale ou que le tas est plein, alors la ligne passe en mode diffusion. Lors de ce changement de mode, l'ensemble de la liste chaînée est effacé. Dans ce cas, un compteur permet de savoir combien de caches possèdent cette ligne et un cache L1 est sélectionné pour devenir responsable de la cohérence. Lors d'une requête de lecture pour une ligne en mode diffusion, le cache responsable de la cohérence répond à la requête. La mise à jour du répertoire est faite en incrémentant le compteur de copies. Enfin, lorsque la donnée n'est pas présente dans les caches, la requête est transmise à la mémoire principale, ce qui correspond à une latence de 100 cycles.

Lors d'une requête de lecture exclusive, la requête est envoyée au cache L2. En consultant le répertoire, ce dernier envoie un message d'invalidation à l'ensemble des éléments de la liste chaînée lorsqu'on est en mode précis. Le répertoire attend un acquittement de l'ensemble des copies puis la réponse est envoyée au cache demandeur. Lorsque la ligne est en mode diffusion, un message d'invalidation est envoyé en diffusion à l'ensemble des caches L1 et le répertoire attend un nombre d'acquittements égal au nombre de copies.

6.1.2 Exploration architecturale avec PyCCExplorer

Dans cette partie, nous présentons les résultats obtenus avec notre méthodologie de simulation. Notre méthodologie permet d'effectuer des explorations architecturales pour l'étude des paramètres des protocoles. Nous avons choisi d'illustrer cette fonctionnalité avec l'exploration architecturale des paramètres pour le protocole Ackwise et celui basé sur une liste chaînée.

6.1.2.1 Exploration architecturale pour Ackwise

Afin de réaliser l'exploration architecturale dans le but de fixer la valeur du seuil du protocole Ackwise, nous nous sommes focalisés sur la latence.

La figure 6.1 montre la latence moyenne pour un seuil fixé à 2, 3, 4, 5, 8 et 64 pour les applications extraites de Splash 2 et PARSEC. Sur cette figure, les latences sont celles relatives au protocole Ackwise pour lequel le seuil est fixé à 5. Nous pouvons observer que les variations selon la valeur du seuil sont faibles. On remarque sur cette figure que la latence est plus élevée avec un seuil bas. Nous pouvons également observer que plus le seuil est élevé plus la représentation de la liste des copies est précise, mais après un seuil égal à 5, les améliorations sont nettement moins importantes. Un compromis raisonnable pour une architecture de 64 cœurs est un seuil d'une valeur autour de 5.

Ces résultats sont cohérents vis-à-vis des résultats obtenus par les auteurs d'Ackwise [KMP+10], ce qui confirme que notre approche de simulation permet l'exploration

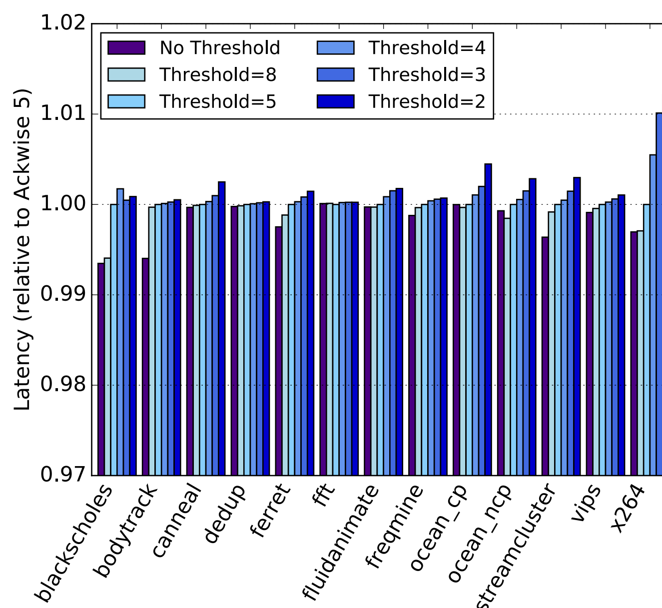


Figure 6.1 – Latence moyenne relative à Ackwise avec un seuil de 5 pour plusieurs valeurs de seuil

archicteturale des paramètres d'un même protocole.

6.1.2.2 Exploration architecturale du protocole basé sur une liste chaînée

Les paramètres du protocole basé sur une liste chaînée sont le seuil, c'est-à-dire la taille maximale de cette liste chaînée, ainsi que le nombre d'entrées du tas. Nous avons évalué l'influence de ces deux paramètres.

Effet du seuil

Dans un premier temps, nous nous intéressons à l'effet du seuil (noté *threshold* sur les figures). Pour cela, nous avons effectué des simulations avec un tas dont la taille n'est pas limitée.

La figure 6.2 montre le pourcentage de lignes de cache en mode diffusion avec un tas infini pour les 13 applications que nous avons sélectionnées. Sur cette figure, on peut observer ce pourcentage pour différentes valeurs de seuil comprises entre 2 et 9. On remarque également que lorsqu'il n'y a pas de seuil, aucune ligne n'est en mode diffusion car l'ensemble des copies est rangée dans la liste chaînée. Sur cette figure, on observe également que le pourcentage de lignes de cache en mode diffusion diminue lorsque le seuil augmente. Lorsque le seuil est égal à 2, on note que les performances sont nettement moins bonnes que lorsque le seuil est plus élevé. Une explication de cette observation est que le nombre de copies pour chaque ligne de cache est plus élevé que 2, ainsi un nombre important de lignes de cache sont en mode diffusion alors qu'avec une valeur de seuil plus élevée, toutes les copies pourraient être rangées dans la liste chaînée.

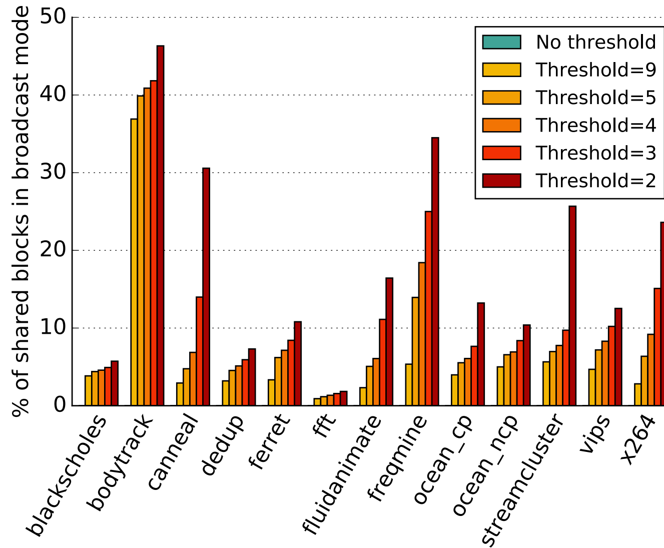


FIGURE 6.2 – Pourcentage de lignes de cache en mode diffusion avec un tas infini pour la représentation basée sur une liste chaînée

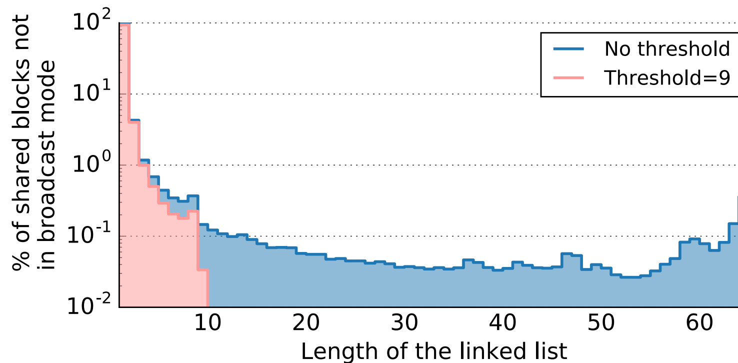


FIGURE 6.3 – Pourcentage des lignes qui ne sont pas en mode diffusion pour chaque taille de liste chaînée

La figure 6.3 détaille la répartition des lignes qui ne sont pas en mode diffusion avec un seuil égal à 9 et sans seuil, calculé en pourcentage de l'ensemble des lignes de cache. Dans ce dernier cas, il n'y a aucune ligne en mode diffusion et cette répartition nous renseigne donc également indirectement sur le taux de partage des lignes de cache. A la différence de la figure 2.10 (présentée dans le chapitre 2) qui mesurait ce partage sur l'ensemble de l'exécution de l'application, la figure 6.3 mesure le partage en prenant des images instantanées du répertoire et tient donc compte des effets dynamiques, tels que les remplacemnts de ligne de cache ou les prises d'exclusivité par exemple. Nous remarquons que l'immense majorité des lignes de cache est partagée par moins de 8 coeurs comme nous l'avons observé dans le chapitre 2. Un pic pour des lignes partagées par tous les coeurs est également observé, ce qui est compatible avec les mesures de l'état de l'art [CRG+11]. L'utilisation d'un seuil ne change pas fondamentalement le profil de partage si ce n'est que, le passage en mode diffusion étant définitif, le pourcentage de

lignes avec une taille de liste proche du seuil est inférieur à la configuration sans seuil à cause d'un passage temporaire au delà du seuil.

Effet de la taille du tas

Le deuxième paramètre à étudier lors de l'exploration architecturale du protocole basé sur une liste chaînée est la taille du tas permettant de mémoriser les listes chaînées.

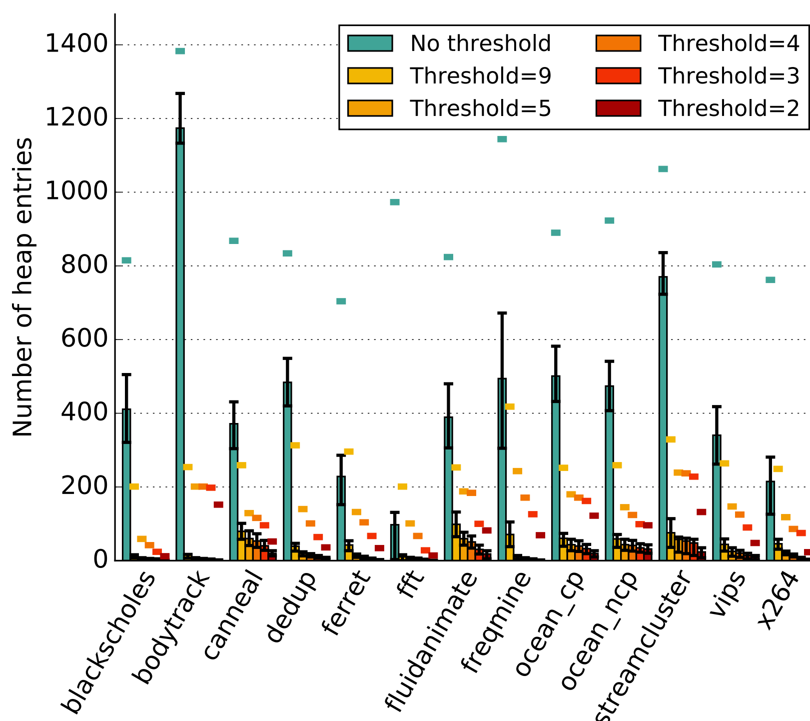
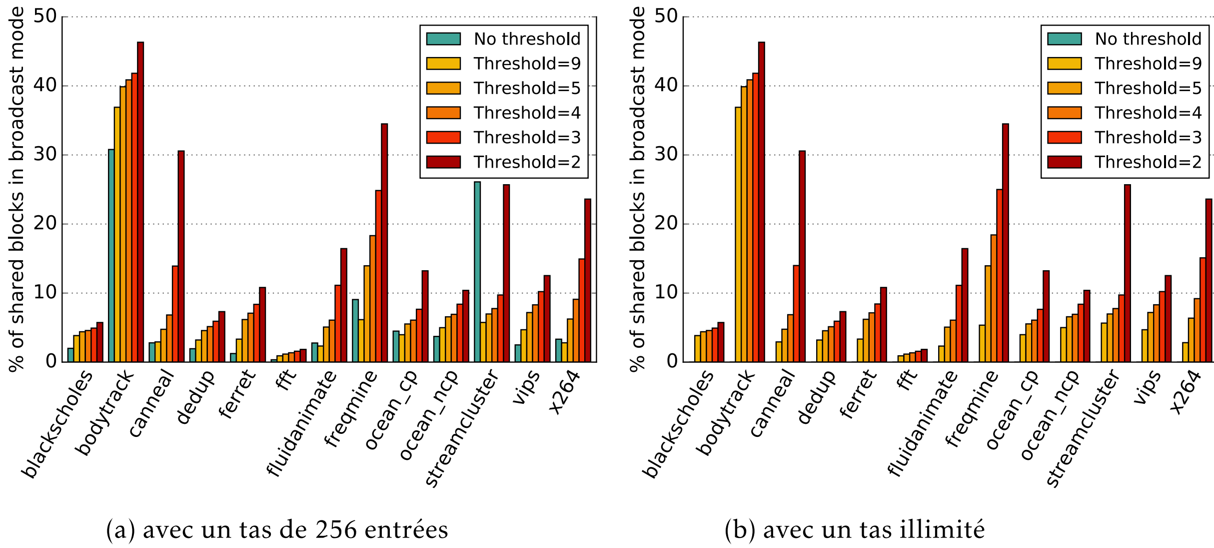


Figure 6.4 – Nombre d'entrées du tas utilisées pour la représentation avec une liste chaînée (avec un tas infini)

La figure 6.4 montre le nombre d'entrées du tas utilisées pour plusieurs applications et plusieurs valeurs de seuil. Sur cette figure, les barres représentent le nombre moyen d'entrées utilisées. On remarque que lorsqu'il y a un seuil, la moyenne du nombre d'entrées utilisées est inférieure à 100. De plus, sur cette figure apparaît également le premier et le troisième quartile. Lorsque le seuil est compris entre 2 et 9, le premier et le troisième quartile sont proches de la moyenne. Sur cette même figure, les points représentent le nombre d'entrées utilisées au maximum. Lorsqu'il y a un seuil, le nombre d'entrées maximales utilisées est rarement supérieur à 300. Sur cette figure, on remarque également que le nombre d'entrées utilisées dans le tas lorsqu'il n'y a pas de seuil est très largement supérieur au nombre d'entrées utilisées lorsqu'il y a un seuil.

Pour la suite de cette étude architecturale, nous proposons de fixer la taille du tas à 256 entrées, ce qui est supérieur à la moyenne du nombre d'entrées utilisées et également proche du maximum nombre d'entrées utilisées lorsqu'il y a un seuil. La figure 6.5a représente le pourcentage de lignes de cache en mode diffusion comme sur la figure 6.2,



(a) avec un tas de 256 entrées

(b) avec un tas illimité

Figure 6.5 – Pourcentage de lignes de cache en mode diffusion pour la représentation de la liste des copies avec et sans limite sur la taille du tas

mais cette fois, le tas est limité à 256 entrées. La figure 6.2 a été dupliquée à côté de cette nouvelle figure afin de faciliter la comparaison de ces deux figures. On remarque que le pourcentage de ligne en mode diffusion reste inchangé lorsqu'il existe un seuil. Lorsque qu'il n'y a pas de seuil, on observe sur cette nouvelle figure que certaines lignes sont en mode diffusion, ce qui n'était pas le cas avec un tas illimité. Dans ce cas, ces lignes sont passées en mode diffusion car le tas était plein. Lorsqu'il existe un seuil, le passage en mode diffusion est le résultat de deux situations : soit la liste chaînée est pleine, soit le tas est plein. Néanmoins, nous n'observons pas de différence notable induite par la limite du nombre d'entrées du tas lorsqu'il existe un seuil.

À l'aide de notre simulateur, nous avons pu mener à bien cette exploration architecturale et nous avons pu déterminer le nombre d'entrées nécessaires dans le tas afin de réduire le nombre de lignes en mode diffusion. Dans notre exemple, nous avons fixé cette taille à 256 entrées, ce qui correspond à 1/16ème du nombre de lignes de cache. Cette étude nous a également permis d'observer l'influence de la valeur du seuil de la liste chaînée.

6.1.3 Classement des représentations de la liste des copies

Notre méthode de simulation permet également de classer les représentations de la liste des copies selon un critère. Nous présentons ce classement pour deux métriques qui sont essentielles pour choisir une représentation de la liste des copies : la latence et le trafic.

6.1.3.1 Latence

La première métrique à laquelle nous nous intéressons pour évaluer les représentations de la liste des copies est la latence d'accès aux données.

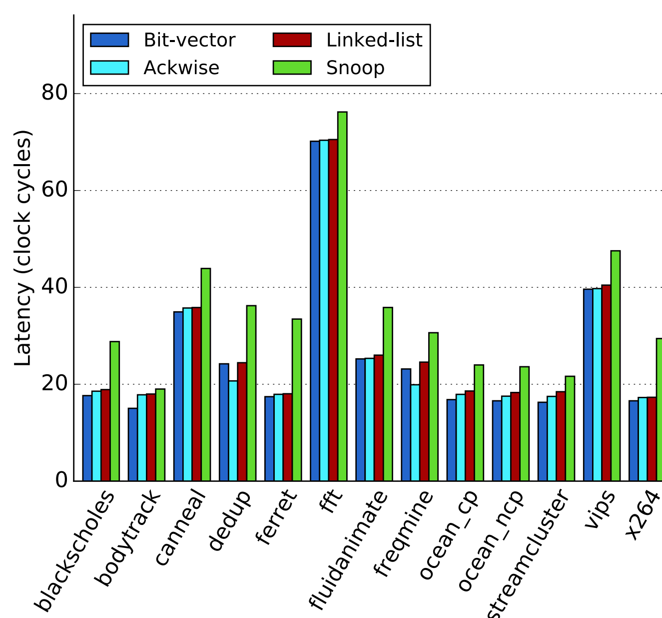


Figure 6.6 – Latence moyenne en nombre de cycles

La figure 6.6 montre la latence moyenne pour chaque application. Sur cette figure, nous pouvons remarquer que l'ordre est le même pour la majorité des applications : le protocole basé sur l'espionnage a toujours une latence plus élevée que les autres stratégies. On observe également que la représentation avec un champ de bits complet permet d'obtenir les meilleurs résultats en moyenne. En effet, cette représentation est la plus précise, ce qui permet de réduire la latence. Nous avons choisi de comparer la latence des autres représentations de la liste des copies à celle utilisant un champ de bits complet. En moyenne, la représentation de la liste des copies Ackwise a une latence supérieure de 2% à cette représentation de référence. Toujours par rapport à la latence du protocole utilisant un champ de bits complet, le protocole basé sur l'espionnage a une latence en moyenne supérieure de 43%. Enfin, la représentation avec une liste chaînée a une latence moyenne supérieure de 6%. La latence élevée du protocole basé sur l'espionnage s'explique car les messages sont envoyés en diffusion et chaque requête nécessite d'attendre les réponses de l'ensemble des destinataires. De plus, cette latence est sous-estimée car le trafic engendré par les messages de diffusion saturent le réseau, ce qui a pour conséquence d'augmenter la latence des requêtes de cohérence. Or notre modélisation du réseau ne prend pas en compte la contention et le comportement non linéaire de la latence qui en découle (voir section 6.1.3.2).

Sur cette figure, on observe également que pour les applications `dedup` et `freqmine`, Ackwise a une latence inférieure à celle de la représentation avec un champ de bits complet, ce qui est contre-intuitif car cette dernière représentation est la représentation la plus précise. Ce résultat s'explique car le nombre de requêtes de lecture exclusive est plus important que pour les autres applications (voir figure 6.7). En effet, lors d'une requête de lecture exclusive, les copies doivent être invalidées. Lorsque le mode diffusion

d'Ackwise est activé, l'invalidation est envoyée à l'aide d'un seul message de diffusion alors que dans le protocole avec un champ de bits complet, les messages d'invalidation sont envoyés un par un à chacune des copies.

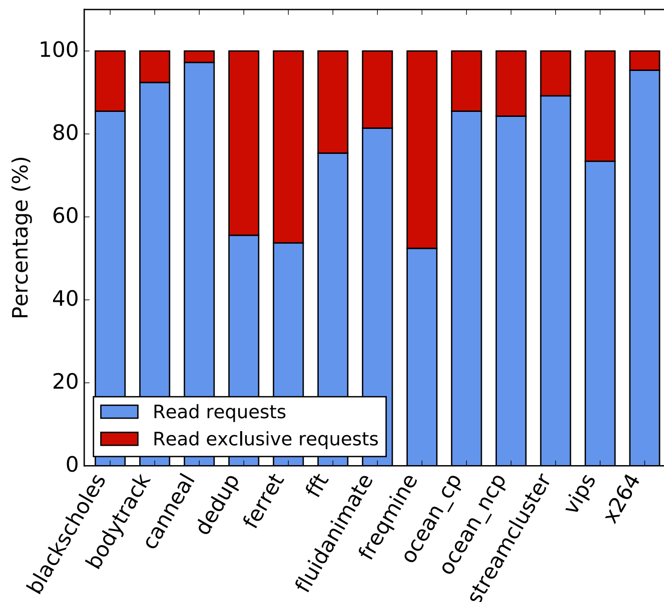


Figure 6.7 – Répartition des requêtes de lecture et de lecture exclusive

La répartition des requêtes de lecture et de lecture exclusive est représentée sur la figure 6.7. Cette figure a été obtenue en analysant les requêtes en entrée de PyCCExplorer. On remarque que pour la majorité des applications, le pourcentage de requêtes de lectures est supérieur à 75%, excepté pour les applications `dedup`, `ferret` et `freqmine` dont le nombre de lectures exclusives est proche du nombre de requêtes de lectures.

En plus d'un taux important de lecture exclusive, un taux de défaut de cache (*miss*) des requêtes dans le cache L2 élevé engendre une latence plus importante pour toutes les représentations de la liste des copies. En effet, lorsqu'il y a défaut de cache dans le cache L2, la requête est transmise à la mémoire principale ce qui correspond à une latence de 100 cycles dans notre simulateur. La figure 6.8 montre la répartition entre les requêtes qui font un succès et un défaut de cache. Pour les requêtes qui font un défaut de cache, on distingue deux catégories :

Compulsory **:** Ce sont les défauts de cache obligatoires qui correspondent au premier accès.

Capacity **et** ***conflict*** **:** Ce sont les défauts de cache lorsque le cache est plein ou que l'ensemble est déjà plein.

Sur la figure 6.8, deux applications se distinguent des autres : `blackscholes` et `fft`. `Blackscholes` possède un taux de défaut de cache obligatoire (*compulsory miss*) plus élevé que les autres applications. Ce résultat s'explique car cette application ne fait pas beaucoup d'accès mémoire. Ainsi, la proportion de défaut de cache correspondant au premier accès de chaque ligne de cache est plus importante. En revanche, `fft` se distingue

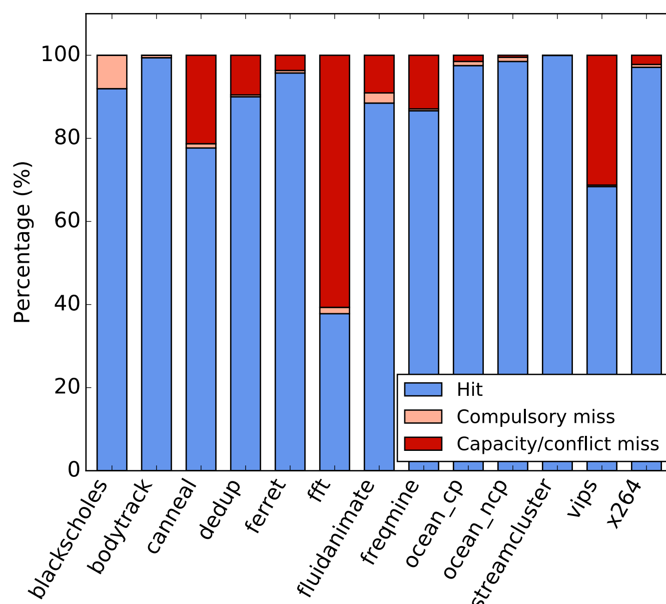


FIGURE 6.8 – Répartition du taux de succès/défaut de cache au niveau du L2

des autres applications pour une proportion de défaut de cache très importante, autour de 60%. Ainsi pour cette application, la latence élevée obtenue figure 6.6 s'explique par le nombre important de requêtes qui sont transmises à la mémoire principale.

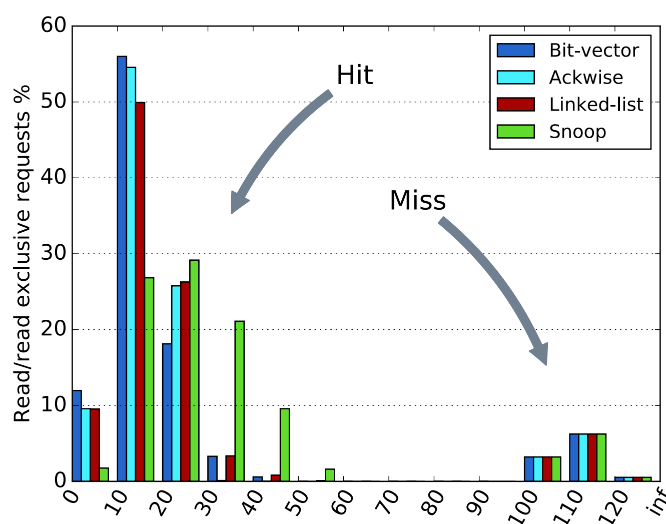


FIGURE 6.9 – Moyenne de la répartition de la latence des requêtes de lecture et de lecture exclusive pour l'ensemble des applications

La corrélation entre le taux de défaut de cache et la latence apparaît sur la figure 6.9. Cette figure représente la répartition de la latence pour les requêtes de lecture et de lecture exclusive pour l'ensemble des applications. Sur cette figure, on distingue deux groupes : le premier correspondant à une latence supérieur à 100 cycles, le deuxième

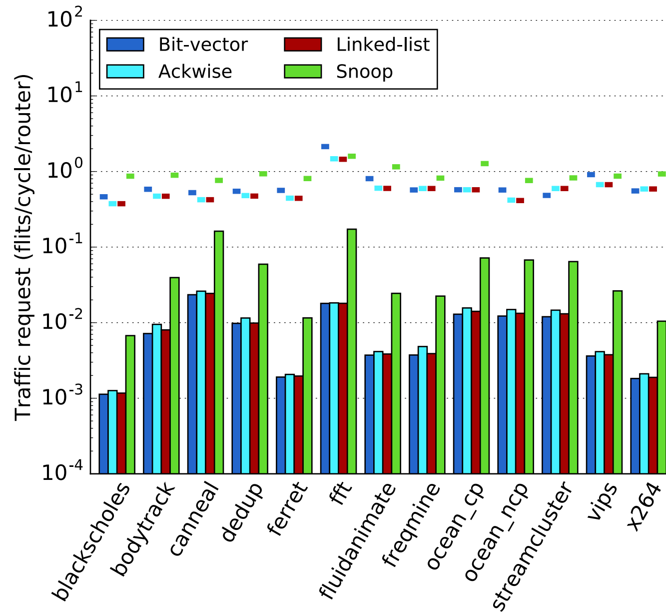
correspondant à une latence inférieure à 60 cycles. Le premier groupe correspond aux accès faisant un défaut dans le cache L2. On remarque que ce taux est le même pour toutes les représentations de la liste des copies. Pour le deuxième groupe correspondant aux accès faisant un succès, on distingue des différences selon la représentation de la liste des copies. On remarque que pour les représentations avec une liste chaînée, Ackwise et avec un champ de bits complet, 60% des requêtes ont une latence inférieure 20 cycles. Pour le répertoire basé sur l'espionnage, on observe un pic pour une latence comprise entre 20 et 30 cycles. Ces observations confirment que la latence augmente avec le nombre de messages envoyés en diffusion. Sur cette figure, on observe également que la représentation de la liste des copies avec un champ de bits complet possède une latence légèrement inférieure à celle des autres représentations. Cela peut être expliqué car le protocole avec un champ de bits complet choisit le meilleur cache L1 pour répondre aux requêtes. On remarque également que la représentation avec une liste chaînée a une latence un peu plus élevée qu'Ackwise et celle avec le champ de bits complet. Ce résultat s'explique par le nombre élevé de messages envoyés en diffusion, ce qui augmente la latence.

6.1.3.2 Trafic

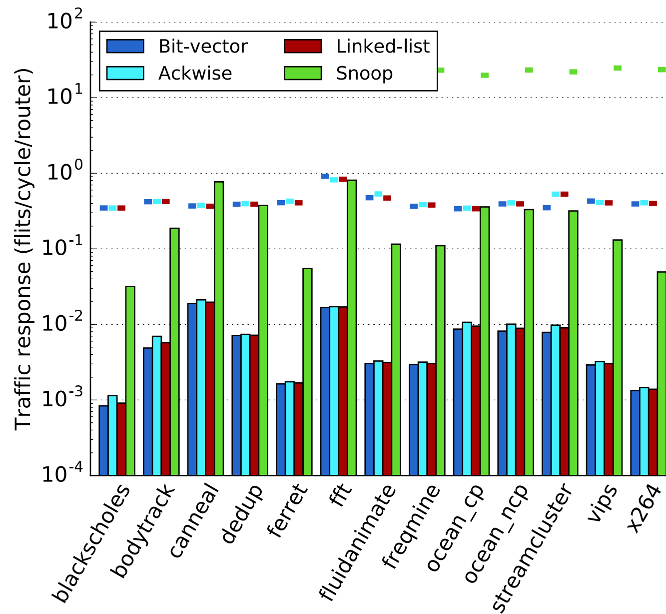
Nous nous intéressons à une deuxième métrique pour classer les représentations de la liste des copies : le trafic induit par la cohérence des caches.

Les figures 6.10a et 6.10b montrent ce trafic pour le canal des requêtes et celui des réponses. Pour illustrer le trafic, nous avons choisi de calculer la moyenne et le maximum pour des fenêtres de 10 000 cycles. Sur ces figures, nous observons que le protocole basé sur l'espionnage génère un trafic plus important que les autres protocoles. Ce trafic est plus élevé que les autres d'un ordre de grandeur pour le canal de requête et de deux ordres de grandeur pour le canal de réponse. Cette différence peut s'expliquer car les messages de cohérence sont envoyés en diffusion et chaque cache L1 répond, ce qui engendre un nombre de réponses plus important. En revanche, notre modélisation du réseau ne prend pas en compte la charge du réseau : nous avons supposé qu'il n'y a pas de contention sur le réseau. Les travaux d'Ogras *et al.* [OBM10] et de Foroutan *et al.* [FTP13] montrent que la latence de l'injection des messages dans le réseau est négligeable lorsque le taux d'injection est inférieur à 10%. Cette hypothèse n'est pas vérifiée dans le cas du protocole basé sur l'espionnage. Ainsi, la latence calculée précédemment a été sous-estimée, mais cela ne remet pas en cause le classement des différentes représentations de la liste des copies.

On remarque également des différences entre les deux canaux pour Ackwise et la représentation avec une liste chaînée. Ces observations s'expliquent car ces deux représentations utilisent des ressources limitées et possèdent un mode de représentation imprécis basé sur l'envoi de messages en diffusion. De plus, on remarque que la représentation avec un champ de bits complet génère un peu plus de messages qu'Ackwise et que la liste chaînée car ces deux dernières représentations bénéficient du support matériel pour les messages de diffusion. Par rapport au champ de bits complet, Ackwise génère un trafic plus important de 17% pour le canal de requête et de 16% pour le canal des réponses. La représentation avec une liste chaînée génère un trafic plus proche de



(a) Canal de requête



(b) Canal de réponse

Figure 6.10 – Moyenne et maximum du trafic pour Ackwise, champ de bits, liste chaînée et espionnage

celui du champ de bits complet : 5% de trafic supplémentaire pour le canal de requête et 6% pour celui des réponses. Le protocole basé sur l'espionnage génère un trafic plus important de 600% et 4 000% respectivement pour le canal de requête et pour le canal de réponse.

Lorsque l'on souhaite choisir une représentation de la liste des copies sur le critère du trafic généré, on note que le protocole basé sur l'espionnage ne doit pas être sélectionné. De plus, la représentation avec un champ de bits complet est la plus performante devant la représentation avec une liste chaînée et Ackwise.

6.1.4 Validation du simulateur PyCCExplorer

Afin de valider notre méthodologie de simulation basée sur l'injection de traces, nous avons comparé nos résultats à ceux obtenus avec des simulateurs plus précis. Nous avons également étudié les divergences avec la simulation de gem5 que nous avons utilisée pour extraire les traces. Enfin, nous présentons la complexité de PyCCExplorer en nous basant sur le nombre de lignes de code et le temps de simulation.

6.1.4.1 Comparaison entre PyCCExplorer et gem5 concernant l'évaluation de la représentation de la liste des copies

Afin de mesurer l'écart entre les simulations avec gem5 et les simulations avec PyCCExplorer, nous avons ajouté des compteurs dans notre simulateur pour compter les requêtes qui permettent de détecter que nous avons dévié de la simulation de gem5. Il existe deux raisons principales pour lesquelles notre modélisation diverge de gem5 : notre cache L2 est inclusif alors que celui de gem5 ne l'est pas ; la politique de remplacement des lignes de cache utilise un algorithme pseudo-aléatoire, ainsi la ligne sélectionnée peut être différente dans notre cache L2. Nous distinguons trois types de requêtes anormales :

- lecture d'une donnée déjà présente (*read already present*),
- éviction d'une ligne de cache inconnue (*eviction of unknown block*),
- éviction d'une ligne de cache d'un cache L1 inconnu (*eviction of L1 block*).

Lecture d'une donnée déjà présente

Les requêtes de lecture d'une donnée déjà présente correspondent à deux situations. La première situation arrive lorsqu'une requête de lecture provient d'un cache L1 qui, d'après le répertoire du cache L2, possède déjà cette donnée. Cela arrive lorsque le cache L2 de gem5 a évincé une ligne de cache alors que notre simulateur a choisi d'en évincer une autre. La deuxième situation est le pendant de la première situation pour les lectures exclusives, c'est-à-dire lorsqu'une requête de lecture exclusive arrive au cache L2 mais ce cache L1 est déjà marqué comme possédant l'exclusivité de cette ligne.

Éviction d'une ligne de cache inconnue

Le deuxième type de requête anormale arrive lorsqu'un cache L2 reçoit un message d'éviction d'une ligne de cache qui n'est pas présente dans le cache L2. Cette situation se produit lorsque notre cache L2 a évincé cette ligne de cache. Dans ce cas, notre cache L2 étant inclusif, un message d'invalidation a été envoyé à toutes les copies et ainsi nous divergeons de la simulation réalisée avec gem5.

Éviction d'une ligne de cache d'un cache L1 inconnu

Enfin, le dernier type concerne aussi les messages d'éviction, mais cette fois il s'agit des messages provenant d'un cache L1 qui n'est pas marqué comme partageant cette

ligne dans le répertoire du cache L2.

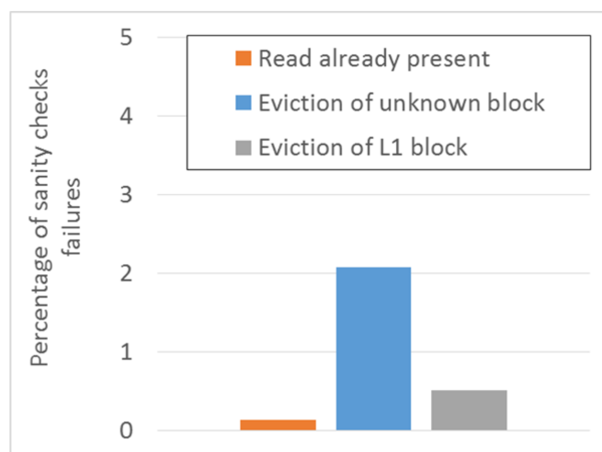


Figure 6.11 – Pourcentage des requêtes incohérentes en entrée de PyCCExplorer et provenant de gem5

La figure 6.11 montre le pourcentage de ces requêtes qui arrivent au niveau du cache L2. On remarque que seulement 0,14% des requêtes traitées par notre cache L2 sont des requêtes de lecture d'une donnée déjà présente. De plus, 2,08% des requêtes sont des requêtes d'éviction d'une ligne de cache inconnue et 0.51% des requêtes sont des requêtes d'éviction d'une ligne d'un cache L1 inconnu. Ainsi, le pourcentage de requêtes incohérentes est inférieur à 3%, ce qui est négligeable et nous permet d'affirmer que la modélisation de notre cache L2 ne diverge pas notablement du cache L2 modélisé dans le simulateur gem5.

6.1.4.2 Comparaison des résultats à ceux d'autres travaux

Afin de valider les résultats obtenus avec PyCCExplorer, nous les avons comparés à ceux obtenus avec des simulateurs plus précis. Nous avons comparé les résultats à ceux de Xu *et al.* [XDZY11] obtenus avec le simulateur Noxim [FPP08], un simulateur *cycle accurate* basé sur SystemC. Dans leurs travaux, les auteurs utilisent des applications provenant de la suite PARSEC et évaluent la latence du réseau, le temps d'exécution, le surcoût des messages de diffusion, ainsi que la consommation d'énergie. Dans leurs travaux, les auteurs comparent leur protocole à trois représentations que nous avons modélisées dans PyCCExplorer : le protocole basé sur l'espionnage, le répertoire avec un champ de bits complet et Ackwise. Ainsi, nous pouvons comparer nos résultats de la latence et nous obtenons le même classement pour ces représentations de la liste des copies, c'est-à-dire de la meilleure représentation à la moins bonne pour la latence : le champ de bits complet, Ackwise et l'espionnage. En revanche, nous avons remarqué que la latence du protocole basé sur l'espionnage est plus élevée dans leurs résultats. Cette différence peut être expliquée par deux raisons. Premièrement, dans leurs travaux, les auteurs simulent 1024 cœurs contre 64 cœurs dans nos simulations, ainsi l'impact des messages envoyés en diffusion est plus élevé lorsque le nombre de cœurs augmente. Deuxièmement, nous avons sous-estimé la latence induite par l'injection des messages

dans le réseau lorsque ce dernier sature, ce qui est le cas avec le protocole basé sur l'espionnage.

Nous avons également comparé nos résultats à ceux des auteurs ayant proposé Ackwise. Kurian *et al.* [KMP+10] utilisent le simulateur Graphite [MKK+10] et des applications provenant de Splash2 et PARSEC. Malheureusement, nous avons seulement une application en commun : `Ocean`. Dans leur travaux, les auteurs simulent deux architectures : l'une avec 64 cœurs et l'autre avec 1024 cœurs. L'architecture avec 64 cœurs est composée d'un réseau 2D maillé comme dans nos simulations, nommé EMesh dans leur article. Dans cet article, les auteurs présentent les performances de leur protocole Ackwise. Pour cela, ils ont réalisé une exploration architecturale du seuil d'Ackwise. Nous avons obtenu des résultats comparable aux leurs : des performances meilleures lorsque la valeur du seuil augmente mais des améliorations moins fortes lorsque le seuil dépassent 5.

6.1.4.3 Complexité de PyCCExplorer

Nous avons également évalué la complexité de notre solution sur deux critères : l'effort de développement et le temps de simulation. Notre outil est écrit en Python, ce qui permet d'écrire du code très compact. Dans PyCCExplorer, la modélisation du comportement du cache L2 est faite en 150 lignes, le réseau est décrit en 120 lignes et 150 lignes sont nécessaires pour chaque représentation de la liste des copies. Ainsi, lorsqu'un utilisateur veut ajouter une nouvelle représentation de la liste des copies, l'effort de développement est réduit car cela nécessite seulement 150 lignes de code. Par comparaison, la modélisation des caches avec ruby dans gem5 nécessite 2 800 lignes de C++ et 4 000 lignes de slicc pour le protocole MESI_Two_Level. Cette grande différence s'explique par plusieurs facteurs. Premièrement, le langage de programmation n'est pas le même et le langage python est très compact. Deuxièmement, la simulation est plus précise dans gem5 : il faut donc décrire plus précisément le comportement, en particulier pour les états transitoires, ce qui induit un nombre de lignes de code plus important.

Le deuxième facteur pour évaluer la complexité de notre méthodologie est le temps de simulation. Notre méthode nécessite une simulation précise pour obtenir les traces, mais les simulations suivantes peuvent être faites avec les mêmes traces. Avec PyCCExplorer, le temps de simulation dépend de la représentation de la liste des copies, du nombre de représentations que l'on simule et également du nombre d'accès mémoire. Les résultats de l'exploration architecturale des paramètres de la représentation avec une liste chaînée ont été obtenus en 12 heures de simulation sur un Intel(R) Xeon(R) CPU E3-1241 v3 @ 3.50GHz, 8 cœurs (HT) avec 32GB de RAM. La simulation des trois autres représentations de la liste des copies (espionnage, Ackwise et champ de bits complet) a été exécutée en 6 heures. Avec gem5 sans ruby, il faut 54 heures pour l'ensemble des applications, mais ce mode ne permet pas de modéliser précisément le protocole de cohérence de caches et encore moins la représentation de la liste des copies. Lorsque l'on utilise le mode ruby de gem5, le temps de simulation augmente considérablement, de l'ordre de plusieurs jours par application. Pour avoir une idée plus précise du temps

de simulation, nous nous sommes intéressés au nombre de transactions par seconde analysées pour une application : `blackscholes`. Pour cette application, gem5 traite 2 750 transactions par seconde mais seulement 290 lorsqu'on utilise le mode ruby qui est le seul à être utilisable pour l'évaluation des protocoles de cohérence de caches. PyCCExplorer permet quant à lui de traiter 110 000 transactions par secondes : il est donc 40 fois plus rapide que gem5 et 380 fois plus rapide que gem5 avec ruby. Bien évidemment, cette comparaison n'a de sens que pour l'objectif que nous nous sommes fixé : la comparaison du comportement en fonction de la représentation de la liste des copies.

6.2 Évaluation du protocole DCC avec PyCCExplorer et implémentation matérielle

Afin d'évaluer notre protocole DCC, nous utilisons notre méthodologie basée sur PyCCExplorer afin de déterminer les paramètres du protocole et de comparer DCC aux protocoles de référence. Dans un second temps, nous nous focaliserons sur l'impémentation matérielle faite en SystemVerilog.

6.2.1 Simulation avec PyCCExplorer

Nous utilisons PyCCExplorer dans le but d'évaluer notre protocole. Pour cela, nous présenterons les spécificités de notre environnement de simulation. Puis, nous nous focalisons sur l'exploration architecturale des paramètres du protocole. Par la suite, nous comparons le protocole DCC à d'autres protocoles de références déjà présentés dans les résultats du simulateur.

6.2.1.1 Environnement de simulation

Nous allons utiliser PyCCExplorer afin de fixer les paramètres de notre protocole et de comparer ses performances à celles des autres protocoles implémentés dans le simulateur. Nous avons implémenté les trois algorithmes de placement du rectangle de cohérence : **idéal**, **premier touché** et **combinatoire** selon les spécifications données dans le chapitre 4. Afin d'obtenir des simulations proches de l'exécution réelle, notre plateforme de simulation doit être adaptée. En effet, quel que soit l'algorithme, le protocole DCC tire parti du placement des tâches à proximité des autres dans un réseau 2D maillé. Or, lors de la simulation avec gem5, la topologie de ce dernier est un bus, c'est-à-dire une topologie à plat. Dans ce cas, le système d'exploitation n'essaye pas d'optimiser le placement des tâches car tous les cœurs sont à égale distance les uns des autres. Afin de remédier à ce problème, nous avons choisi de calculer un placement optimisé des tâches sur un réseau 2D maillé après l'exécution sur gem5. L'espace des solutions étant trop grand à parcourir, nous avons choisi d'appliquer un algorithme de recuit simulé. Cet algorithme nous permet de trouver un placement des cœurs sur le réseau qui minimise la fonction d'énergie suivante :

$$\sum_{i=0}^{N}\sum_{j=0}^{N} d_{ij} \times l_{ij}$$

Cette fonction est la somme pour toutes les paires de cœurs de la distance d multipliée par le nombre de lignes de cache l partagées par chaque paire de cœurs. La distance entre deux cœurs est calculée à l'aide de la distance de Manhattan. Le placement des cœurs ainsi obtenu est utilisé lors de l'injection des traces dans le simulateur. Des informations supplémentaires sur la mise en place de l'algorithme de recuit simulé sont données dans l'annexe A.

6.2.1.2 Étude des paramètres du protocole

Le protocole DCC possède trois paramètres que nous allons fixer à l'aide de PyCCExplorer. Le premier paramètre est la taille du rectangle de cohérence. Le deuxième paramètre est la valeur du seuil de la liste chaînée. Et, enfin, le dernier paramètre est le nombre d'entrées du tas.

Taille du rectangle de cohérence

Nous considérons seulement les tailles de rectangle qui sont des puissances de deux. De plus, la taille du rectangle de cohérence est égale à la taille du champ de bits dans le répertoire. Ainsi, nous voulons limiter la taille de chaque entrée du répertoire qui doit contenir toutes les informations pour encoder le rectangle, c'est-à-dire l'origine du rectangle ainsi que sa forme.

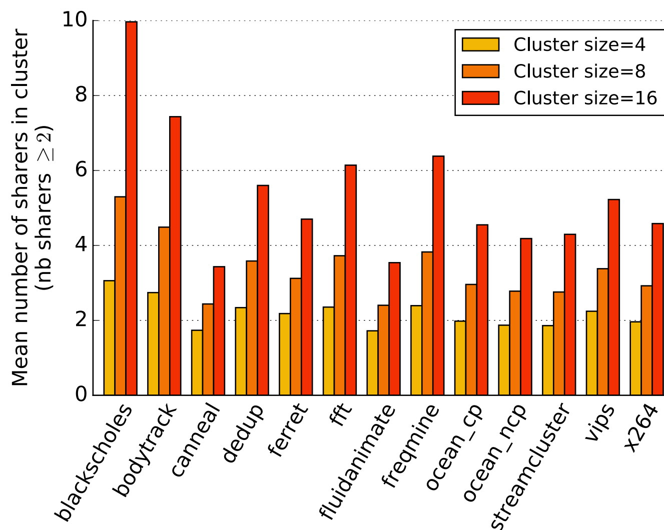


FIGURE 6.12 – Moyenne de l'utilisation du rectangle de cohérence avec le protocole DCC idéal pour les copies partagées

La figure 6.12 montre la moyenne du nombre de copies à l'intérieur du rectangle de cohérence lorsque la ligne de cache est partagée, c'est-à-dire qu'il existe au moins deux copies. Sur cette figure apparaît la moyenne de l'occupation du rectangle pour des rectangles de taille 4, 8 et 16. Pour obtenir ces résultats, nous avons appliqué l'algorithme de placement du rectangle idéal sans seuil et avec un tas illimité. Cet algorithme permet de maximiser l'utilisation du rectangle de cohérence. On remarque que lorsqu'on augmente la taille du rectangle, le nombre de copies moyen dans le rectangle augmente.

Malgré cela, on observe également que la moyenne du nombre de copies à l'intérieur du rectangle ne dépasse que rarement la moitié de la taille du rectangle. Ce résultat s'explique par le taux de partage des données très faible. Ainsi, pour une ligne de cache partagée par trois cœurs, le meilleur rectangle pourra englober au maximum trois copies dès que la taille du rectangle est supérieure ou égale à trois.

Dans le cadre de cette thèse, nous avons choisi de fixer la taille du rectangle de cohérence à 16 car cette taille nous permet de ranger un nombre important de copies. De plus, nous avons souhaité faire des expérimentations pour une taille de rectangle suffisante pour un nombre de cœurs plus important. Enfin, nous souhaitons limiter le nombre de messages envoyés en diffusion, un rectangle d'aire égale à 16 nous permet de ranger plus de copies dans le rectangle ce qui évite ou retarde le passage en mode diffusion.

Seuil de la liste chaînée

Après avoir fixé la taille du rectangle de cohérence, nous nous intéressons à la valeur du seuil de la liste chaînée. Nous avons vu les avantages d'une limite sur la taille de cette liste dans le chapitre 4 section 4.2.2. De plus, pour l'algorithme **combinatoire**, le nombre d'entrée du bloc *tiling* est contraint par le seuil de la liste chaînée. Nous avons vu que le coût matériel de ce bloc augmente vite lorsque le nombre d'entrées augmente.

Afin de fixer la valeur de ce paramètre, nous avons effectué des simulations avec l'algorithme DCC **idéal** et un tas illimité. La figure 6.13 montre le pourcentage des lignes de cache en mode diffusion pour un seuil compris entre 2 et 6. Lorsque le seuil augmente, le nombre de lignes en mode diffusion diminue mais cette diminution est plus importante entre un seuil de 2 et un seuil de 3 qu'entre les autres valeurs de seuils. Avec un seuil de 4, le bloc *tiling* doit avoir au minimum 6 entrées (un pour le rectangle, un pour le demandeur et 4 pour chaque élément de la liste chaînée). Lorsqu'on augmente la valeur du seuil, le coût matériel du bloc *tiling* nécessaire au fonctionnement de l'algorithme **combinatoire** devient trop important.

Dans la suite des simulations que nous présentons, nous avons fixé le seuil à 4 copies dans la liste chaînée.

Taille du tas

Enfin, le dernier paramètre du protocole DCC à fixer est le nombre d'entrées du tas. Ce paramètre est directement corrélé à la valeur du seuil car plus la liste chaînée est grande, plus le nombre d'entrées dans le tas doit être élevé. Nous venons de fixer le seuil de la liste chaînée à 4, nous allons donc déterminer le nombre d'entrées du tas pour cette valeur de seuil.

Afin de déterminer le nombre d'entrées nécessaires dans le tas, nous avons effectué des simulations pour chacun des trois algorithmes de placement du rectangle de cohérence. Dans ces simulations, nous avons extrait le nombre d'entrées moyen utilisées par fenêtre de 10 000 cycles. La figure 6.14 représente la moyenne, le maximum, le premier et le troisième quartile du nombre d'entrées du tas utilisés pour mémoriser les listes chaînées. Pour obtenir ces résultats, nous avons effectué les simulations avec

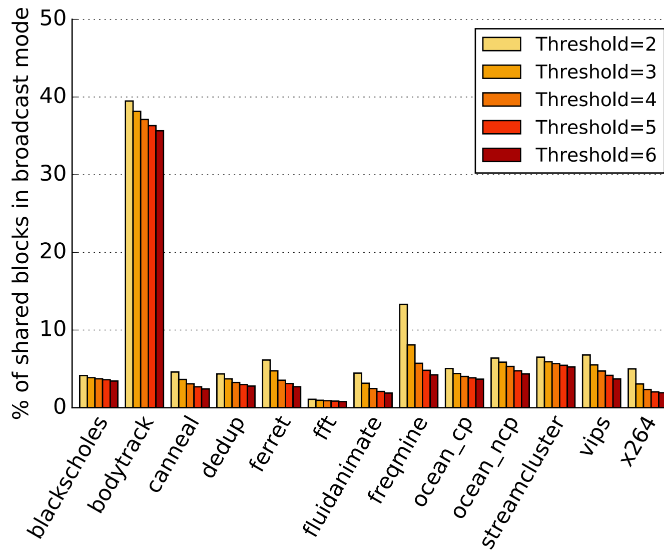


FIGURE 6.13 – Pourcentage des lignes de cache en mode diffusion avec le protocole DCC idéal et un tas illimité

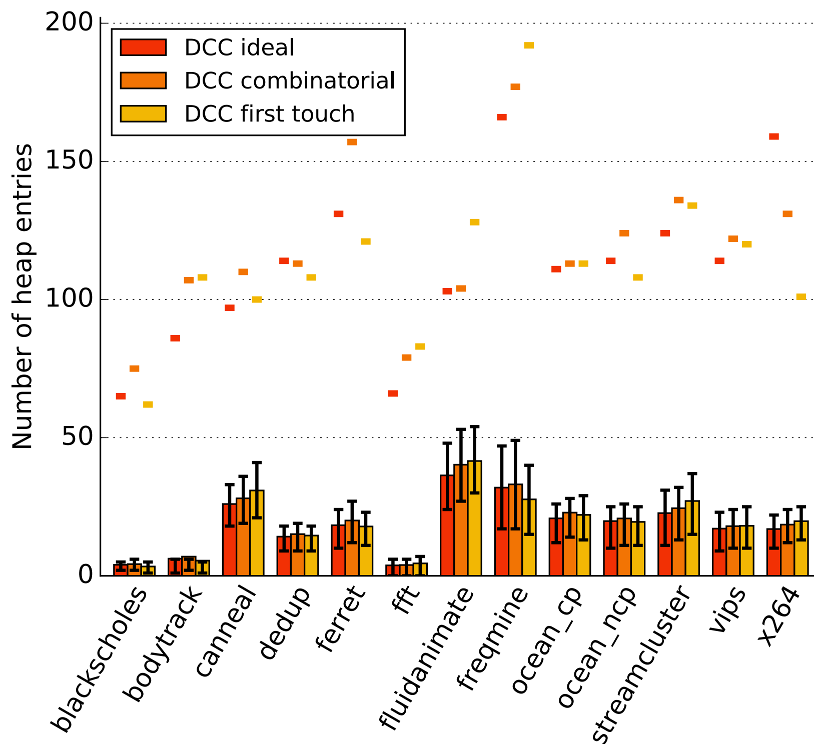


FIGURE 6.14 – Nombre d'entrées du tas utilisées pour mémoriser la liste chaînée de DCC

un nombre d'entrées illimité dans le tas. On observe des résultats assez similaires pour l'ensemble des algorithmes de placement du rectangle, cependant le type d'application influence fortement l'utilisation du tas. Néanmoins, on remarque que pour toutes les

applications et tous les algorithmes, le nombre moyen d'entrées utilisées est inférieur à 50. Le maximum dépend des applications et se situe majoritairement entre 100 et 150 entrées. Nous pouvons également observer que le premier et le troisième quartile sont proches de la moyenne, ce qui montre que la répartition du nombre d'entrées utilisées est concentrée autour de la moyenne.

À l'aide de ces observations, nous avons choisi de fixer le nombre d'entrées du tas à 128, ce qui supérieur à la moyenne du nombre d'entrées utilisées pour chaque application.

6.2.1.3 Comparaison des représentations de la liste des copies

Dans cette partie, nous nous focalisons sur les performances du protocole DCC. Pour cela nous comparons les trois variantes de DCC aux quatre représentations de la liste des copies implémentées dans PyCCExplorer : l'espionnage, le champ de bits complet, Ackwise et la liste chaînée. Pour cette comparaison, nous avons choisi de nous intéresser aux métriques suivantes : la latence, le trafic, ainsi que le nombre de messages envoyés en diffusion. Pour effectuer les simulations, nous avons utilisé les valeurs des paramètres que nous venons de déterminer : un rectangle de cohérence pouvant contenir 16 copies, une liste chaînée de taille maximale de 4 et un nombre d'entrées du tas fixé à 128.

Choix des paramètres des protocoles de référence

La méthodologie de simulation que nous avons mise en place vise à classer les représentations de la liste des copies selon plusieurs critères, il faut donc avoir des représentations de référence. Les protocoles de référence ont été présentés précédemment. Deux de ces représentations de la liste des copies possèdent des paramètres à fixer lors de l'implémentation : la valeur du seuil pour Ackwise ainsi que pour la liste chaînée, et cette dernière représentation de la liste des copies nécessite également de fixer le nombre d'entrées dans le tas.

Afin de faire une comparaison juste vis-à-vis de DCC, nous avons choisi la valeur du seuil d'Ackwise afin que le coût d'une entrée du répertoire soit comparable. Nous avons vu qu'une entrée du répertoire de DCC nécessite 33 bits plus le coût du tas (voir chapitre 4 section 4.2.4). Le coût matériel d'une entrée du répertoire d'Ackwise est de $log_2 N \times k + 1$ où N est le nombre de cœurs, soit 64 dans notre cas et k est la valeur du seuil. Avec un seuil fixé à 5, une entrée du répertoire Ackwise nécessite 31 bits, alors qu'avec un seuil fixé à 6 cela nécessite 37 bits. Nous avons choisi de comparer DCC à Ackwise avec un seuil fixé à 5 car nous avons vu qu'au-delà de ce seuil, les performances d'Ackwise augmentent moins vite.

Pour la représentation de la liste des copies avec une liste chaînée, l'entrée du répertoire permet de mémoriser un CID dont la taille est de $log_2 N$ bits, les autres copies étant rangées dans le tas. Afin d'avoir un temps d'accès comparable à celui de DCC, nous avons choisi de fixer la taille de la liste chaînée à 5, ce qui correspond à une copie dans le répertoire et quatre copies dans le tas. Le coût matériel de cette représentation est de $log_2 N + log_2 H$ où H est le nombre d'entrées dans le tas. Afin d'être comparable à DCC, nous avons choisi la même taille de tas, soit 128 entrées. Avec ce nombre d'entrées

dans le tas, le coût matériel du répertoire utilisant une liste chaînée est de 13 bits, ce qui est inférieur à celui de DCC.

Latence

La première métrique à laquelle nous nous intéressons pour comparer les représentations de la liste des copies est la latence. La figure 6.15 montre la latence moyenne pour chaque application et les différentes représentations de la liste des copies. Sur cette figure, nous retrouvons les résultats obtenus précédemment pour les représentations de la liste des copies autres que DCC. On remarque que la latence induite par les différentes variantes de DCC est 5% supérieure à celle de la représentation de la liste des copies avec un champ de bits complet. Cette représentation de la liste des copies est celle qui permet de minimiser la latence, c'est donc de cette représentation que l'on cherche à se rapprocher. La latence de DCC est influencée par le choix du *keeper* qui est utilisé lorsque l'on passe dans le mode diffusion. Or, dans notre implémentation, le *keeper* correspond à la première copie lorsque l'on passe dans le mode diffusion. Une autre solution aurait été de choisir parmi toutes les copies celle qui se trouve le plus au centre de la grille afin de minimiser la latence dans le cas où la donnée est partagée par tous les cœurs. Sur cette figure, on observe également que pour la latence, l'algorithme de placement du rectangle de cohérence influence peu les résultats.

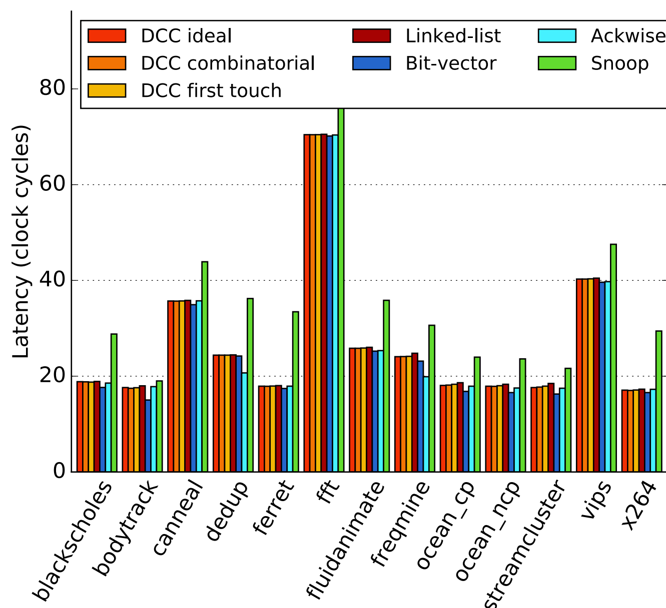
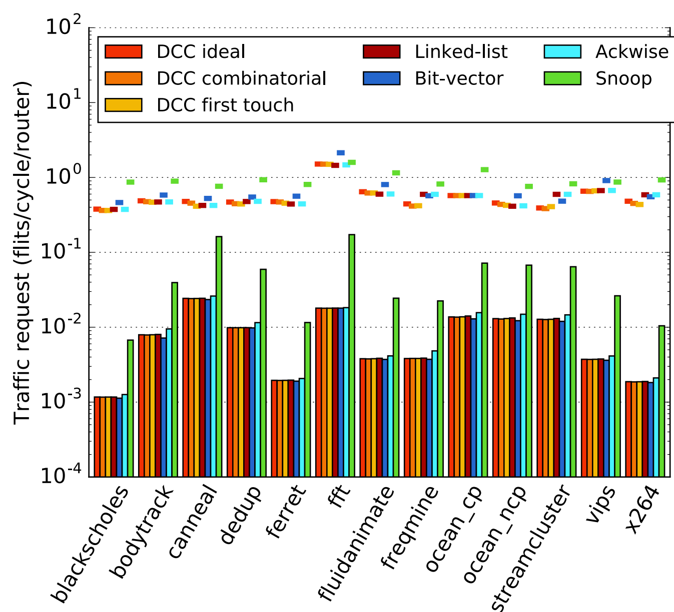


Figure 6.15 – Moyenne de la latence en nombre de cycles

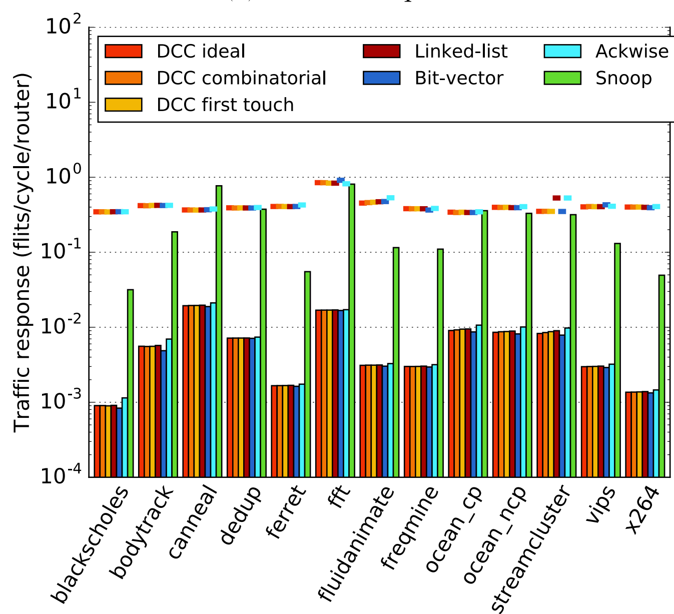
À l'aide de ces résultats, nous pouvons en déduire que les performances de DCC pour la latence sont proches de la liste chaînée qui est 6% supérieure à celle du champ de bits complet. En revanche, Ackwise permet d'obtenir une latence supérieure de 2% par rapport au champ de bits complet, ce qui est meilleur que DCC.

Trafic

La deuxième métrique à laquelle nous nous intéressons est le trafic généré par les messages de cohérence. Pour cela, nous avons séparé le trafic généré sur chacun des deux canaux : celui pour les requêtes et celui pour les réponses. Nous avons fait ce choix car les deux canaux ne sont pas symétriques. En effet, pour les représentations utilisant des messages de diffusion, un seul message est envoyé et plusieurs réponses sont attendues.

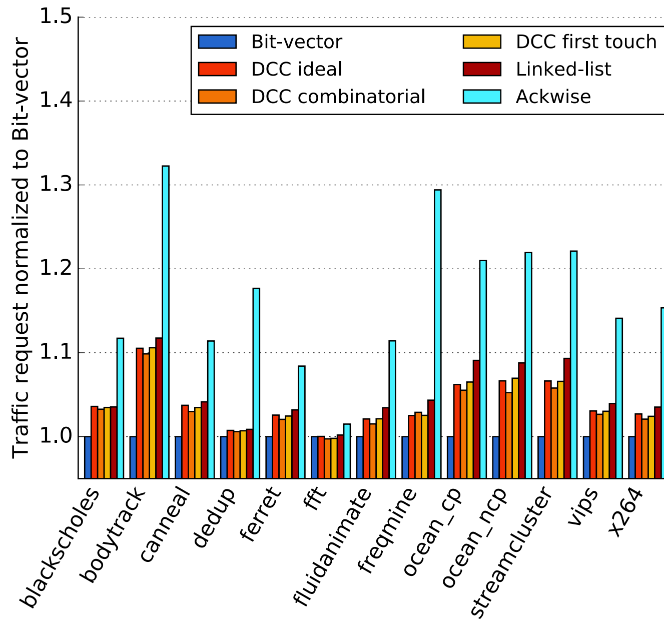


(a) Canal de requête

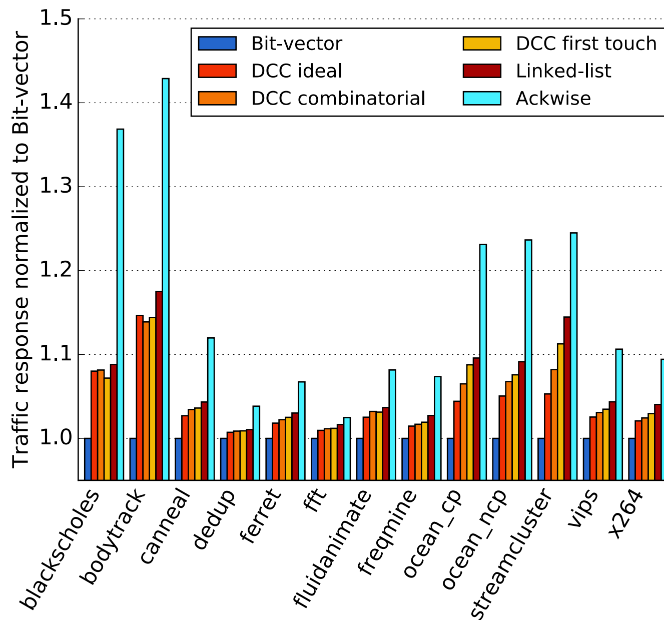


(b) Canal de réponse

FIGURE 6.16 – Moyenne et maximum du trafic



(a) Canal de requête



(b) Canal de réponse

Figure 6.17 – Moyenne du trafic normalisé par rapport au champ de bits

La figure 6.16a représente le trafic moyen ainsi que le maximum de messages générés sur l'ensemble des fenêtres de 10 000 cycles pour le canal de requête et la figure 6.16b montre les mêmes données pour le canal des réponses. Les figures 6.17a et 6.17b montrent le trafic sur chacun des canaux, normalisé par rapport à la représentation de la liste des copies utilisant un champ de bits complet. Sur ces figures, on remarque que les trois algorithmes de placement du rectangle de cohérence influencent légèrement le

trafic généré. On observe un nombre de messages envoyés sur le réseau supérieur de 4% sur chacun des deux canaux pour l'algorithme **idéal** par rapport à la représentation avec un champ de bits complet. Pour l'algorithme **premier touché**, le trafic n'est pas tout à fait symétrique sur les deux canaux, mais le trafic reste 4% et 5% supérieur à celui généré par la représentation la plus précise. Enfin, l'algorithme **combinatoire** confirme cette asymétrie entre les deux canaux : 3% de plus pour le canal de requête et 5% de plus pour le canal des réponses, toujours par rapport à la représentation avec un champ de bits complet. Cette asymétrie entre les deux canaux est engendrée par les messages envoyés en diffusion qui génèrent un nombre de réponses plus important.

Le classement des représentations de la liste des copies sur le critère du trafic place DCC entre la représentation avec une liste chaînée et la représentation précise avec un champ de bits complet. Notre solution est meilleure qu'Ackwise qui produit un trafic supérieur à celui du champ de bits complet de 17% et 16% selon le canal. Notre protocole DCC permet donc de réduire le nombre de messages générés sur le réseau par rapport à Ackwise. En revanche, DCC génére un trafic proche de celui de la représentation basée sur une liste chaînée. Pour le canal des requêtes, la figure 6.17a permet d'obtenir le classement suivant pour les variantes de DCC et de la liste chaînée, du trafic le plus faible au plus élevé : DCC **combinatoire**, DCC **idéal**, DCC **premier touché** et la liste chaînée. Une explication possible au classement de la version **combinatoire** avant la version **idéal** est que pour les données très partagées, les lignes peuvent passer plus rapidement en mode diffusion avec l'algorithme **combinatoire** et ainsi les messages seront envoyés en diffusion, ce qui génère un nombre plus faible de messages grâce au support matériel. En revanche, pour le canal de réponse, l'algorithme **idéal** permet bien d'obtenir les meilleurs performances.

Nombre de messages envoyés en diffusion

Nous venons de voir que DCC permet de réduire le nombre de messages générés sur le réseau. Afin d'aller plus loin dans l'évaluation des messages générés, nous nous sommes focalisés sur les messages envoyés en diffusion. En effet, les messages de diffusion sont coûteux et peuvent saturer le réseau.

La figure 6.18 montre le nombre de messages envoyés en diffusion en échelle logarithmique pour trois représentations de la liste des copies dont le comportement est très différent : Ackwise, espionnage et DCC **idéal**. Sur cette figure on observe que le protocole basé sur l'espionnage génère un nombre de messages envoyés en diffusion en moyenne trois ordres de grandeur de plus que DCC **idéal**. On peut également observer une différence entre DCC **idéal** et Ackwise d'en moyenne un ordre de grandeur. Ainsi, pour un coût matériel proche, DCC permet de diminuer le nombre de messages envoyés en diffusion par rapport à Ackwise.

Afin de classer les différents algorithmes de placement du rectangle de cohérence, nous avons normalisé le nombre de messages de diffusion par rapport à DCC **idéal**, lui-même comparé aux autres représentations dans la figure 6.18. En effet, les différences entre ces algorithmes est moins marquée et nous avons donc dessiné une figure focalisée sur cette comparaison en échelle linéaire. La figure 6.19 compare donc les trois versions de DCC et la représentation basée sur une liste chaînée. Sur cette figure, on observe que

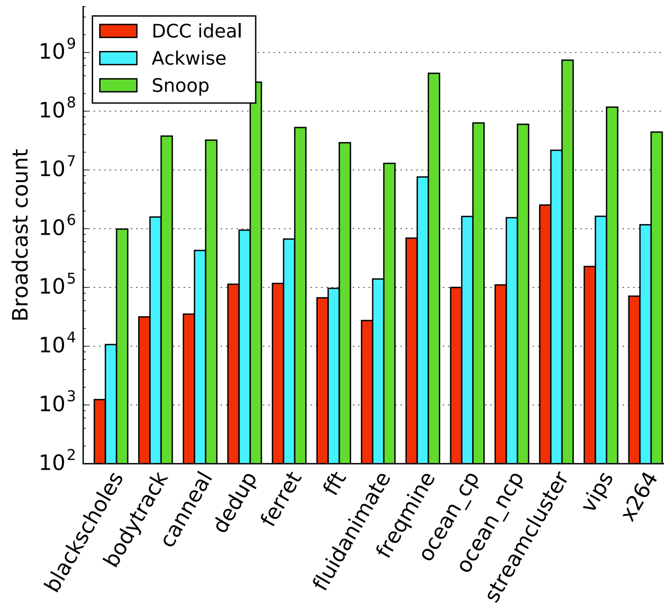


FIGURE 6.18 – Nombre de messages envoyés en diffusion

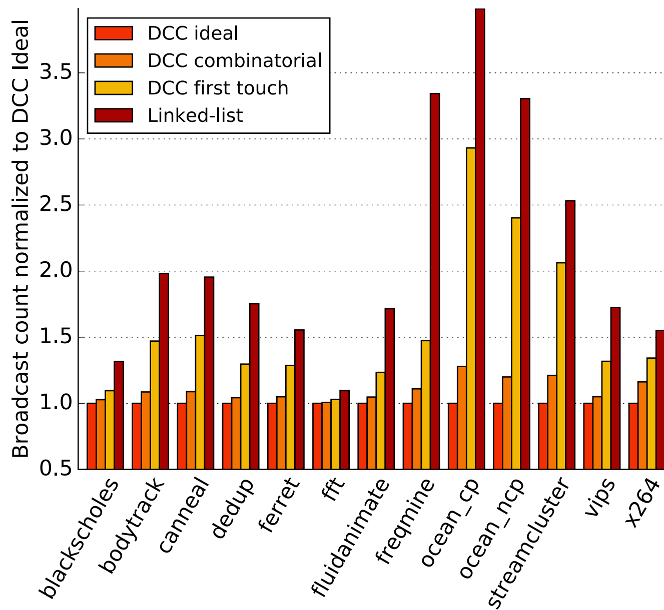


FIGURE 6.19 – Nombre de messages envoyés en diffusion normalisé par rapport à DCC **idéal**

l'algorithme **combinatoire** est très proche de l'algorithme idéal. On note également que les performances sont nettement dégradées avec l'algorithme **premier touché**. Enfin, on observe que la représentation de la liste des copies avec une liste chaînée génère un nombre bien plus important de messages de diffusion, avec en moyenne deux fois plus de messages que DCC **idéal**.

Ainsi, lorsque l'on s'intéresse au nombre de messages envoyés en diffusion, le protocole DCC permet d'obtenir de meilleurs résultats que la représentation basée sur l'espionnage ainsi que sur Ackwise dont les performances concernant les autres métriques (latence et trafic) sont proches de DCC. Nous avons également observé que l'algorithme **combinatoire** génère un nombre de messages de diffusion proche de la version **idéal**, ce qui confirme que cette variante est une bonne approximation de la version **idéal**.

6.2.1.4 Conclusion sur l'ensemble des métriques

Le tableau 6.1 résume l'ensemble des comparaisons que nous avons effectuées entre les différentes représentations de la liste des copies pour la latence, le trafic ainsi que le nombre de messages de diffusion. Ces résultats sont normalisés par rapport à la représentation utilisant un champ de bits complet, sauf pour le nombre de diffusion où les résultats sont normalisés par rapport à DCC **idéal**. Si l'on fait abstraction de la représentation la plus précise, on remarque que DCC **combinatoire** obtient de bons résultats en particulier sur le nombre de messages de diffusion.

Table 6.1 – Comparaisons des représentations de la liste des copies

Protocole	**Latence**	**Trafic requête**	**Trafic réponse**	**Nombre de diffusion**
Espionnage	+42,7%	× 6,32	× 40,88	× 789
Champ de bits	+0% (Réf.)	+0% (Réf.)	+0% (Réf.)	Aucune diffusion
Ackwise	**+1,9%**	+16,8%	+16,2%	× 12,67
Liste chaînée	+6,4%	+5%	+6,5%	+114%
DCC idéal	+4,8%	+3,9%	**+4%**	+0% (Réf.)
DCC 1er touché	+5,2%	+3,9%	+5,3%	+57,4%
DCC comb	+3,4%	**+3,4%**	+4,7%	**+10,4%**

6.2.2 Implémentation matérielle du protocole DCC

Afin de valider la faisabilité de l'algorithme **combinatoire**, nous avons implémenté en SystemVerilog le bloc central de cet algorithme : le bloc *tiling* qui place le rectangle de cohérence. Le bloc *tiling* est un bloc paramétrable : le nombre d'entrées peut varier ainsi que le choix entre l'usage des multiplications ou des LUTs lors de l'élagage. Nous avons développé un module Python chargé de générer le modèle SystemVerilog en fonction du nombre d'entrées. En particulier, ce module Python génère les combinaisons de $\binom{n}{k}$ sous-blocs. Notre implémentation a été testée à l'aide d'un *test bench* dont les assertions sont générées par notre simulateur PyCCExplorer. Ainsi, nous avons vérifié que l'implémentation matérielle donne les mêmes résultats que l'implémentation dans le simulateur.

6.2.2.1 Paramètres de synthèse

Nous avons choisi de réaliser la synthèse du bloc *tiling* pour des coordonnées d'entrée de 5 bits ce qui correspond à une architecture de 1024 cœurs. Ce choix a été fait pour obtenir le coût matériel du bloc *tiling* pour une architecture *manycore* plus importante.

Cette synthèse a été réalisée avec l'outil Design Compiler de Synopsys. Nous avons choisi la technologie 22nm FDSOI 22FDX de GlobalFoundries au *corner* lent de 0.72V, à 125°C et sans polarisation de substrat. Le bloc *tiling* est un bloc purement combinatoire, ainsi des registres d'entrées et de sorties sont utilisés pour extraire les résultats en terme de fréquence d'utilisation. Ces registres sont inclus dans les résultats concernant l'aire du bloc, mais ils sont négligeables comparés au coût de la logique du bloc.

6.2.2.2 Fréquence de fonctionnement

Les figures 6.20 et 6.21 montrent la fréquence de fonctionnement limite pour le bloc *tiling* possédant entre 2 et 8 entrées. La figure 6.20 présente ces résultats pour le bloc *tiling* dans sa version avec des multiplications alors que la figure 6.21 représente les fréquences maximales d'utilisation pour le bloc *tiling* dans sa version avec des LUTs. Lors de nos simulations avec PyCCExplorer, nous avons fixé la taille de la liste chaînée à 4, ce qui nécessite au minimum 6 entrées pour le bloc *tiling*. Ainsi, on observe que la variante avec les LUTs donne un meilleur résultat avec une fréquence maximale de 1,33 GHz avec 6 entrées, contre 1,21 GHz pour la version avec les multiplications. On remarque également sur ces figures que pour un nombre d'entrées fixé, la version avec les LUTs permet d'obtenir des fréquences de fonctionnement maximales plus élevées.

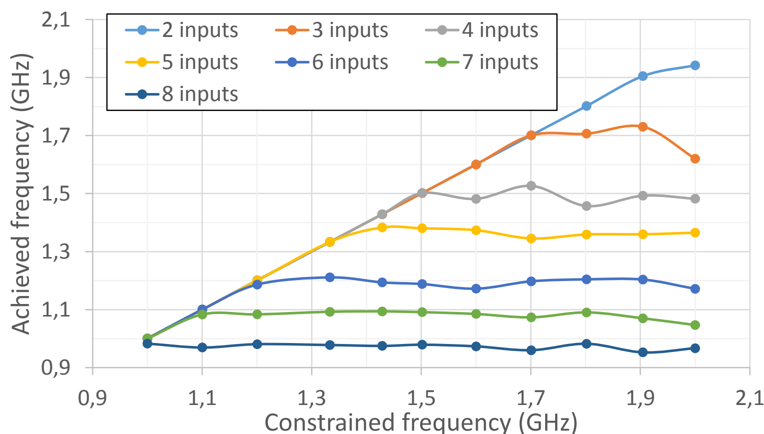


Figure 6.20 – Fréquence de fonctionnement du bloc *tiling* avec des multiplications

Nous avons comparé la fréquence de maximale du bloc *tiling* à celle d'un cache L2 de 256 ko synthétisé avec la même technologie. La fréquence de fonctionnement de ce cache est de 1,25 GHz, ce qui est compatible avec les résultats obtenus par le bloc *tiling* avec des LUTs et 6 entrées.

6.2.2.3 Surface du bloc *tiling*

Un deuxième paramètre important pour valider la faisabilité du bloc *tiling* est l'aire de ce bloc. En effet, nous voulons que la surface occupée par le matériel dédié à la cohérence des caches soit faible comparée à la surface nécessaire aux données.

La figure 6.22 montre l'aire du bloc *tiling* à la fréquence de fonctionnement maximale pour un nombre d'entrées compris entre 2 et 8. Sur cette figure, on observe les résultats

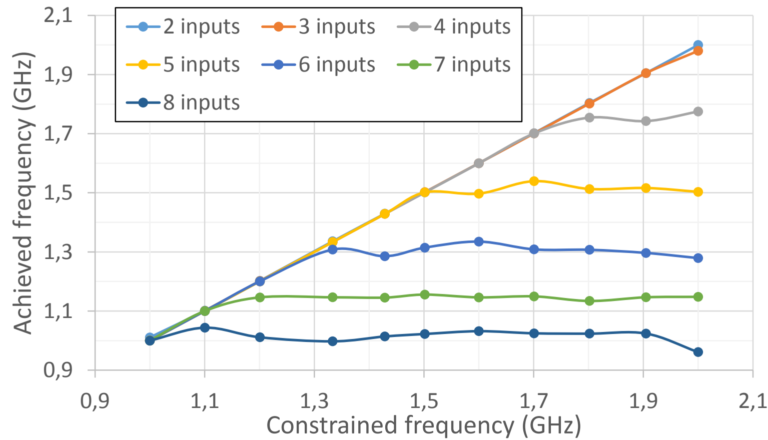


FIGURE 6.21 – Fréquence de fonctionnement du bloc *tiling* avec des LUTs

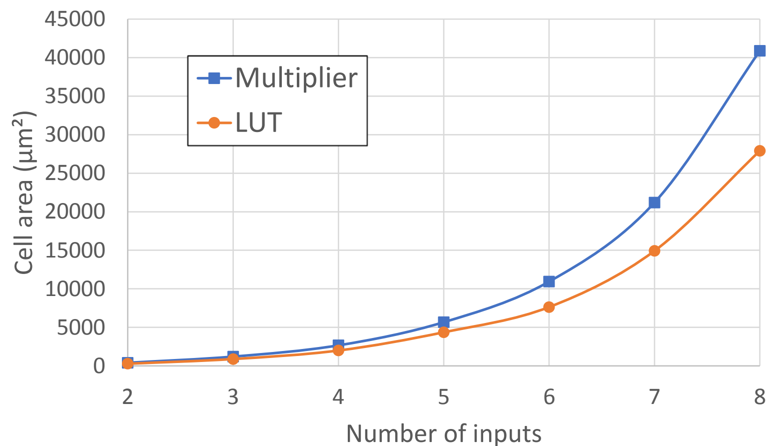


FIGURE 6.22 – Surface du bloc *tiling* à la fréquence maximale de fonctionnement en fonction du nombre d'entrées

pour les deux variantes : avec des multiplications et avec des LUTs. On remarque que la surface nécessaire à l'implémentation avec des LUTs est toujours plus petite que pour l'implémentation avec des multiplications. Pour l'implémentation correspondant à nos simulations, soit avec 6 entrées, l'aire est de 7 635 μm^2 pour la version avec des LUTs et de 10 938 μm^2 pour la version avec des multiplications. La surface occupée par les données d'un cache L2 de 256 ko est de 450 000 μm^2 environ dans la même technologie. Ainsi, le bloc *tiling* représente moins de 2% de la surface totale d'un cache L2 de cette taille.

6.3 Conclusion

Dans ce chapitre, nous avons présenté les résultats des expérimentations réalisées dans le but d'évaluer les représentation de la liste des copies. Nous avons utilisé PyCCExplorer afin de réaliser l'exploration architecturale de deux représentations de la liste des copies : Ackwise et la représentation basée sur une liste chaînée. Nous avons ainsi pu observer l'influence du seuil d'Ackwise sur la latence. De plus, nous avons pu déter-

miner le nombre d'entrées dans le tas pour le protocole avec une liste chaînée et nous nous sommes également intéressés à l'influence du seuil pour cette liste chaînée. Nous avons également utilisé cet outil afin de classer quatre représentations qui nous servent de référence pour notre méthodologie. Nous avons observé que le protocole basé sur l'espionnage a une latence supérieure aux autres et qu'il génère également un nombre beaucoup plus important de messages. Nous avons remarqué que la représentation de la liste des copies la plus précise, c'est-à-dire avec un champ de bits complet, permet d'obtenir la latence la plus faible et un nombre de messages générés moins important. Enfin, nous avons remarqué qu'Ackwise permet d'obtenir une latence proche de celle de la représentation avec un champ de bits complet alors que la représentation avec une liste chaînée se rapproche des performances de cette représentation sur le critère du trafic généré sur le réseau.

Nous avons également validé notre méthodologie en comparant le comportement de notre cache L2 et celui de gem5. Nous avons observé des différences inférieures à 3%, démontrant que nous n'avons pas divergé de la modélisation du cache L2 de gem5. Nous avons comparé nos résultats à ceux obtenus avec des simulateurs précis et notre méthodologie permet d'obtenir le même classement que ces simulateurs. Nous nous sommes intéressés à la complexité de notre méthode de simulation en nous focalisant sur l'effort de développement ainsi que le temps de simulation. L'ajout d'une nouvelle représentation de la liste des copies dans PyCCExplorer ne nécessite que 150 lignes de code Python. Le temps de simulation de notre outil permet également d'effectuer une exploration architecturale rapidement. En effet, l'exploration architecturale de la représentation avec une liste chaînée a nécessité 12 heures et la simulation nécessaire au classement des trois autres représentations de référence a duré seulement 6 heures. Nous avons aussi montré que notre simulateur est 40 fois plus rapide que gem5 et 380 fois plus rapide que gem5 avec ruby.

Par la suite, nous avons utilisé PyCCExplorer afin d'effectuer une exploration architecturale des paramètres de DCC. Nous avons fixé la taille du rectangle de cohérence à 16 copies puis la valeur du seuil de la liste chaînée à 4 et enfin le nombre d'entrées dans le tas à 128 pour un cache L2 de 256 kB. Nous avons comparé notre protocole DCC sous ses trois variantes aux quatre protocoles de référence. Ainsi, nous avons pu évaluer l'influence de la représentation de la liste des copies sur la latence, le trafic ainsi que le nombre de messages envoyés en diffusion. Nous avons établi que DCC se classe troisième du point de vue de la latence la plus faible, derrière le champ de bits complet et Ackwise mais devant la liste chaînée et l'espionnage. Pour le trafic, DCC arrive deuxième juste derrière le champ de bits complet. Enfin, pour le nombre de messages envoyé en diffusion, notre solution se classe première des protocoles qui génèrent des messages de diffusion. Pour ce qui est de la différence entre les algorithmes de placement du rectangle de cohérence, l'algorithme **combinatoire** est toujours meilleur que l'algorithme **premier touché**. Nous avons pu constater que les performances de l'algorithme **combinatoire** sont proches de l'algorithme **idéal**, qui permet d'obtenir les meilleurs résultats.

Enfin, nous avons réalisé l'implémentation matérielle en SystemVerilog du bloc *tiling* qui place le rectangle de cohérence dans l'algorithme **combinatoire**. Nous avons pu déterminer d'une part la fréquence maximale de fonctionnement pour ce bloc en

fonction du nombre d'entrées, et d'autre part la surface nécessaire du bloc à sa fréquence maximale. Nous avons également pu observer que l'implémentation avec des LUTs permet d'obtenir de meilleurs résultats que celle avec des multiplications. Les résultats de synthèse nous ont montré que la fréquence de fonctionnement et la surface nécessaire de ce bloc avec 6 entrées sont compatibles avec ceux d'un cache L2 de 256 ko.

Chapitre 7: Travaux futurs

Dans ce chapitre, nous identifions des travaux futurs à mener pour améliorer les contributions de cette thèse. La première partie se concentre sur les fonctionnalités et modules que l'on pourrait ajouter dans PyCCExplorer. La deuxième partie se focalise sur le protocole DCC.

7.1 Améliorations à apporter à PyCCExplorer

7.1.1 Simulation d'architecture avec plus de 64 cœurs

Dans le cadre de cette thèse, nous avons utilisé un jeu de traces extrait du simulateur gem5. Actuellement, la plateforme Alpha simulée avec gem5 est limitée à 64 cœurs et il n'existe pas une autre plateforme permettant des simulations avec plus de cœurs. Afin d'effectuer des simulations d'architectures possédant un plus grand nombre de cœurs avec PyCCExplorer, le trafic provenant d'un autre simulateur que gem5 pourrait être extrait. On pourrait également modifier gem5 pour simuler plus de 64 cœurs, mais l'opération n'est pas simple, car elle dépend non seulement des modules de gem5 proprement dits, mais aussi de la taille des champs de bits dans l'*Instruction Set Architecture* des processeurs considérés. De plus, il faudra également recompiler un linux pour le nouveau nombre de cœurs compatible avec les contraintes de gem5.

7.1.2 Ajout d'autres topologies de réseau

Lors de notre étude de l'influence de la représentation de la liste des copies sur les performances, nous nous sommes focalisés sur les architectures basées sur un réseau de type 2D maillé, et seul ce type de réseau est implanté dans PyCCExplorer. Une amélioration possible de notre simulateur est l'ajout d'autres topologies de réseaux comme les bus, les hypercubes, les tores, etc.

7.1.3 Ajout d'autres représentations de la liste des copies

La littérature propose d'autres manières de représenter la liste des copies, comme par exemple *coarse bit-vector directory*. L'ajout d'un protocole de cohérence de caches avec un répertoire utilisant des informations temporelles pourrait également permettre de comparer les différentes approches ayant pour but de limiter la taille du répertoire tout en limitant la latence d'accès aux données et le trafic engendré sur le réseau.

7.1.4 Ajout de méthodes décentralisées

L'utilisation de filtres de Bloom pour filter les messages et ne les propager que dans des sous-parties du réseau a été proposée. Bien que ces méthodes fassent l'impasse sur un certain nombre de problèmes (présence de faux positifs *et* de faux négatifs), il pourrait être utile d'étudier la manière de modéliser de manière abstraite ces approches.

7.2 Limites des simulations du protocole DCC

7.2.1 Simulation des applications avec l'initialisation

Les traces extraites de gem5 correspondent aux messages de cohérence entre les caches L1 et L2 lorsque l'on est dans la ROI. Ce choix a été fait car c'est sur cette partie du code que les données sont partagées et donc que l'on peut observer l'influence de la représentation de la liste des copies. Or, dans les applications réelles, la phase d'initialisation est exécutée et les caches ne sont pas remis à 0 à la fin de cette phase. Nous pensons donc que des simulations sur l'ensemble de la durée de vie d'une application permettrait de voir les effets de l'initialisation. Ainsi, pour le protocole DCC, nous pensons que l'algorithme **premier touché** qui est assez proche des autres dans nos simulations pourrait s'en éloigner à cause de l'initialisation faite par un *thread* présent sur un cœur différent de celui qui effectue le traitement.

Néanmoins, dans la plupart des applications, le nombre d'accès mémoire lors de la phase d'initialisation est négligeable par rapport à celui dans la ROI [SR16]. De plus, dans le cas où les simulations ne prennent en compte que la ROI, les résultats correspondent à la situation optimale.

7.2.2 Simulation sans le placement optimal des cœurs

Notre protocole DCC tire partie du placement des tâches communicantes à proximités des autres. Or, lors de la simulation avec gem5, les cœurs sont à égale distance les uns des autres, ainsi le système d'exploitation ne déplace pas les tâches pour les rapprocher. Afin de remédier à ce problème, nous avons choisi d'appliquer un placement optimal des cœurs. Ce placement est effectué à base d'un algorithme de recuit simulé basé sur l'analyse, après l'exécution, des accès mémoires. Ce placement est l'une des solutions approchant le placement optimal et nous ne savons pas si le système d'exploitation cherche effectivement à approcher ce placement. Afin d'évaluer l'influence du placement des cœurs sur la grille, une simulation des différentes versions du protocole DCC sans le placement optimal des cœurs pourrait être exécutée.

Afin d'aller plus loin, une solution serait de simuler une architecture avec un réseau de type 2D maillé avec un simulateur précis et d'injecter ce trafic dans PyCCExplorer. Malheureusement, pour effectuer des simulations avec gem5 et un réseau de ce type, il faut utiliser le mode ruby. Ce mode augmente considérablement le temps des simulations et les traces ne peuvent plus être extraites avec les moniteurs intégrés dans gem5. Une autre solution serait d'utiliser un autre simulateur précis permettant de simuler une architecture *manycore* avec un réseau maillé et d'extraire le trafic entre les caches L1 et les caches L2.

7.2.3 Simulation avec d'autres architectures

Dans le cadre de cette thèse nous nous sommes focalisés sur une architecture de 64 cœurs sur une grille de taille 8×8. Il nous semblerait intéressant d'effectuer des simulations avec d'autres types d'architectures comme par exemple des architectures avec des *clusters*. Dans ce genre d'architecture, les cœurs appartenant au même *cluster* communiquent rapidement entre eux. De plus, un groupe de cœurs correspond à un nœud du réseau.

Comme nous l'avons vu précédemment, nos simulations ont été faites seulement avec 64 cœurs, des simulations avec un nombre plus important de cœurs permettraient d'évaluer plus largement le passage à l'échelle du protocole DCC.

7.3 Pistes de recherche autour du protocole DCC

7.3.1 Choix du *keeper*

Comme nous l'avons vu dans le chapitre 6 section 6.2.1.3, lorsque l'on passe dans le mode diffusion le *keeper*, c'est-à-dire le cache qui va répondre aux requêtes de cohérence, est identifié par la première copie dans le champ de bits du répertoire DCC. Nous avons également vu que les données sont soit très peu partagées, soit partagés par tous les cœurs ou presque. Dans le premier cas, notre protocole va essayer de placer les copies dans le rectangle et dans la liste chaînée alors que dans le deuxième cas, le mode diffusion est utilisé. Dans ce cas, afin de limiter la latence d'accès aux données, le *keeper* devrait être choisi parmi les caches possédant la donnée et qui sont le plus au centre du réseau.

Nous pensons également qu'il existe encore des pistes de recherche pour sélectionner le meilleur *keeper*, notamment lorsque le *keeper* évince la ligne de cache et qu'il faut choisir un nouveau *keeper*.

7.3.2 Intégration du bloc *tiling* dans une architecture

Lors des expérimentations avec le protocole DCC nous avons présenté les résultats des simulations avec PyCCExplorer, ainsi que les résultats de synthèse du bloc *tiling*. Afin d'aller plus loin, le bloc *tiling* pourrait être intégré dans une architecture. Pour cela, la logique de contrôle et la machine à états à haut niveau devront être développées.

7.3.3 Ajouter un tas partagé pour mémoriser les rectangles de cohérence

Afin de réduire le coût matériel d'une entrée du répertoire, nous proposons d'ajouter un niveau d'indirection entre le répertoire et un tas partagé utilisé pour mémoriser les informations sur le rectangle de cohérence. En effet, plus de la moitié des accès mémoire sont privés à un seul cœur et ce cas ne nécessite pas le stockage d'un rectangle de cohérence avec sa liste des copies.

Nous avons imaginé plusieurs solutions permettant d'ajouter ce niveau d'indirection en partageant plus ou moins les structures de données. Une première solution est d'ajou-

ter un tas partagé pour mémoriser les informations de cohérence au format du protocole DCC. Une deuxième solution est d'ajouter deux tas partagés, l'un pour mémoriser les lignes qui utilisent un rectangle de cohérence et l'autre pour les lignes en mode diffusion. Nous allons maintenant détailler ces deux solutions.

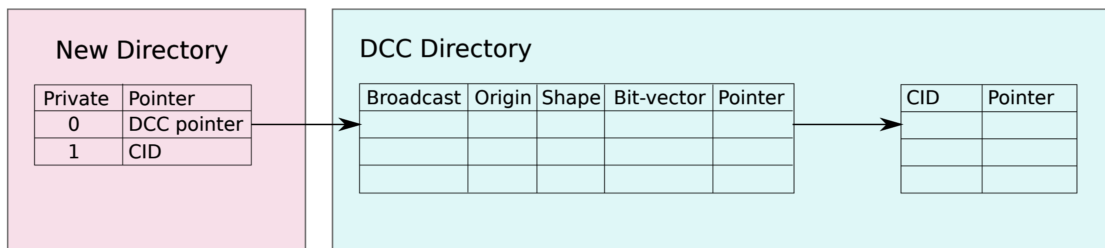


Figure 7.1 – Répertoire DCC avec ajout d'un niveau d'indirection selon si la ligne est privée ou partagée

Pour la première solution, la figure 7.1 montre le nouveau répertoire et les structures du répertoire DCC (les mêmes que celles présentées précédemment). Le premier champ de ce nouveau répertoire indique si la ligne est partagée ou non. Lorsque la ligne est privée, le deuxième champ permet de mémoriser le CID de la copie. Lorsque la ligne est partagée, le deuxième champ stocke un pointeur vers un tas partagé qui contient la liste des copies au format décrit dans le protocole DCC. Dans cette solution, il faut donc dimensionner la taille du tas nécessaire pour mémoriser les rectangles de cohérence. Pour cela, on peut s'appuyer sur le nombre de lignes partagées ainsi que le nombre d'entrées du répertoire. Afin, d'être plus précis, cette exploration architecturale peut être réalisée à l'aide de notre outil PyCCExplorer. Une telle expérimentation n'a pas été réalisée durant cette thèse.

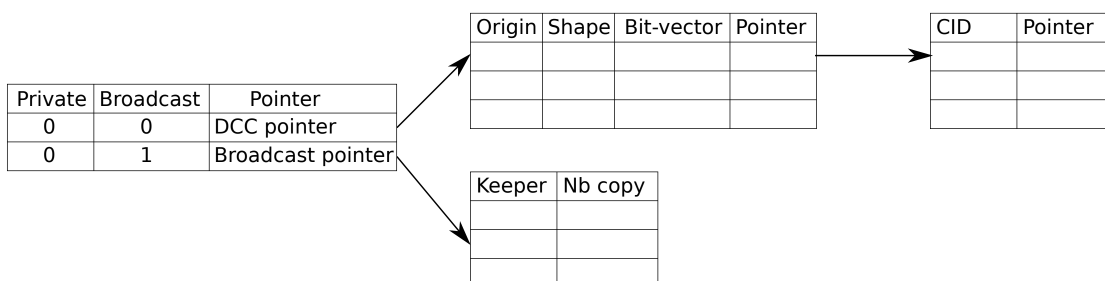


Figure 7.2 – Répertoire DCC avec ajout d'un niveau d'indirection selon le mode de la ligne

Pour la deuxième solution, l'entrée du répertoire est légèrement différente, un champ *broadcast* est ajouté afin de savoir si le pointeur pointe vers le tas partagé pour les lignes en mode diffusion ou celui stockant les informations au format du protocole DCC. La figure 7.2 montre les différentes structures nécessaires au stockage de la liste des copies. Ces différentes structures sont les suivantes :

— Le répertoire pour chaque ligne de cache ;

— Le tas utilisé pour mémoriser les informations sur le rectangle de cohérence lorsque la ligne est partagée et dans le mode précis ;
— Le tas utilisé pour ranger les listes chaînées ;
— Le tas utilisé pour ranger le *keeper* et le nombre de copies lorsque la ligne est partagée et qu'elle est dans le mode diffusion.

Les tailles des trois tas utilisés dépendent les unes des autres. Le tas pour les lignes en mode diffusion a une dépendance forte sur la valeur du seuil et la taille du tas stockant la liste chaînée. En effet, plus le seuil est haut, plus le tas utilisé par les listes chaînées est grand, et donc plus le nombre de lignes en mode diffusion est faible et ainsi le nombre d'entrées du tas pour ces lignes doit être faible. En revanche, plus le nombre de lignes en mode diffusion est élevé, moins le nombre d'entrées utilisées dans le tas par les rectangles de cohérence sera grand. Le dimensionnement de ces deux tas est plus complexe que pour la première solution et nous n'avons pas non plus réalisé cette expérimentation.

Chapitre 8: Conclusion

Dans cette thèse, nous nous sommes intéressés aux différentes représentations de la liste des copies utilisées dans les protocoles de cohérence de caches, et ce, en particulier, pour les architectures *manycores*. Nous nous sommes également intéressés à un problème connexe que rencontrent tous les architectes : la simulation d'architectures *manycores* en vue de l'exploration de l'espace de conception. En l'occurrence, il s'agit de choisir la meilleure représentation de la liste des copies selon une métrique pouvant prendre en compte la latence, le trafic ou encore les ressources matérielles.

Nous avons exposé une liste de questions dans le chapitre 2 et les travaux présentés dans le cadre de cette thèse ont eu pour but de répondre à ces questions. Les réponses à ces questions sont résumées ci-dessous.

Comment peut-on évaluer un protocole de cohérence de caches et plus particulièrement la représentation de la liste des copies ?

Nous avons vu dans le chapitre 5 qu'il existe plusieurs méthodes de simulation allant des méthodes analytiques aux simulations. Dans le cadre de cette thèse, nous nous sommes focalisés sur les techniques de simulation. Il existe de nombreux de simulateurs précis comme gem5 qui permettent la simulation d'architectures mais nous avons vu que ces simulateurs n'étaient pas adaptés à la simulation d'architectures *manycores*. En effet, les temps de simulation augmentent rapidement avec le nombre de cœurs simulés et plus le niveau de détail est important, plus le temps de simulation augmente. Lorsque l'on souhaite effectuer une exploration architecturale, il faut généralement de nombreuses simulations afin de déterminer les bons paramètres. Ainsi, la durée d'une simulation est critique.

Dans le cadre de cette thèse, nous avons proposé une méthode de simulation rapide en trois étapes basée sur l'injection de traces dans un modèle de cache à haut niveau d'abstraction. Cette méthode permet de classer par ordre relatif de performance différentes représentations de la liste des copies. La première étape consiste à extraire les messages de cohérence entre les caches provenant d'une simulation précise, dans notre cas les traces proviennent de gem5. La deuxième étape repose sur une modélisation à haut niveau d'abstraction de la hiérarchie mémoire, qui est l'une des contributions majeures de cette thèse. La modélisation décrit le cache L2, son répertoire avec la représentation de la liste des copies et le réseau. L'outil PyCCExplorer permet de décrire concrètement les modèles et d'injecter le trafic pour les simuler. Nous avons implanté

quatre représentations de la liste des copies qui servent de références pour évaluer les performances de nouvelles représentations de la liste des copies. PyCCExplorer est instrumenté afin d'extraire un certain nombre de métriques comme la latence, le trafic ou encore le taux de partage des lignes de cache. Enfin, la dernière étape consiste à analyser les résultats obtenus.

Notre méthodologie permet de classer les représentations de la liste des copies de façon relativement rapide : 6 heures de simulation pour le classement des protocoles espionnage, champ de bits complet et Ackwise, et 12 heures de simulation pour l'exploration architecturale du protocole avec une liste chaînée. Cette méthode est près de 400 fois plus rapide que gem5 avec le mode ruby qui permet la modélisation des protocoles de cohérence de caches. De plus, l'effort de développement pour l'ajout d'une nouvelle représentation de la liste des copies est faible : environ 150 lignes de Python.

Quelles sont les représentations de la liste des copies qui permettent de réduire la taille du répertoire des protocoles de cohérence de caches ?

Dans le chapitre 3 nous avons vu qu'il existait deux classes de protocoles de cohérence de caches : ceux basés sur l'espionnage et ceux utilisant un répertoire. Nous avons vu que les stratégies utilisant un répertoire limité comme Ackwise ou encore *coarse bit-vector directory* permettent de réduire la taille de chaque entrée du répertoire. Lorsque le nombre de copies dans l'entrée du répertoire atteint la limite, il faut passer dans un mode dégradé. Dans le cas d'Ackwise, le mode dégradé ne mémorise plus la liste des copies mais seulement un compteur de copies et les messages sont envoyés en diffusion. Les performances de ces protocoles dépendent du choix du seuil avant de passer dans ce mode dégradé.

Nous nous sommes également intéressés aux répertoires dynamiques qui utilisent des ressources communes afin de mémoriser la liste des copies. Nous avons évalué le protocole utilisant une liste chaînée et nous avons pu en voir les limites : le tas partagé par toutes les lignes peut être pollué par des lignes très partagées lorsqu'il n'y a pas de seuil pour passer en mode dégradé. Lorsque ce mode existe, il est important de fixer le seuil correctement, comme pour les protocoles avec un répertoire limité.

Même si les représentations de la liste de copies limitées semblent être une solution pour diminuer la taille des entrées du répertoire, cela engendre un nombre important de messages envoyés en mode diffusion. Afin de remédier à ce problème, nous avons présenté le protocole DCC qui permet de limiter l'usage du mode dégradé ainsi que la taille de chaque entrée du répertoire. Cette représentation mémorise la liste des copies d'un rectangle sous la forme d'un champ de bits et utilisent une liste chaînée de taille limitée pour les copies qui ne sont pas dans le rectangle. La liste chaînée est rangée dans un tas partagé, comme le protocole avec une liste chaînée. Lorsque la liste chaînée est pleine ou que le tas n'a plus d'entrées disponibles, nous utilisons un mode dégradé et les messages sont envoyés en diffusion. Nous avons montré que notre représentation de la liste des copies permet d'obtenir une latence 2% plus élevée que celle de la représentation la plus précise. Pour le trafic généré, il est plus élevé de 3% pour le canal des requêtes et 5% pour le canal des réponses. Enfin, le nombre de

messages envoyés en diffusion est moins élevé qu'avec les autres représentations de la liste des copies.

On observe une différence d'un ordre de grandeur avec Ackwise, et de deux ordres de grandeur avec le protocole basé sur l'espionnage. Par rapport à la liste chaînée, nous avons remarqué une diminution par 2 du nombre de messages envoyés en diffusion. Ainsi DCC permet un compromis entre la taille de chaque entrée dans le répertoire et les performances temporelles.

Peut-on exploiter les caractéristiques dynamiques des applications pour trouver une représentation plus compacte de la liste des copies ?

Lors de notre état de l'art, nous n'avons pas trouvé de représentation de la liste des copies qui permette d'exploiter les caractéristiques dynamiques des applications. Dans le cadre de cette thèse, nous avons proposé une nouvelle représentation de la liste des copies qui permet de mémoriser un sous-ensemble des copies présentes dans un rectangle dont la position et la forme évoluent au cours de l'exécution des applications. En effet, le système d'exploitation place les tâches communicantes à proximité et nous avons observé que 93,4% des lignes de cache sont partagées par au maximum 8 cœurs. Le rectangle de cohérence est propre à chaque ligne de cache et permet de mémoriser la majorité des cœurs accédant à cette ligne. Notre représentation permet également de mémorisée les copies qui ne sont pas dans le rectangle dans une liste chaînée. Cette liste chaînée ajoute également de la souplesse : lors des premiers accès, il est difficile de savoir où se situe la zone des cœurs partageant la ligne de cache. Dans ce cas, toutes les copies sont mémorisées et lors des accès suivants l'algorithme de placement du rectangle décidera du rectangle permettant de ranger le plus de copies.

Dans cette thèse, nous avons proposé trois algorithmes de placement du rectangle de cohérence : **idéal**, **premier touché** et **combinatoire**. Nous avons présenté les résultats de l'algorithme **idéal** qui correspond à la borne maximale des performances atteignables avec le protocole DCC sans tenir compte de l'implantation. Nous avons également présenté deux algorithmes qui peuvent être implantés concrètement avec un coût matériel raisonnable. Nous nous sommes focalisés sur l'algorithme **combinatoire** qui donne théoriquement de meilleures performances, et nous en avons proposé une implantation matérielle. Nous avons observé que cet algorithme permet d'approcher les performances de l'algorithme **idéal** tout en n'occupant que 2% de la surface d'un banc de cache L2. Nous avons aussi montré que la fréquence de fonctionnement de notre bloc est compatible avec la fréquence de fonctionnement d'un cache L2.

Annexes

Annexe A: Algorithme de placement des tâches

À cause du temps de simulation bien trop importante avec la modélisation précise (nommée *ruby*) de la hiérarchie mémoire avec gem5, nos simulations sont faites avec le modèle mémoire classique. Dans cette configuration, les cœurs sont reliés avec un réseau de type bus. Il s'agit dont d'une topologie à plat. Ainsi, vu d'un processeur donné, tous les caches sont à égale distance. Or, lorsque le réseau est de type 2D maillé, les caches sont à des distances différentes selon les nœuds auxquels ils sont rattachés. Dans ce cas, un objectif important du système d'exploitation est de rapprocher, topologiquement, les tâches communicantes. Pour mémoire, dans nos expérimentations, une tâche est assignée à un unique cœur et un cœur exécute une unique tâche. Afin d'obtenir un comportement proche de ce qu'il se passerait avec un système d'exploitation moderne tournant sur une architecture de type grille, nous avons choisi de placer les tâches communicantes à proximité les unes des autres à l'aide d'une heuristique. Ce placement est fait une fois pour toutes de manière statique et il est mémorisé afin que toutes nos simulations avec PyCCExplorer soient faites avec le même placement.

La recherche d'une solution optimale étant à la fois inaccessible, à cause de la taille de l'espace des solutions, et invraisemblable, car jamais un système d'exploitation ne garantira une telle solution, nous avons choisi d'appliquer à nos tâches un classique recuit simulé [KGV+83] dont, pour mémoire, le principe est rappelé dans l'algorithme 2.

A.1 Présentation de l'algorithme de recuit simulé

C'est un algorithme itératif, qui, partant d'une solution initiale choisie selon des critères dépendant du problème, vise à minimiser une fonction d'énergie, dans notre cas $\sum_{i=0}^{N}\sum_{j=0}^{N} d_{ij} \times l_{ij}$, en faisant décroître la température selon une loi de décroissance arbitraire. Pour chaque couple (i, j), correspondant à deux nœuds de la grille, l_{ij} correspond au nombre de lignes partagées entre les caches appartenant à chacun des nœuds. Dans notre cas, il y a un seul cache par nœud. Dans cette même formule, d_{ij} correspond à la distance de Manhattan entre les deux nœuds sur la grille.

À chaque itération, une modification est appliquée au système. Dans notre cas nous échangeons la position de deux cœurs sur la grille. L'énergie est alors calculée et comparée à l'énergie précédente afin d'obtenir la variation Δ_E entre les deux solutions. Si

Algorithme 2: Algorithme de recuit-simulé

```
   Données en entrée : S0 : la configuration initiale ;
   T0 : la température initiale ;
   Nb_Iter_Temp : le nombre de diminutions de la température ;
   Nb_Iter : le nombre d'itérations pour chaque température
   Résultat : S_min : une configuration proche de la solution optimale
 1 S ← S0;
 2 T ← T0;
 3 e_min ← cout(S);
 4 for i = 0 to Nb_Iter_Temp do
 5     for j = 0 to Nb_Iter do
 6         S' ← voisin de S;
 7         e ← cout(S');
 8         Δ_E ← e − e_min;
 9         if Δ_E < 0 then            /* S' est une configuration meilleure que S */
10             e_min ← e;
11             S ← S';
12         else
13             p ← random();
14             if p < e^(−Δ_E/T) then   /* On accepte S' avec une probabilité e^(−Δ_E/T) */
15                 e_min ← e;
16                 S ← S';
       /* Décroissance de la température */
17     T ← T × 0.9;
```

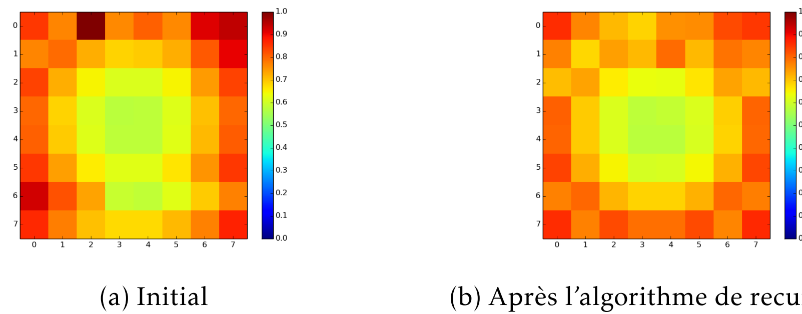
la variation est négative, alors la nouvelle solution permet de diminuer l'énergie et ainsi cette solution est retenue comme étant la meilleure de toutes celles déjà parcourues. Dans les autres cas, la solution est acceptée avec la probabilité $e^{-\frac{\Delta_E}{T}}$ où T est la température. La température doit décroître tout au long de l'exécution. Dans notre cas, la température décroît par palier : toutes les 400 inversions de cœurs sur la grille. L'algorithme s'arrête lorsque le système a atteint une température basse.

Nous avons choisi d'appliquer une loi de décroissance simple : $T_{i+1} = 0.9 \times T$. Dans notre application, la température initiale est fixée à 60 000 et nous faisons baisser la température 100 fois pour atteindre 1,59. Les paramètres de ce protocole sont souvent choisis de manière empirique. Nous avons essayé plusieurs températures initiales ainsi qu'un nombre variable d'itération, où chaque itérations se termine par une baisse de la température. Nous avons également essayé plusieurs lois de décroissance simple entre $T_{i+1} = 0.8 \times T$ et $T_{i+1} = 0.99 \times T$. La température initiale élevé permet d'accepter facilement des solutions qui ne sont pas meilleures que la solutions précédente, ce qui permet de mélanger rapidement les nœuds et ainsi de ne pas être bloqué à cause d'une solution comportant un minimum local. Les paramètres que nous avons retenus sont ceux permettant de minimiser la fonction d'énergie.

A.2 Placement des caches sur la grille avant et après l'algorithme

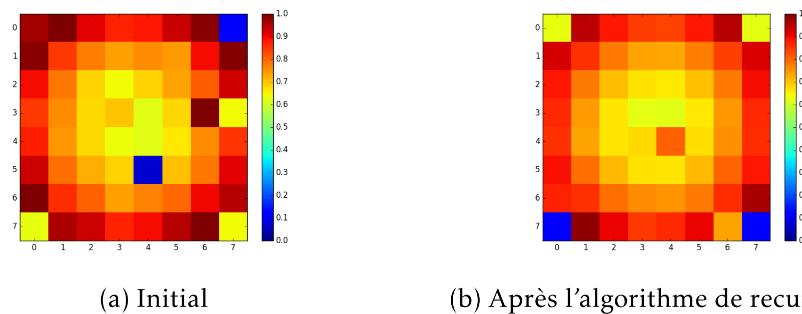
Dans cette partie, nous présentons le placements des caches sur une grille avant et après l'algorithme de recuit simulé. Plus un nœud est rouge, plus ses communications sont coûteuses. Ce coût correspond à la fonction que l'on souhaite minimiser avec l'algorithme de recuit simulé pour un seul cœur, soit $\sum_{i=0}^{N} d_{ij} \times l_{ij}$ où j est le cœur dont on souhaite connaître le coût de ses communications.

L'ensemble des figures A.1 à A.13 montre ce placement avant et après l'application de l'algorithme de recuit simulé. On remarque que le placement du cache critique (en rouge sur les figures initiales) permet de réduire le coût de la communication de ce cache. On remarque également le coût des communications globales a diminué.



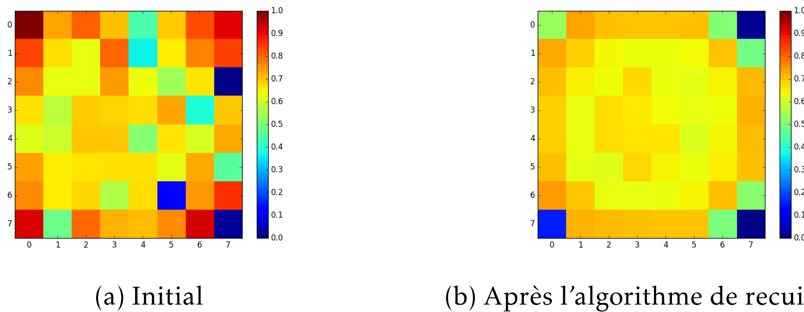
(a) Initial (b) Après l'algorithme de recuit simulé

FIGURE A.1 – Placement des caches avant et après l'algorithme pour `blackscholes`



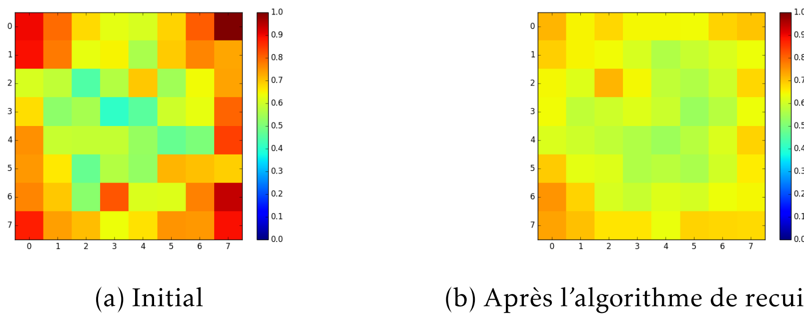
(a) Initial (b) Après l'algorithme de recuit simulé

FIGURE A.2 – Placement des caches avant et après l'algorithme pour `bodytrack`



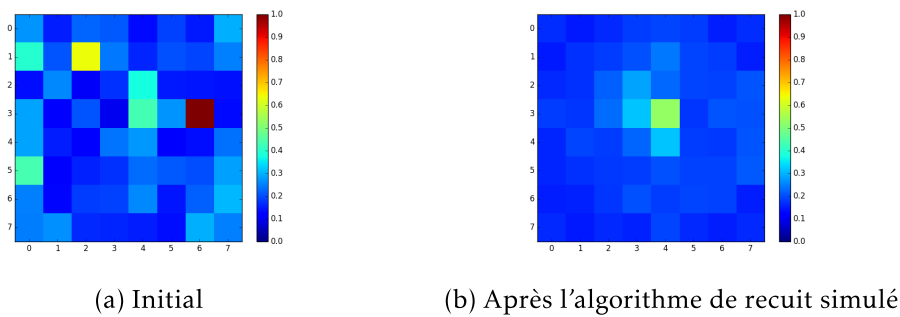
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.3 – Placement des caches avant et après l'algorithme pour `canneal`



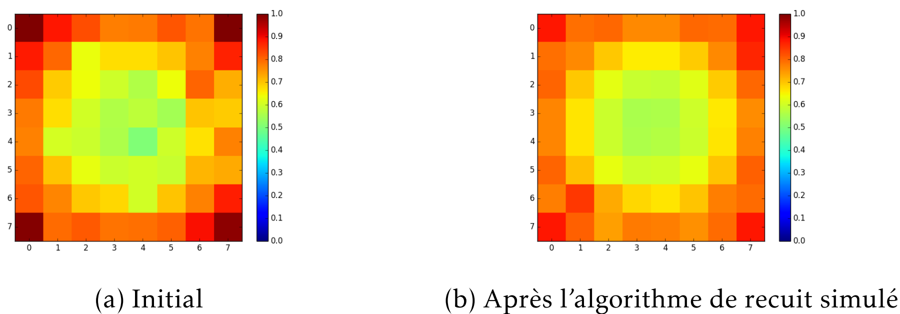
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.4 – Placement des caches avant et après l'algorithme pour `dedup`



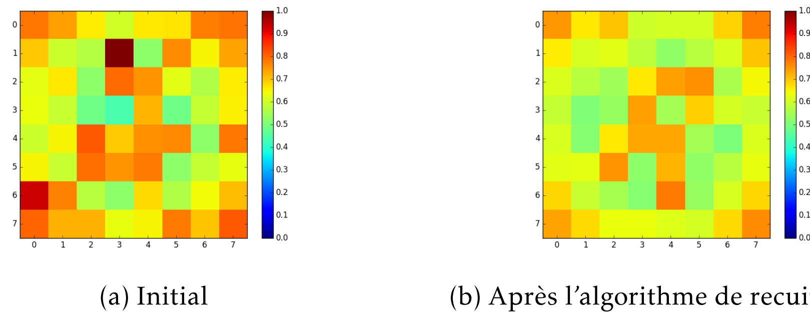
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.5 – Placement des caches avant et après l'algorithme pour `ferret`



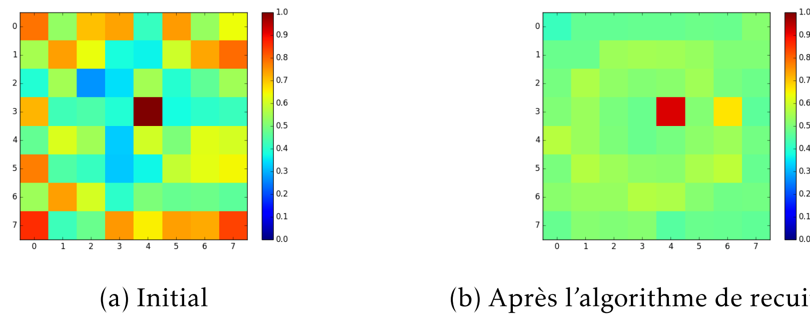
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.6 – Placement des caches avant et après l'algorithme pour `fft`



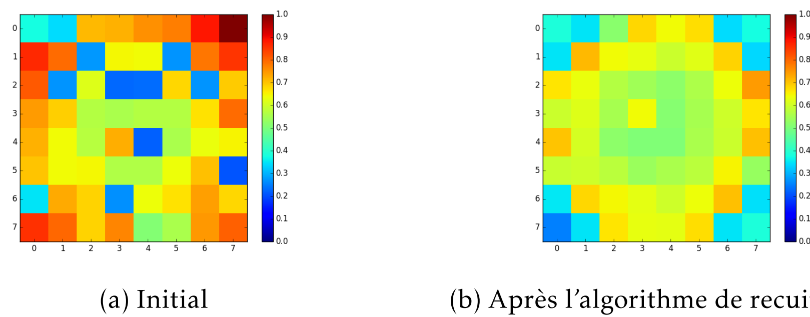
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.7 – Placement des caches avant et après l'algorithme pour `fluidanimate`



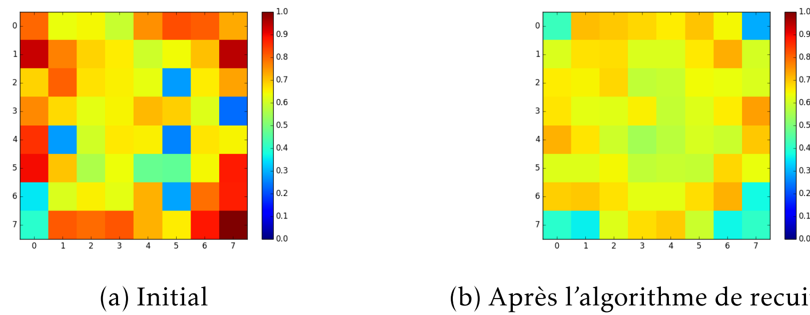
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.8 – Placement des caches avant et après l'algorithme pour `freqmine`



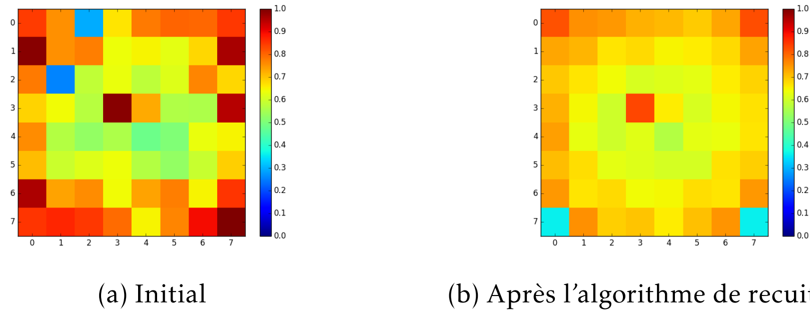
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.9 – Placement des caches avant et après l'algorithme pour `ocean_cp`



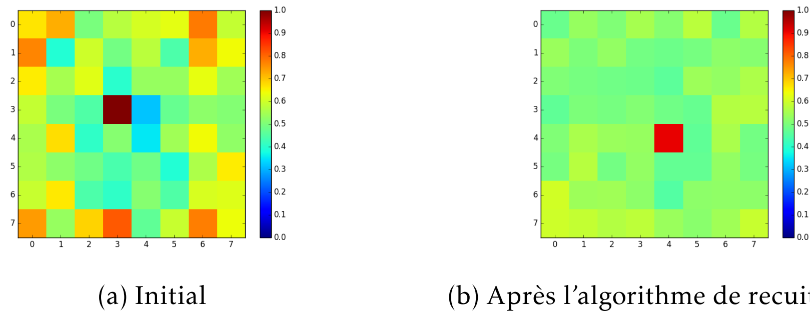
(a) Initial (b) Après l'algorithme de recuit simulé

Figure A.10 – Placement des caches avant et après l'algorithme pour `ocean_ncp`



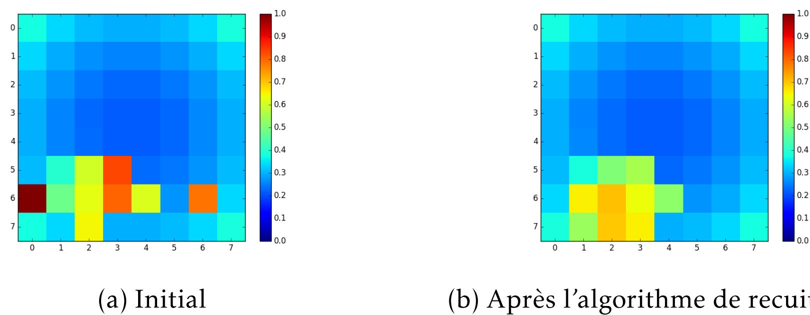
(a) Initial (b) Après l'algorithme de recuit simulé

FIGURE A.11 – Placement des caches avant et après l'algorithme pour `streamcluster`



(a) Initial (b) Après l'algorithme de recuit simulé

FIGURE A.12 – Placement des caches avant et après l'algorithme de recuit pour `vips`



(a) Initial (b) Après l'algorithme de recuit simulé

FIGURE A.13 – Placement des caches avant et après l'algorithme pour `x264`

Publications

Publications dans des conférences internationales

- **Julie Dumas**, Eric Guthmuller, César Fuguet Tortolero, Frédéric Pétrot. Trace-Driven Exploration of Sharing Set Management Strategies for Cache Coherence in manycores. *International NEWCAS Conference (NEWCAS 2017)* IEEE
- **Julie Dumas**, Eric Guthmuller, César Fuguet Tortolero, Frédéric Pétrot. A Method for Fast Evaluation of Sharing Set Management Strategies in Cache Coherence Protocols. *International Conference on Architecture of Computing Systems (ARCS 2017)* (pp. 111-123) Springer

Poster

- **Julie Dumas**, Eric Guthmuller, Frédéric Pétrot. Analyse des accès mémoires dans une machine manycoeur partagée cohérente. *École d'hiver Francophone sur les Technologies de Conception des Systèmes embarqués Hétérogènes (FETCH 2016)*

Bibliographie

[AKPJ09] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh et Niraj K Jha : Garnet : A detailed on-chip network model inside a full-system simulator. *In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[Ali12] Mohammad Alisafaee : Spatiotemporal coherence tracking. *In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 341–350. IEEE Computer Society, 2012.

[APJ09a] Niket Agarwal, Li-Shiuan Peh et Niraj K Jha : In-network coherence filtering : snoopy coherence without broadcasts. *In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 232–243. ACM, 2009.

[APJ09b] Niket Agarwal, Li-Shiuan Peh et Niraj K Jha : In-network snoop ordering (inso) : Snoopy coherence on unordered interconnects. *In High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 67–78. IEEE, 2009.

[ASP17] Hela Belhadj Amor, Hamed Sheibanyrad et Frédéric Pétrot : A meta-routing method to create multiple virtual logical networks on a single hardware noc. *In VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*, pages 200–205. IEEE, 2017.

[BBB+91] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber *et al.* : The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti *et al.* : The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[BDH+06] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi et Steven K Reinhardt : The m5 simulator : Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[BGOS12] Anastasiia Butko, Rafael Garibotti, Luciano Ost et Gilles Sassatelli : Accuracy evaluation of gem5 simulator system. *In 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, pages 1–7. IEEE, 2012.

[BJ14] Mario Badr et Natalie Enright Jerger : Synfull : Synthetic traffic models capturing cache coherent behaviour. *In Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 109–120. IEEE, 2014.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh et Kai Li : The parsec benchmark suite : Characterization and architectural implications. *In Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

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

[BNGD13] Christian Bernard, Huy-Nam Nguyen, Eric Guthmuller et Yves Durand : Design and implementation of an in-network cache coherence protocol. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 298. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013.

[CMM+16] Vincenzo Catania, Andrea Mineo, Salvatore Monteleone, Maurizio Palesi et Davide Patti : Cycle-accurate network on chip simulation with noxim. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 27(1):4, 2016.

[Cou16] Rachel Courtland : Transistors could stop shrinking in 2021. *IEEE Spectrum*, 53(9):9–11, 2016.

[CP99] Jong Hyuk Choi et Kyu Ho Park : Segment directory enhancing the limited directory cache coherence schemes. *In Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 258–267. IEEE, 1999.

[CRG+11] Blas A Cuesta, Alberto Ros, María E Gómez, Antonio Robles et José F Duato : Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. *ACM SIGARCH Computer Architecture News*, 39(3):93–104, 2011.

[DCA+16] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux et Israel Koren : Affinity-based thread and data mapping in shared memory systems. *ACM Computing Surveys*, 49(4):64, 2016.

[DCS+14] Bhavya K Daya, Chia-Hsin Owen Chen, Suvinay Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P Chandrakasan et Li-Shiuan Peh : Scorpio : a 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering. *ACM SIGARCH Computer Architecture News*, 42(3):25–36, 2014.

[EPS06] Noel Eisley, Li-Shiuan Peh et Li Shang : In-network cache coherence. *In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 321–332. IEEE Computer Society, 2006.

[Fle] Flexus Team : http://parsa.epfl.ch/simflex/flexus.html. Flexus website.

[FNW15] Yaosheng Fu, Tri M Nguyen et David Wentzlaff : Coherence domain restriction on large scale systems. *In Proceedings of the 48th International Symposium on Microarchitecture*, pages 686–698. ACM, 2015.

[FPP08] Fabrizio Fazzino, Maurizio Palesi et David Patti : Noxim : Network-on-chip simulator. *URL : http ://sourceforge. net/projects/noxim*, 2008.

[FPRA16] Ricardo Fernández-Pascual, Alberto Ros et Manuel E. Acacio : Optimization of a linked cache coherence protocol for scalable manycore coherence. *In Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*, pages 100–112, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[FTP13] Sahar Foroutan, Yvain Thonnart et Frederic Petrot : An iterative computational technique for performance evaluation of networks-on-chip. *IEEE Trans. Comput.*, 62(8):1641–1655, 2013.

[GdMP08] Pierre Guironnet de Massas et Frédéric Pétrot : Comparison of memory write policies for noc based multicore cache coherent systems. *In Design, Automation and Test in Europe*, pages 997–1002, 2008.

[GM09] P Guironnetde Massas : *Etude de méthodes et mécanismes pour un acces transparent et efficace aux données dans un systeme multiprocesseur sur puce*. Thèse de doctorat, Institut National Polytechnique de Grenoble-INPG, 2009.

[GWM90] Anoop Gupta, Wolf-Dietrich Weber et Todd Mowry : Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. Rapport technique, Computer Systems Laboratory, Stanford University, 1990.

[HDH09] Hemayet Hossain, Sandhya Dwarkadas et Michael C Huang : Ddcache : Decoupled and delegable cache data and metadata. *In Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 227–236. IEEE, 2009.

[HDH11] Hemayet Hossain, Sandhya Dwarkadas et Michael C Huang : Pops : Coherence protocol optimization for both private and shared data. *In Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 45–55. IEEE, 2011.

[JF96] Kazuki JOE et Akira FUKUDA : Analytic modeling of cache coherence based parallel computers. *IEICE TRANSACTIONS on Information and Systems*, 79(7):925–935, 1996.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing et Gurindar S. Sohi : Distributed-directory scheme : Scalable coherent interface. *Computer*, 23(6): 74–77, 1990.

[KGV+83] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi *et al.* : Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KMP+10] George Kurian, Jason E Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C Kimerling et Anant Agarwal : Atac : a 1000-core cache-coherent processor with on-chip optical network. *In Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 477–488. ACM, 2010.

[KR13] Stefanos Kaxiras et Alberto Ros : A new perspective for efficient virtual-cache coherence. *ACM SIGARCH Computer Architecture News*, 41(3):535–546, 2013.

[LC96] Tom Lovett et Russell Clapp : Sting : A cc-numa computer system for the commercial marketplace. *ACM SIGARCH Computer Architecture News*, 24(2):308–317, 1996.

[LDG+15] Hao Liu, Clement Devigne, Lucas Garcia, Quentin Meunier, Franck Wajsburt et Alain Greiner : Rwt : Suppressing write-through cost when coherence is not needed. *In IEEE Computer Society Annual Symposium on VLSI*, pages 434–439. IEEE, 2015.

[LPB06] Mirko Loghi, Massimo Poncino et Luca Benini : Cache coherence tradeoffs in shared-memory mpsocs. *ACM Transactions on Embedded Computing Systems*, 5(2), 2006.

[MH94] Shubhendu S Mukherjee et Mark D Hill : An evaluation of directory protocols for medium-scale shared-memory multiprocessors. *In Proceedings of the 8th international conference on Supercomputing*, pages 64–74, 1994.

[MKK+10] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep et Anant Agarwal : Graphite : A distributed parallel simulator for multicores. *In 16th IEEE International Symposium on High Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[MSB+05] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill et David A Wood : Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[OBM10] Umit Y Ogras, Paul Bogdan et Radu Marculescu : An analytical approach for network-on-chip performance analysis. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 29(12):2001–2013, 2010.

[PG08] Avadh Patel et Kanad Ghose : Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. *In Low Power Electronics and Design (ISLPED), 2008 ACM/IEEE International Symposium on*, pages 247–252. IEEE, 2008.

[RAG10] Alberto Ros, Manuel E Acacio et Jose M Garcia : Cache coherence protocols for many-core cmps. *In Parallel and Distributed Computing*. InTech, 2010.

[RK12] Alberto Ros et Stefanos Kaxiras : Complexity-effective multicore coherence. *In Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, pages 241–251. IEEE, 2012.

[SC15] Sudhanshu Shukla et Mainak Chaudhuri : Pool directory : Efficient coherence tracking with dynamic directory allocation in many-core systems. *In Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 557–564. IEEE, 2015.

[SHW11] D.J. Sorin, M.D. Hill et D.A. Wood : A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 16, 2011.

[Smi82] A.J. Smith : Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[SoC] SoCLib Team : http://www.soclib.fr/trac/dev. SoCLib website.

[SR16] Gabriel Southern et Jose Renau : Analysis of parsec workload scalability. *In Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 133–142. IEEE, 2016.

[SST06] Karin Strauss, Xiaowei Shen et Josep Torrellas : Flexible snooping : Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. *In Computer Architecture, 2006. ISCA'06. 33rd International Symposium on*, pages 327–338. IEEE, 2006.

[SWP06] Vassos Soteriou, Hangsheng Wang et Li-Shiuan Peh : A statistical traffic model for on-chip interconnection networks. *In 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2006.

[Sys] SystemCASS Team : https://www-asim.lip6.fr/trac/systemcass/wiki. SystemCass website.

[TDF93] Manu Thapar, Bruce Delagi et Michael J Flynn : Linked list cache coherence for scalable shared memory multiprocessors. *In Proceedings of Seventh International Parallel Processing Symposium*, pages 34–43. IEEE, 1993.

[TSS+97] Radhika Thekkath, Amit Pal Singh, Jaswinder Pal Singh, Susan John et John Hennessy : An evaluation of a commercial cc-numa architecture-the convex exemplar spp1200. *In Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 8–17. IEEE, 1997.

[UM97] Richard A Uhlig et Trevor N Mudge : Trace-driven memory simulation : A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[Wis] Wisconsin Wind Tunnel Team : http://pages.cs.wisc.edu/~wwt/. Wisconsin Wind Tunnel website.

[WJ90] Andrew W Wilson Jr : Multiprocessor cache simulation using hardware collected address traces. *In Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, pages 252–260, 1990.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh et Anoop Gupta : The splash-2 programs : Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[XDZY11] Yi Xu, Yu Du, Youtao Zhang et Jun Yang : A composite and scalable cache coherence protocol for large scale cmps. *In Proceedings of the international conference on Supercomputing*, pages 285–294. ACM, 2011.

[YD15] Xiangyao Yu et Srinivas Devadas : Tardis : Time traveling coherence algorithm for distributed shared memory. *In Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 227–240. IEEE, 2015.

[YWG+15] Yuan Yao, Guanhua Wang, Zhiguo Ge, Tulika Mitra, Wenzhi Chen et Naxin Zhang : Selectdirectory : a selective directory for cache coherence in many-core architectures. *In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 175–180. EDA Consortium, 2015.

[ZSD10] Hongzhou Zhao, Arrvindh Shriraman et Sandhya Dwarkadas : Space : Sharing pattern-based directory coherence for multicore scalability. *In Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 135–146. ACM, 2010.

[ZSS+14] Lunkai Zhang, Dmitri Strukov, Hebatallah Saadeldeen, Dongrui Fan, Mingzhe Zhang et Diana Franklin : Spongedirectory : Flexible sparse directories utilizing multi-level memristors. *In Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 61–74. ACM, 2014.

Résumé – Suite à l'émergence des architectures *manycores*, le problème du passage à l'échelle des protocoles de cohérence de caches se pose. Il existe deux classes de protocoles : ceux basés sur l'espionnage et ceux utilisant un répertoire. Les premiers, qui doivent transmettre à tous les caches les informations de cohérence, engendrent un nombre important de messages dont peu sont effectivement utiles. En revanche, les protocoles avec répertoires visent à n'envoyer des messages qu'aux caches qui en ont besoin. L'implémentation la plus évidente utilise un champ de bits complet dont la taille dépend uniquement du nombre de cœurs. Ce champ de bits représente la liste des copies. Pour passer à l'échelle, un protocole doit émettre un nombre raisonnable de messages et limiter le matériel utilisé et en particulier celui pour la liste des copies.

Afin d'évaluer et de comparer les protocoles et leurs représentations de la liste des copies, nous proposons tout d'abord une méthode de simulation basée sur l'injection de traces dans un modèle de cache à haut niveau. Cette méthode permet d'effectuer rapidement l'exploration architecturale des protocoles.

Nous proposons également une représentation dynamique de la liste des copies pour le passage à l'échelle des protocoles de cohérence. Pour une architecture à 64 cœurs, 93% des lignes de cache sont partagées par au maximum 8 cœurs. De plus, le système d'exploitation cherche à placer les tâches communicantes proches les unes des autres. Notre représentation tire parti de ces deux observations en utilisant un champ de bits pour un sous-ensemble des copies et une liste chaînée. Le champ de bits correspond à un rectangle à l'intérieur duquel la liste des copies est exacte. La position et la forme de ce rectangle évoluent au cours de la durée de vie des applications. Plusieurs algorithmes pour le placement du rectangle sont proposés et évalués. Pour finir, nous effectuons une comparaison avec les représentations de la liste des copies de l'état de l'art.

Abstract – Cache coherence protocol scalability problem for parallel architecture is also a problem for on chip architecture, following the emergence of manycores architectures. There are two protocol classes : snooping and directory-based. Protocols based on snooping, which send coherence information to all caches, generate a lot of messages whose few are useful. On the other hand, directory-based protocols send messages only to caches which need them. The most obvious implementation uses a full bit vector whose size depends only on the number of cores. This bit vector represents the sharing set. To scale, a coherence protocol must produce a reasonable number of messages and limit hardware ressources used by the coherence and in particular for the sharing set.

To evaluate and compare protocols and their sharing set, we first propose a method based on trace injection in a high-level cache model. This method enables a very fast architectural exploration of cache coherence protocols.

We also propose a new dynamic sharing set for cache coherence protocols, which is scalable. With 64 cores, 93% of cache blocks are shared by up to 8 cores. Futhermore, knowing that the operating system looks to place communicating tasks close to each other. Our dynamic sharing set takes advantage from these two observations by using a bit vector for a subset of copies and a linked list. The bit vector corresponds to a rectangle which stores the exact sharing set. The position and shape of this rectangle evolve over application's lifetime. Several algorithms for coherent rectangle placement are proposed and evaluated. Finally, we make a comparison with sharing sets from the state of the art.

Thèse préparée au CEA-Leti et au laboratoire TIMA
ISBN : 978-2-11-129235-2