Conception d'une architecture extensible pour le calcul massivement parallèle

Ania Kaci

UNIVERSITÉ PARIS-EST

ÉCOLE DOCTORALE MATHÉMATIQUES ET STIC

THÈSE

Pour l'obtention du grade de

Docteur de l'Université Paris-Est
Spécialité : Informatique

par

Ania KACI

Conception d'une architecture extensible pour le calcul massivement parallèle

Directeur de thèse : **Patrick SIARRY**.
Encadrants : **Huy-Nam NGUYEN** et **Amir NAKIB**.

Jury de thèse :

Président du jury :		
Boubaker DAACHI	Professeur des universités	Université Paris 8
Rapporteurs :		
Nouredine MELAB	Professeur des universités	Université Lille 1
Farouk YALAOUI	Professeur des universités	Univ. de Techno. de Troyes
Examinateurs :		
Huy-Nam NGUYEN	Ingénieur de recherche, docteur	Atos-Bull
Amir NAKIB	Maître de Conférences, HDR	Université de Paris-Est Créteil
Patrick SIARRY	Professeur des universités	Université de Paris-Est Créteil

REMERCIEMENTS

Cette thèse constitue une riche expérience qui ne peut s'achever sans remercier les personnes qui m'ont encadrée, aidée et soutenue pendant ces trois dernières années. Je tiens à remercier :

Les membres du jury :

Boubaker DAACHI, Nouredine MELAB et Farouk YALAOUI.

Mon directeur de thèse :

Patrick SIARRY.

Mes encadrants :

Huy-Nam NGUYEN et Amir NAKIB.

Les membres du laboratoire LISSI.

Les membres de la R&D Atos-Bull (Les Clayes-Sous-Bois).

Ma famille :

Mes parents, mon mari et mes sœurs.

Résumé

En réponse à la demande croissante de performance par une grande variété d'applications (exemples : modélisation financière, simulation sub-atomique, bio-informatique, etc.), les systèmes informatiques se complexifient et augmentent en taille (nombre de composants de calcul, mémoire et capacité de stockage). L'accroissement de la complexité de ces systèmes se traduit par une évolution de leur architecture vers une hétérogénéité des technologies de calcul et des modèles de programmation. La gestion harmonieuse de cette hétérogénéité, l'optimisation des ressources et la minimisation de la consommation constituent des défis techniques majeurs dans la conception des futurs systèmes informatiques.

Cette thèse s'adresse à un domaine de cette complexité en se focalisant sur les sous-systèmes à mémoire partagée où l'ensemble des processeurs partagent un espace d'adressage commun. Les travaux porteront essentiellement sur l'implémentation d'un protocole de cohérence de cache et de consistance mémoire, sur une architecture extensible et sur la méthodologie de validation de cette implémentation.

Dans notre approche, nous avons retenu les processeurs 64-bits d'ARM et des co-processeurs génériques (GPU, DSP, etc.) comme composants de calcul, les protocoles de mémoire partagée AMBA ACE et AMBA ACE-Lite ainsi que l'architecture associée « *CoreLink CCN* » comme solution de départ. La généralisation et la paramétrisation de cette architecture, ainsi que sa validation dans l'environnement de simulation gem5, constituent l'épine dorsale de cette thèse.

Les résultats obtenus à la fin de la thèse tendent à démontrer l'atteinte des objectifs fixés.

Mots-clés : *Systèmes informatiques, mémoire partagée, cohérence de cache, consistance mémoire, modélisation transactionnelle, TLM, hétérogénéité, simulation, réseaux d'interconnexion, architecture, calcul parallèle.*

Abstract

In response to the increasing request for performance from a large spectrum of applications (eg. financial modeling, sub-atomic simulation, bioinformatics, etc.), computer systems become more complex and increase in size (number of computing components, memory and storage capacity). The increased complexity of these systems leads to an evolution of their architecture towards the usage a heterogeneous computing technologies and associated programming models. The harmonious management of this heterogeneity together with resource optimization and minimization of power consumption represents major technical challenges in the design of future computer systems.

This thesis addresses one part of this complexity by focusing on shared memory subsystems where all processors share a common address space. Works will focus on the implementation of a cache coherence and memory consistency protocol on an extensible architecture and a methodology for the validation of this implementation.

In our approach, we selected 64-bit ARM processors together with generic co-processors (GPU, DSP, etc.) as computing components, shared memory protocols AMBA ACE and AMBA ACE-Lite and the associated architecture « CoreLink CCN » as a starting solution. Generalization and parameterization of this architecture and its validation in the simulation environment gem5 represents the backbone of this thesis.

The results obtained at the end of the thesis tend to demonstrate the achievement of our objectives.

Keywords : *Computer systems, shared memory, cache coherency, memory consistency, Transactional Level modeling, TLM, heterogeneity, simulation, interconnection networks, architecture, parallel computing.*

Table des matières

Glossaire

ACE	AXI Coherency Extensions
AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
BCS	Bull Coherent Switch
CCN	Cache Coherent Networks
CMP	Chip Multi Processor
CPU	Central Processing Unit
DSP	Digital Signal Processor
FIFO	First In Fist Out
FPU	Floating-point unit
gem5	General Execution-driven Multiprocessor/Memory 5
GPU	Graphics processing unit
IO	Input/Output
IP	Intellectual Properties
LRU	Least Recently Used
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory management unit
NIC	Network Interface Card
NUMA	Non Uniform Memory Access
OMNET++	Objective Modular Network Testbed in C++
RISC	Reduced Instruction Set Computing
RTL	Register-Transfer Level
RTOS	Real-Time Operating System
SoC	System On Chip
TLM	Transaction Level Modeling
UMA	Uniform Memory Access
VHDL	VHSIC Hardware Description Language

Chapitre 1

Contexte et objectifs de la thèse

1.1 Introduction

De nos jours, l'informatique est devenue indispensable dans le quotidien de chacun. Son évolution fait face aux besoins croissants des applications manipulant des données très volumineuses et par conséquent de plus en plus gourmandes en puissance de calcul. Les architectures parallèles les plus récentes sont conçues pour répondre à ces exigences en offrant une haute puissance de calcul. Les multiprocesseurs, interconnectés par des réseaux haut-débit représentent le modèle architectural dominant pour les machines parallèles d'aujourd'hui. Ils sont devenus une technologie rentable même pour des systèmes de tailles moyennes [Hagersten and Hill, 1999].

Les systèmes informatiques les plus récents se composent principalement d'un ensemble, souvent hétérogène, de processeurs, de nombreux accélérateurs spécifiques à certaines applications, de périphériques entrée/sortie, de mémoires, etc. Ils peuvent être classifiés selon le modèle de programmation de leurs mémoires [Culler et al., 1999] : systèmes à mémoire partagée, où la mémoire est accessible à tous les processeurs comme étant un espace d'adressage commun et global et systèmes à mémoires distribuées, où les processeurs échangent les données de leur mémoire entre eux par envoi de messages.

Le principe de la mémoire partagée exige la mise en place d'un protocole rigoureux servant à assurer la cohérence des données entre les différents caches du système et la consistance par rapport à la mémoire Il définit les régles permettant de gérer les accès à la mémoire partagée. La complexité de leur conception varie en fonction du nombre de composants de calcul dans le système. A titre d'exemple, le nombre d'états à modéliser dans le protocole AMBA ACE de ARM est 7000^N, où N est le nombre de composants de calcul [Oury et al., 2015]. Cette thèse adresse ce problème de complexité.

Nous focalisons nos travaux sur les sous-systèmes à mémoire partagée en proposant l'implémentation d'un protocole de cohérence de cache et de consistance mémoire sur une architecture générique et extensible. Cette architecture sera conçue autour de l'interconnexion existante *CoreLink CCN* de ARM qui sera étendue à des configurations supportant des centaines de composants hétérogènes de calcul. Nous proposons ensuite une méthodologie de validation par simulation, de cette architecture, qui sera basée sur la modélisation au niveau transactionnel (*TLM*). Dans ce chapitre, nous commençons par

présenter le contexte général de la thèse en introduisant les systèmes à mémoires partagées, la consistance mémoire et la cohérence de cache. Nous décrivons par la suite les principaux objectifs de cette thèse et donnons une description résumée des contributions de cette thèse ainsi que sa structure.

1.2 Contexte de la thèse

1.2.1 Systèmes à mémoire partagée

Les systèmes considérés dans nos travaux suivent l'architecture des systèmes multiprocesseurs où les processeurs sont interconnectés via un réseau et partagent l'espace mémoire. Chaque cœur du système peut accéder à chaque emplacement mémoire avec les mêmes ou différentes latences. Les processeurs peuvent avoir des mémoires locales servant de cache aux données.
L'architecture générique d'un système à multiprocesseurs est basée sur l'utilisation d'un réseau hiérarchique servant à interconnecter ses nœuds de calcul et la mémoire partagée. L'accès peut adopter l'un des deux modèles suivants : le modèle d'accès uniforme à la mémoire (UMA pour *Uniform Memory Access*) et le modèle d'accès non-uniforme à la mémoire (NUMA pour *Non-Uniform Memory Access*) [Culler et al., 1999].

Dans le modèle UMA, pour la même adresse, chaque processeur a le même temps d'accès intrinsèque à l'emplacement mémoire correspondant [Manchanda and Anand, 2010]. L'organisation d'un système basé sur le modèle UMA sera illustrée dans le chapitre 2.

Le modèle NUMA distribue la mémoire à travers les processeurs [Baxter et al., 1999]. La latence d'accès à une mémoire non-locale dépend de la distance entre le processeur et l'emplacement physique de la mémoire demandée. Plusieurs approches ont été proposées dans le but de réduire cette latence. Une des solutions est l'utilisation des caches [Gharachorloo et al., 1990a]. L'organisation d'un système basé sur le modèle NUMA sera illustrée dans le chapitre 2.
Nous détaillerons les systèmes à mémoire partagée dans le chapitre 2.

1.2.2 Problème de consistance mémoire et de cohérence de cache

La consistance mémoire

Lorsque l'on conçoit une architecture à mémoire partagée, une attention particulière est portée au modèle de consistance mémoire. Le modèle de consistance mémoire est défini par les contraintes imposées à l'ordre dans lequel les requêtes mémoires sont traitées vis-à-vis de leur ordre d'apparition dans le programme exécuté. Dans les systèmes à mémoire partagée, les processeurs peuvent effectuer simultanément des opérations de lecture et d'écriture vers des emplacements communs de la mémoire partagée, c'est pourquoi, l'ordre d'accès à ces différents emplacements est important. Ces opérations peuvent être datées afin de définir un ordre total sur les opérations mémoire, à l'aide d'horloges de Lamport [Puzak, 1985a]. Dans [Puzak, 1985a], $< p$ est noté comme étant l'ordre dans lequel les opérations apparaissent dans le programme. Par rapport à la mémoire, cet ordre est total pour un système monoprocesseur, mais partiel pour un système multiproces-

seurs. Cet ordre peut être différent de l'ordre du programme, ceci dépend du modèle de consistance mémoire présent dans l'architecture.

Considérons un exemple où un *thread* $th1$ modifie la valeur d'une variable v à new_v et un autre *thread* $th2$ lit la valeur de la variable v. Si les deux *threads* sont évoqués simultanément, le comportement attendu est que $th2$ lise la valeur new_v ; or le résultat dépend de l'ordre d'exécution des opérations de lecture et d'écriture de la variable v. La valeur new_v est lue par $th2$ si une exécution séquentielle de $th1$ et $th2$ est réalisée. Cet ordre est une caractéristique du modèle de consistance mémoire.

Si les règles du modèle de consistance mémoire sont respectées, la consistance mémoire est garantie et les résultats des opérations de lecture et d'écriture en mémoire seront prévisibles. Ce problème sera détaillé dans le chapitre 2.

La cohérence de cache

L'utilisation des caches dans les systèmes remonte à 1962 [Prybylski et al., 1988]. Ils sont largement utilisés dans les architectures actuelles. Un cache est une mémoire locale d'accès rapide servant à accélérer l'accès aux données les plus utilisées et les plus récentes. Le principe de la mémoire partagée exige la mise en place d'un protocole rigoureux servant à assurer la cohérence des données entre les différents caches et la consistance par rapport à la mémoire [Correale Jr, 1991a].
A tout instant, le contenu d'une même adresse doit être le même dans tous les caches.
On distingue deux familles de protocoles de cohérence :

- Les protocoles à écriture simultanée (*write-through*) : pour lesquels toutes les écritures sont propagées vers la mémoire [Archibald and Baer, 1986] ;
- Les protocoles à écriture différée (*write-back*) : pour lesquels les écritures après réservation sont recopiées en mémoire uniquement lorsque cela est nécessaire [Li and Hudak, 1989].

La cohérence de cache peut être gérée par des techniques matérielles ou des techniques logicielles que nous détaillerons dans le chapitre 2. Initialement, la cohérence de cache était gérée par logiciel, afin de profiter de la flexibilité des solutions logicielles. Toutefois, en raison de leurs mauvaises performances, la gestion des protocoles par des solutions matérielles a été privilégiée. Le protocole AMBA ACE, proposé par ARM [AMBA and Specification-AXI, 2011], est un exemple de protocole permettant de gérer la cohérence par une solution matérielle.

1.3 Objectifs de la thèse

La conception des systèmes à mémoire partagée étant de plus en plus complexe, des méthodes d'évaluation et d'analyse des architectures en amont sont nécessaires. La modélisation RTL (*Register Transfer Level*) arrivant plus tard dans le processus de conception, la réalisation de modèles au niveau TLM (*Transaction Level Modeling*) pour supporter la validation dynamique des architectures dès les premières étapes du flot de conception est

très importante.

L'objectif principal de cette thèse est d'implémenter un protocole de cohérence de cache et de consistance mémoire, sur une architecture extensible et de proposer une méthodologie de validation de cette implémentation. Ce travail de vérification se révèle particulièrement complexe vu la taille de l'espace d'états à prendre en compte. Dans le cas d'étude de la vérification du protocole que nous avons retenu pour la gestion de la cohérence de cache et de la consistance mémoire dans notre architecture, le nombre de combinaisons des éléments de la spécification ACE est très élevé. Chaque combinaison constituant un scénario doit être complètement vérifiée. Cette taille augmente avec le nombre de composants de calcul dans le système. En effet, dans le protocole AMBA ACE, le nombre de scénarii à vérifier est approximativement de 7000^N, où N est le nombre de composants de calcul [Oury et al., 2015].

Dans notre approche, nous avons retenu des processeurs 64-bits d'ARM et des co-processeurs génériques (GPU, DSP, etc.) comme composants de calcul, les protocoles de mémoire partagée AMBA ACE et AMBA ACE-Lite, ainsi que l'architecture associée « *CoreLink CCN* » comme solution de départ. Nos travaux porteront sur la généralisation et la paramétrisation de cette architecture, ainsi que sa validation dans l'environnement de simulation gem5.

Les résultats obtenus à la fin de la thèse tendent à démontrer l'atteinte des objectifs fixés.

1.4 Plan du manuscrit

La suite du manuscrit est structurée comme suit. Le chapitre 2 présente un état de l'art portant sur les architectures à mémoire partagée, leurs modèles de programmation, la consistance mémoire, la cohérence de cache ainsi que les mécanismes de gestion de la cohérence associés. Le chapitre 3 décrit les systèmes hybrides à base de multiprocesseurs ARM et plus particulièrement ceux à base de réseaux d'interconnexion *CoreLink CCN*. Il introduit également les différents mécanismes de gestion de la cohérence des données dans ces systèmes en donnant une description détaillée du protocole de cohérence de cache AMBA ACE. Le chapitre 4 portera sur la description détaillée de l'architecture générique proposée et l'implémentation du protocole AMBA ACE sur cette architecture. Le chapitre 5 est consacrée à la méthodologie de vérification et de validation. Après une présentation des différentes techniques de validation des systèmes informatiques et de l'environnement de simulation gem5, nous donnons les détails de nos implémentations effectuées dans gem5. Le chapitre 6 fournit les résultats de validation de l'architecture proposée. Ces résultats permettent d'évaluer notre modélisation et implémentation du protocole de cohérence de cache AMBA ACE sur cette architecture. Quelques mesures de performance sont effectuées afin d'évaluer cette méthodologie de vérification. Enfin, une conclusion sur nos travaux et des perspectives pour des recherches ultérieures sont données dans le chapitre 6.5.
Cette thèse est soumise au régime de confidentialité de la société Bull S.A.S et donc ne sera pas accessible au grand public.

Chapitre 2

Etat de l'art : les systèmes à mémoire partagée, la consistance mémoire et la cohérence de cache

Ce chapitre est consacré à la présentation du contexte général de la thèse. La première partie décrit les systèmes à mémoire partagée en présentant les différents modèles d'organisation de la mémoire et les techniques d'interconnexion des composants. Nous donnons par la suite trois exemples d'architectures de systèmes à mémoire partagée. Dans la deuxième partie, nous présentons la consistance mémoire en décrivant la consistance séquentielle, la consistance forte et la consistance relaxée. Dans la troisième partie, nous définissons la cohérence de cache et présentons les mécanismes de sa gestion. Nous décrivons par la suite, un protocole de cohérence de cache proposé par ARM, à savoir le protocole $AMBA\ ACE$.

2.1 Les systèmes à mémoire partagée

L'une des classes les plus importantes des machines parallèles est la classe des multiprocesseurs à mémoire partagée. Dans ce type de machines parallèles, la communication résultant des opérations d'accès à la mémoire est implicite. On la trouve dans des systèmes comportant de quelques processeurs à des centaines de *clusters* souvent hétérogènes [Lenoski and Weber, 2014].
Les systèmes à mémoire partagée consistent en un ensemble de nœuds de calcul interconnectés par un réseau hiérarchique et composés de plusieurs cœurs se partageant l'espace mémoire auquel ils sont reliés via ce réseau, comme illustré dans les figures 2.2 et 2.3 [Culler et al., 1999]. La coopération et la coordination entre les cœurs sont accomplies en lisant et en modifiant des variables partagées. Une écriture d'un nœud de calcul est visible aux lectures des autres nœuds. Des opérations atomiques de synchronisation entre les processus indépendants sont établies.
Les programmes parallèles, pour ces systèmes, sont structurés en segments privés pour les données privées et segments partagés se trouvant dans la même région partagée entre plusieurs processus ou *threads* du programme, comme illustré dans la figure 2.1 [Kumar et al., 1994].
Chaque cœur peut accéder à chaque emplacement mémoire avec différentes latences. La latence d'accès à la mémoire est un des facteurs clés déterminant les performances d'un

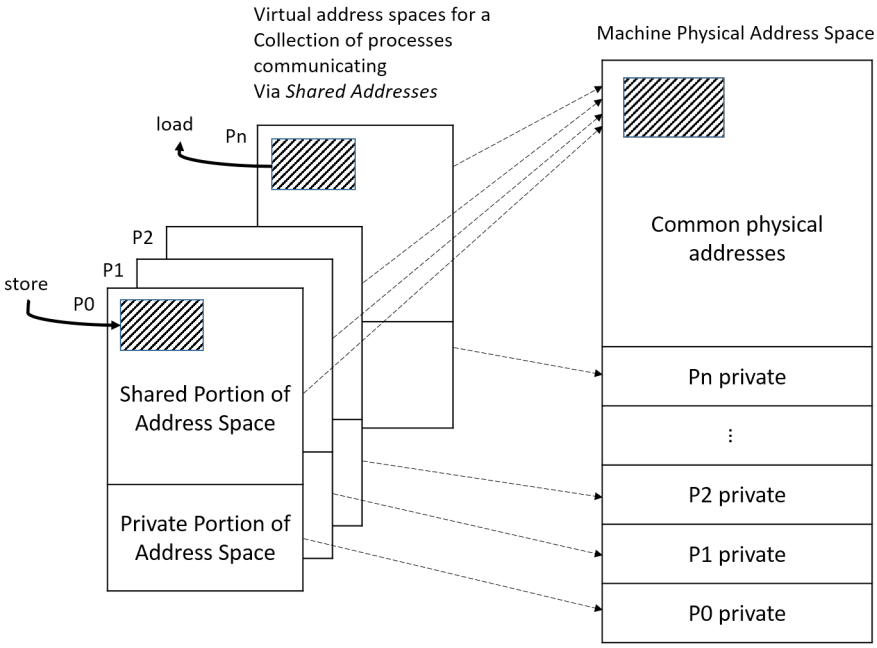

FIGURE 2.1 – Exemple d'un modèle mémoire pour un programme parallèle à mémoire partagée.

tel système. Les optimisations proposées dans le but de réduire cette latence dans les systèmes multicœurs sont multiples [Alpern et al., 1994]. Une des solutions est d'introduire des mémoires locales servant de cache aux données [Wilson Jr, 1987].

2.1.1 Les modèles de mémoire

L'organisation de la hiérarchie mémoire est un élément clé dans la conception des systèmes à mémoire partagée. La latence d'accès à la mémoire a un impact important sur les performances d'un tel système. Cet accès peut adopter l'un des deux modèles suivants :

Le modèle UMA

L'approche la plus naturelle pour construire des systèmes à mémoire partagée est de maintenir un accès uniforme à la mémoire, comme illustré dans la figure 2.2. Chaque accès à la mémoire est un message envoyé sur le bus/réseau d'interconnexion [Protic et al., 1998].

Le modèle NUMA

L'approche alternative est d'assigner à un groupe de processeurs une mémoire, comme illustré dans la figure 2.3. Le contrôleur de la mémoire locale détermine si l'accès est à la mémoire locale ou bien une transaction à envoyer sur le réseau. L'accès aux données privées est souvent effectué localement [Schweizer and Carroll, 1990].

Dans les systèmes à mémoire partagée, la hiérarchie mémoire peut suivre trois modèles qui dépendent fortement de la taille du système considéré.

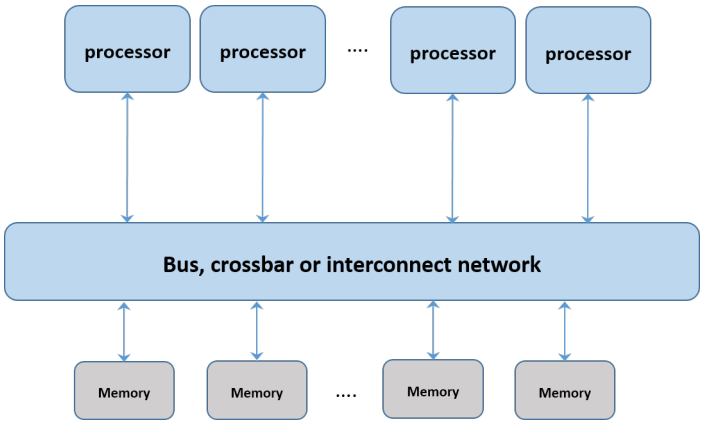

FIGURE 2.2 – Exemple d'un modèle UMA.

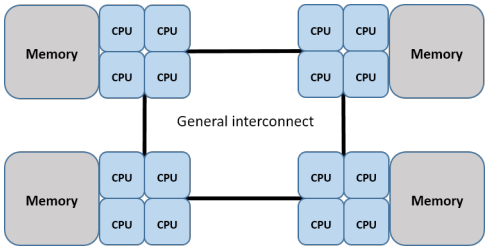

FIGURE 2.3 – Exemple d'un modèle NUMA.

— Dans la première catégorie, appelée l'approche du cache partagé, le système d'interconnexion est situé entre les processeurs et le premier niveau de cache partagé, qui est lui même connecté à la mémoire partagée. Cette approche a été utilisée pour connecter un nombre de processeurs allant de 2 à 8 processeurs [Nguyen et al., 2014] ;
— La deuxième catégorie est l'approche de la mémoire partagée basée sur un bus. Le bus est situé entre les caches des processeurs et la mémoire partagée, comme illustré dans la figure 2.4. Elle a été largement utilisée dans les systèmes de taille moyenne allant jusqu'à 30 processeurs ;
— La troisième catégorie est scalable à plusieurs nœuds de calcul. Le système d'interconnexion est placé entre les caches et la mémoire partagée. Il est considéré comme un réseau point à point et la mémoire est divisée en plusieurs modules logiques qui sont connectés aux différents points du système d'interconnexion [Rodgers, 1985].

Dans la section suivante, nous détaillons ces différents modes d'interconnexion.

2.1.2 Les systèmes d'interconnexion

Avec l'émergence des systèmes multicœurs, un intérêt particulier a été porté aux techniques d'interconnexion. En effet, les goulots d'étranglement liés aux communications peuvent compromettre la performance. La conception de nombreux systèmes a été basée sur la communication via des bus. Cependant, ils sont vite arrivés à saturation avec l'augmentation du nombre de cœurs [Dally and Towles, 2004a].
Les systèmes hétérogènes les plus récents communiquent à travers un réseau d'interconnexion. Le choix d'un système d'interconnexion est souvent fortement lié aux objectifs, de performances, de coûts et de consommation en énergie, de chaque constructeur [Rixner

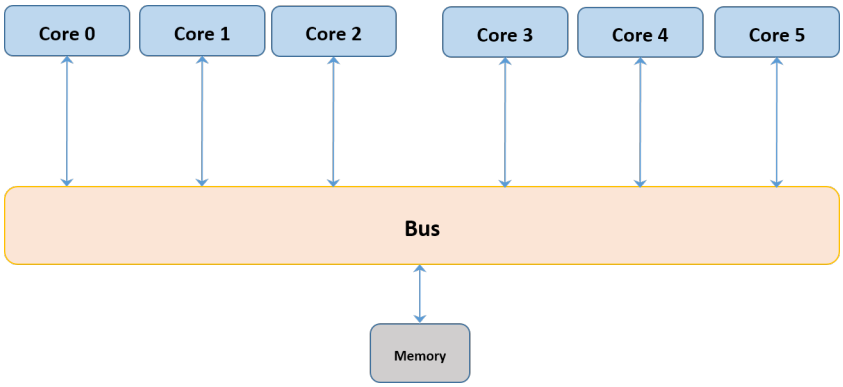

FIGURE 2.4 – Exemple d'un système basé sur une interconnexion par bus.

et al., 2000]. Dans ce qui suit, nous présentons les 3 systèmes d'interconnexion les plus communs : les bus, les *crossbars* et les réseaux.

Les bus

Les bus se composent d'un ensemble de fils partagés entre les composants. Un bus peut être vu comme étant un canal unique partagé entre plusieurs nœuds de calcul. La figure 2.4 illustre un ensemble de cœurs connectés à la mémoire à travers un bus.
La fréquence d'un bus dépend de la longueur des fils utilisés pour connecter tous les composants à ce bus. Cette contrainte de connexion de tous les composants à un seul bus limite sa fréquence avec l'augmentation du nombre de cœurs dans le système. Le trafic dû aux requêtes d'accès à la mémoire et sa mise à jour , contribue à augmenter significativement la charge du bus partagé [Kumar et al., 2005].
A tout instant, une seule requête est exécutée, c'est pourquoi, les interconnexions basées sur les bus ne sont plus adaptées aux systèmes multicœurs. Afin d'y remédier, des bus multiples ont été conçus dans les systèmes les plus récents. Ces bus sont répartis selon les types des composants et les messages véhiculés dans le système, ainsi que d'autres caractéristiques du système [Salminen et al., 2002]. A titre d'exemple, le protocole de bus AMBA de ARM spécifie un bus haute performance (AHB) et un bus périphérique (APB) dans le même système. Plus de détails sont donnés dans la section 3.1.2.

Les *crossbars*

Dans un système basé sur une interconnexion par crossbar, les composants sont connectés au crossbar via des canaux séparés. Tant que le crossbar n'a pas reçu des requêtes sur le même canal, les messages peuvent être envoyés en parallèle [Goodman, 1991]. La figure 2.5 illustre un système dont les composants sont interconnectés via un *crossbar*. Dans cet exemple, le processeur 0 peut envoyer un message à la mémoire *Memory* 0 au même cycle que le processeur 1, qui envoie un message à la mémoire *Memory* 1. Avec l'augmentation du nombre de composants, l'arbitrage devient complexe et les performances peuvent se dégrader [Purcell and Cheng, 2004].

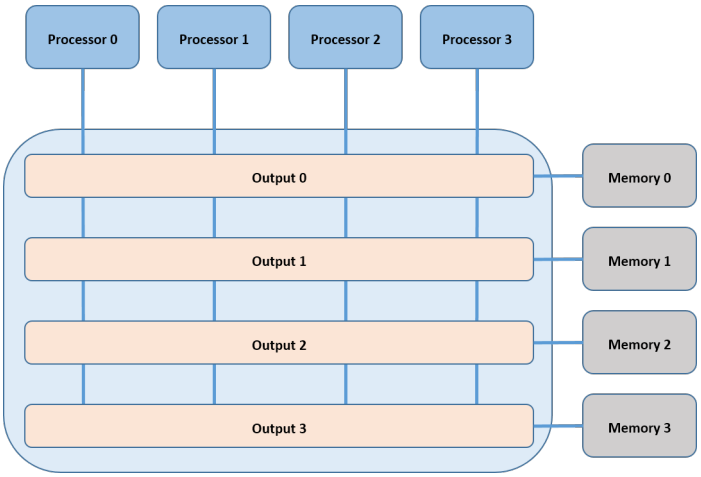

FIGURE 2.5 – Exemple d'un système basé sur une interconnexion par *crossbar*.

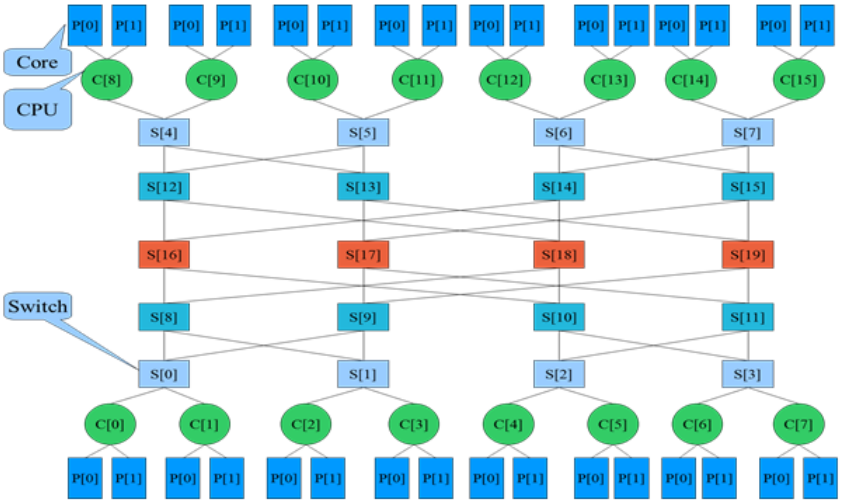

FIGURE 2.6 – Exemple d'un système basé sur un réseau d'interconnexion avec une topologie $fat\ tree$.

Les réseaux d'interconnexion

Dans les réseaux d'interconnexion, la communication est distribuée entre les *clusters* de calcul en utilisant des éléments de routage intermédiaires. Cette distribution augmente la scalabilité. A l'intérieur de chaque *cluster*, la mémoire est partagée entre les processeurs. Le temps de communication entre les noeuds de calcul dépend de la distance entre eux. Il est alors non-uniforme [Alpern et al., 1994].
Les *clusters* sont interconnectés entre eux suivant une certaine topologie. Le choix de la topologie et la stratégie de routage dans la conception de réseaux d'interconnexion est important car il affecte directement les performances du système. Les systèmes avec un grand nombre de nœuds de calcul doivent favoriser l'implémentation d'une topologie de réseau distribuée. La figure 2.6 illustre une topologie $fat\ tree$ d'un système de 16 processeurs [Nguyen et al., 2014].

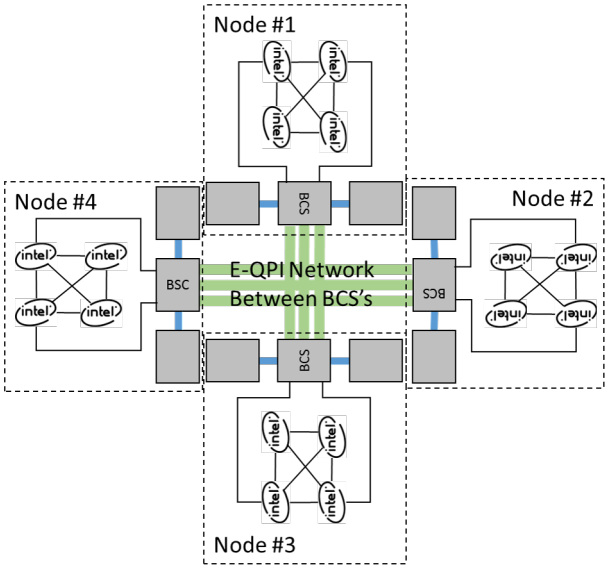

FIGURE 2.7 – Architecture BCS à 4 noeuds et 16 sockets.

2.1.3 Quelques architectures de systèmes à mémoire partagée

Les premiers systèmes multiprocesseurs ont été introduits par des compagnies comme *Synapse*, *Flex*, *Sequent* and *Myrias* [Chan et al., 1993]. Ils étaient composés de 10 à 20 microprocesseurs. Après l'introduction du microprocesseur *Intel i*80386 à 32 bits, ces systèmes ont connu un succès commercial sans précédent. Au milieu des années 90, ils sont devenus omniprésents dans l'industrie allant d'ordinateurs de bureau à de larges serveurs supportant une centaine de processeurs. Cette évolution n'a pas cessé. Aujourd'hui, on parle de supercalculateurs. Dans cette section, nous donnons des exemples de systèmes à mémoire partagée en citant les plus connus pour les applications suivantes : serveurs et supercalculateurs.

Bullion

Il s'agit d'une famille de différents modèles de serveurs *Bull*. Ces serveurs comprenent de nombreuses innovations technologiques permettant de répondre aux besoins courants des services informatiques. Ils sont basés sur une architecture modulaire dotée de 2 à 16 processeurs Intel Xeon supportant jusqu'à 160 cores, 320 processeurs logiques et 24To de mémoire. Chaque module BCS (Bull Coherent switch) regroupe ensemble 4 CPUs qui constitue un sous système à mémoire partagée, comme illustré dans la figure 2.7. Cette connexion direct permet de fournir des latences faibles. BCS utilise le protocole X-QPI entre les modules BCS et le protocole QPI entre les sockets CPUs [BUL, 2016].

Cray T3E

Le *Cray T3E* a été conçu pour supporter des milliers de processeurs partageant un espace d'adressage global [Scott, 1996]. Chaque nœud de calcul contient un processeur Alpha DEC, une mémoire locale, une interface réseau intégrée dans le contrôleur mémoire et un réseau de *switchs*. La machine est organisée, comme illustrée sur la figure 2.8, en cube à 3 dimensions où chacun des nœuds est connecté à ses 6 voisins via des liens

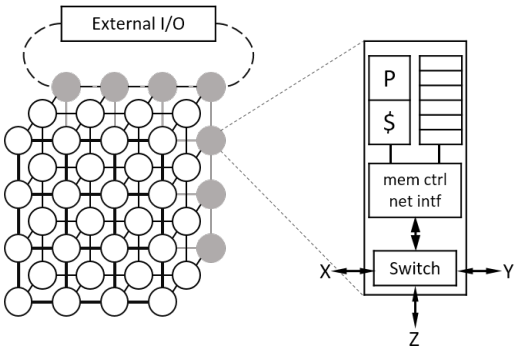

FIGURE 2.8 – Organisation de Cray T3E

points à points de 480MB/s. Le contrôleur mémoire capture l'accès à la mémoire distante et effectue une transaction vers le contrôleur mémoire du nœud distant. Ce message est automatiquement routé vers le nœud *destination* tout en passant par des nœuds intermédiaires. Les données ne sont pas stockées dans les caches, ceci est dû à l'inexistence d'un mécanisme *hardware* permettant d'assurer la consistance mémoire.
Les périphériques I/O sont distribués sur la surface du cube et à travers une collection de nœuds. Ils sont connectés aux composants externes via un réseau I/O additionnel.

Sun Enterprise Server

Le serveur à multiprocesseurs *Sun Ultrasparc Enterprise* est un *design* à grande échelle. La figure 2.9 montre sa structure physique et son organisation logique [Chan et al., 1993]. Il utilise une structure hiérarchique où chaque carte est soit un processeur dual avec une mémoire, soit des I/Os avec un bus pipeliné à 256 bits délivrant un débit de 2.5 GB/s. La configuration complète supporte 16 cartes des deux types. La carte contenant le processeur consiste en deux processeurs *Ultrasparc* avec chacun un cache de niveau 1 de 16KB, un cache de niveau 2 de taille égale à 512 KB, deux bancs mémoire de 512 bits et un *switch* interne. La carte des I/Os contient 3 bus pour les extensions I/O, un connecteur *SCSI*, un port Ethernet 100bT et deux interfaces *FiberChannel*. Une configuration complète peut contenir 24 processeurs et 6 cartes I/O. La mémoire est accessible à la même distance à tous les processeurs qui y accèdent via le bus commun.

2.2 La consistance mémoire

Les mono-processeurs présentent une vue simple et intuitive de la mémoire aux programmeurs. A tout instant, une opération mémoire est supposée s'exécuter suivant l'ordre spécifié dans le programme. Une opération de lecture retourne alors la valeur de la dernière écriture au même emplacement mémoire. Cet ordre n'est pas obligatoirement maintenu entre toutes les opérations mémoire, cependant, un ordre séquentiel doit être assuré pour les opérations d'accès au même emplacement mémoire [Gharachorloo et al., 1990b]. Le ré-ordonnancement des opérations d'accès permet de fournir des implémentations mono-processeur efficaces. Dans le cas des systèmes à mémoire partagée, plusieurs processeurs concurrents accèdent à des emplacements mémoire communs, c'est pourquoi le comportement des opérations d'accès doit être spécifié clairement. En effet, afin d'écrire des programmes efficaces et corrects pour des systèmes à mémoire partagée, le programmeur

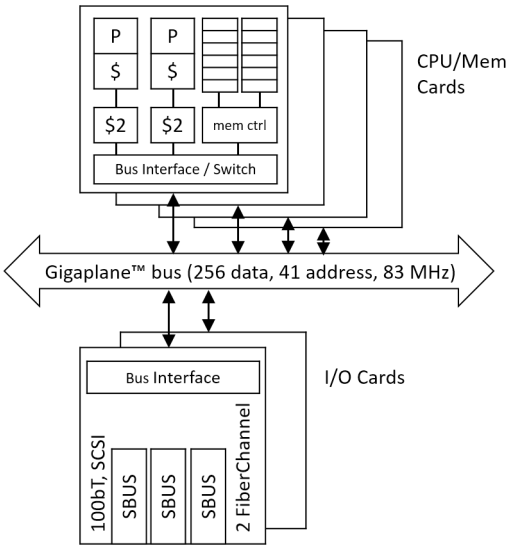

FIGURE 2.9 – Organisation de Sun Enterprise Server

doit connaître le comportement de la mémoire lors des opérations de lecture et d'écriture effectuées par des processeurs concurrents.
Le modèle de consistance mémoire pour les systèmes à mémoire partagée donne une spécification de l'apparence de la mémoire au programmeur, pour que le comportement attendu corresponde au vrai comportement supporté par le système [Gharachorloo et al., 1990b]. Ces modèles définissent un ordre d'accès à la mémoire lors de l'exécution d'un programme. Intuitivement, une opération de lecture doit retourner la valeur de la dernière opération d'écriture au même emplacement mémoire. Dans un mono-processeur, la dernière valeur est définie par l'ordre du programme. Ce n'est pas le cas dans les systèmes multiprocesseurs où les programmes sont exécutés sur des processeurs différents. Nous présentons dans ce qui suit quelques modèles de consistance mémoire.

2.2.1 La consistance séquentielle

Une extension intuitive du modèle à mono-processeur a été appliquée au cas multiprocesseurs. Ce modèle a été proposé par *Lamport* comme suit [Lamport, 1979].
Un système multiprocesseurs est consistant séquentiellement s'il existe un ordre séquentiel tel que le résultat d'une exécution n'est le même que si les opérations de tous les processeurs ont été exécutées dans un ordre séquentiel, et les opérations de chaque processeur individuel apparaissent dans cette séquence suivant l'ordre spécifié dans son programme. Chaque processeur est tenu d'effectuer ses opérations d'accès dans l'ordre du programme. A tout instant, une seule opération est prise en compte par la mémoire, ainsi, les opérations sont atomiques. Un ordre séquentiel unique résulte de l'arbitrage entre les opérations des différents processeurs.
Deux aspects sont à considérer : (i) maintenir l'ordre du programme à travers les opérations issues des processeurs individuels, et (ii) maintenir un seul ordre séquentiel à travers les opérations de tous les processeurs [Cohen and Schirmer, 2010].
Ce modèle donne au programmeur une vue simple du système. Chaque processeur émet des opérations vers la mémoire dans l'ordre du programme qu'il exécute. Un *switch* four-

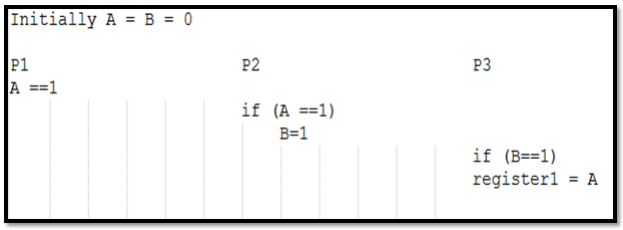

FIGURE 2.10 – Exemple d'un modèle de consistance séquentielle.

nit une sérialisation globale à travers toutes les opérations vers la mémoire en connectant un seul processeur à la mémoire à chaque pas de temps. La figure 2.10 illustre un exemple montrant les sémantiques d'un modèle de consistance séquentielle. 3 processeurs partagent les variables A et B, toutes les deux initialisées à 0. Supposons que P2 retourne la valeur 1 (modifiée par P1) pour sa lecture de variable A et écrit dans la variable B, P3 retourne alors la valeur 1 (écrite par P2) pour la variable B. L'effet de l'écriture de P1 est vu par le système entier. L'exécution séquentielle garantit que P3 voit l'effet de l'écriture de P1 et par conséquent retourne la valeur 1 pour sa lecture de variable A.

2.2.2 La consistance forte (*Total store OrderTSO*)

Le modèle TSO est largement utilisé dans les implémentations SPARC et les architectures x86. Il s'agit d'un modèle plus relaxé que le modèle de consistance séquentielle. Il permet l'utilisation des *buffers* d'écriture (en FIFO) pour chaque cœur du système. Des instructions FENCE (barrières mémoire) sont utilisées. L'exécution d'une instruction FENCE sur un cœur c assure que les opérations mémoire effectuées par c avant le FENCE soient placées en mémoire avant les opérations mémoire issues par c après le FENCE [Zucker, 1992].
Une exécution TSO est définie comme suit :
(a) Tous les cœurs insèrent leurs opérations *load* et *store* dans un ordre mémoire m qui dépend de leur ordre dans le programme p, selon si elles sont vers la même ou différentes adresses (i.e., a==b ou a !=b). Il existe 4 cas :

- Si L(a) <p L(b) alors L(a) <m L(b) /* *Load-Load* */
- Si L(a) <p S(b) alors L(a) <m S(b) /* *Load-store* */
- Si S(a) <p S(b) alors S(a) <m S(b) /* *store-store* */
- Si S(a) <p L(b) alors S(a) <m L(b) /* *store-Load* */ /* Modification 1 : *Enable FIFO WriteBuffer* */

(b) Chaque *load* obtient la valeur du dernier *store* avant ce *load* à la même adresse :
Valeur de L(a) = Valeur de MAX <m {S(a) | S(a) <m L(a) ou S(a) <p L(a)}
(c) La dernière règle de (a) est modifiée comme suit :

- Si S(a) <p FENCE alors S(a) <m FENCE /* *store - FENCE* */
- Si FENCE <p L(a) alors FENCE <m L(a) /* *FENCE - load* */

2.2.3 La consistance relaxée

Plusieurs modèles de consistance mémoire relaxée ont été proposés. Ils sont principalement basés sur deux caractéristiques clés : (i) comment relaxer la nécessité de l'ordre du

programme, et (ii) comment relaxer l'exigence de l'atomicité des écritures [Devietti et al., 2011].
On distingue d'une part des modèles basés sur la relaxation de l'ordre entre une opération d'écriture et l'opération de lecture la suivant, entre deux écritures, ou entre une lecture et la lecture ou l'écriture la suivant. D'autre part, des modèles où la relaxation est sur l'exigence de l'atomicité existent. Ils permettent de retourner la valeur modifiée par un autre processeur avant que toutes les copies de cache ne soient invalidées ou mises à jour, i.e, avant que l'écriture ne soit visible à tous les processeurs. Enfin, des modèles relaxant les exigences sur l'ordre du programme et l'atomicité des écritures ont été proposés [Smith, 1982]. Ces modèles permettent à un processeur de lire la valeur qu'il a modifiée avant qu'elle soit visible aux autres processeurs, i.e, avant que les opérations d'invalidation ou de mise à jour n'atteignent les autres processeurs.

2.3 La cohérence de cache

Avec l'explosion de la taille de la mémoire et le dépassement des vitesses d'accès mémoire par les vitesses des processeurs, la mémoire est devenue rapidement un goulot d'étranglement. La latence d'accès à la mémoire dépend fortement de sa taille : plus la taille est grande, plus l'accès est lent [Goodman, 1983a]. En effet, les mémoires sont d'autant plus petites et rapides qu'elles sont proches du processeur. L'intégration des caches dans certains composants améliore la performance et peut réduire la consommation d'énergie.
Les caches sont des mémoires permettant un accès plus rapide aux données les plus récentes. Ils sont organisés en lignes de données, représentant l'unité de transfert entre la mémoire et les caches. Les requêtes d'accès aux données sont adressées de manière hiérarchique à la mémoire. Si un cache peut satisfaire la requête, on dit qu'il y a un succès, sinon il s'agit d'un défaut de cache (la donnée est invalide dans ce cache) [Goodman, 1983b].
Plusieurs types de messages sont échangés entre les processeurs, la hiérarchie de cache et les mémoires, ainsi que les I/Os. La figure 2.11 illustre le mécanisme de communication entre les caches. Les messages envoyés en direction de l'interface externe sont des messages sortants et les messages arrivant en direction du processeur sont des messages entrants. Ces messages sont souvent stockés entre les niveaux de cache dans le but d'améliorer les performances. Des canaux séparés pour les requêtes et les réponses peuvent être mettre mis en place.

2.3.1 Le problème de cohérence de cache

Les architectures multiprocesseurs peuvent comporter plusieurs niveaux caches. A chaque processeur est associé un cache de niveau $L1$. Une même donnée peut se trouver dans plusieurs caches simultanément et peut donc être modifiée par plus d'un processeur. La présence de copies multiples nécessite un mécanisme de gestion pour assurer l'accès simultané à la même valeur d'une ligne de cache présente dans plusieurs caches [Censier and Feautrier, 1978]. Pour illustrer ce problème, on considère un exemple de 2 cœurs avec chacun un cache privé $L1$. Si le cœur 0 écrit dans le bloc B en mettant à jour son cache seulement, les lectures de cœur 1 retourneront l'ancienne valeur jusqu'à la prochaine mise à jour de la mémoire. Cette incohérence provoque un comportement incorrect du système.

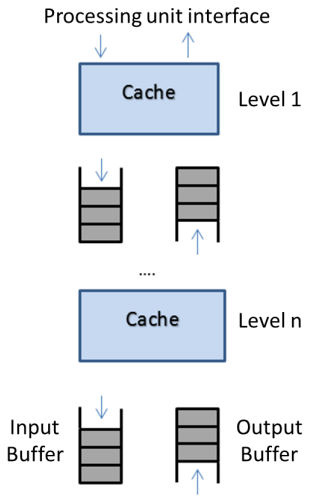

FIGURE 2.11 – Mécanismes de communication entre les caches.

2.3.2 Les protocoles de cohérence de cache et mécanismes de leur gestion

Avec l'apparition des architectures multiprocesseurs à mémoire partagée, de nombreux mécanismes de cohérence de cache ont été proposés.

— La gestion de cohérence de cache par consultation générale (*snooping-based protocols*) : est un mécanisme basé sur l'utilisation de la connexion physique préexistante reliant les nœuds de calcul et la mémoire [Goodman, 1983b]. Les requêtes sont envoyées à tous les processeurs du système. Un processeur répond s'il est impliqué dans cette transaction. Par exemple, si une requête d'écriture est diffusée sur le réseau, tous les caches qui détiennent une copie valide, l'invalident. Bien que ce soit un mécanisme relativement simple à mettre en œuvre, le partage du réseau peut créer un goulot d'étranglement et altérer les performances. Plusieurs méthodes d'amélioration ont été proposées pour réduire la charge sur le réseau [Correale Jr, 1991b] ;

— La gestion de cohérence par répertoire : l'idée est de maintenir le statut des lignes de cache de tous les processeurs du système dans une structure décrite sous forme d'un répertoire (*directory*). Quand un processeur a un défaut de cache sur un bloc, il consulte l'entrée du répertoire correspondant à ce bloc via des transactions envoyées sur le réseau, localise les copies valides de la ligne de cache et détermine les actions à prendre. La communication pour récupérer les copies valides est réalisée en envoyant des transactions additionnelles sur le réseau. Les changements effectués sur les états des lignes de cache sont communiqués à l'entrée du répertoire de ce bloc de sorte que le répertoire soit à jour [Agarwal et al., 1988].

Un protocole de cohérence de cache est défini comme un ensemble de règles régissant les changements d'états de caches. Les états représentent un droit d'accès aux données en indiquant si une donnée est valide ou non. Divers schémas de gestion de lignes de caches ont été proposés. Il existe deux grandes familles de protocoles de cohérence :

Les protocoles *Write-Through*

Cette technique consiste à mettre à jour la mémoire après toute opération d'écriture. Tout accès en écriture à une donnée nécessite l'invalidation des autres copies disponibles. Par conséquent, tous les processeurs ont donc en permanence des données à jour [Vashi and Strickland, 1996]. Le protocole MI ($Modified - Invalid$) illustre cette technique, qui est un schéma basique, où l'état *invalid* indique que la ligne est invalide ou non présente dans le cache et l'état $Modified$ indique qu'une ligne est présente et valide dans le cache.

Les protocoles *Write-Back*

Cette approche permet plusieurs accès simultanés en lecture seule d'une donnée. En revanche, un seul accès en écriture est possible. Lorsqu'un processeur veut modifier une donnée, il doit envoyer un message aux autres processeurs pour l'invalider et attendre leurs retours. Les mises à jour sont propagées uniquement quand un processus accède à une donnée invalide, ce qui évite les transmissions inutiles [Ulfsnes, 2013]. Les protocoles de type *write-back* les plus communs sont détaillés dans ce qui suit :

- MSI ($Modified - Shared - Invalid$) : est une extension du protocole MI avec un état additionnel S. Une ligne est à l'état M si c'est l'unique copie modifiée de la mémoire. Une ligne à l'état S indique que sa valeur n'est pas modifiée par rapport à la mémoire. Elle peut être valide dans d'autres caches ;
- MESI ($Modified - Exclusive - Shared - Invalid$) : Un état additionnel est introduit. L'état E indique que la ligne est présente et valide exclusivement dans le cache. La ligne a la même valeur en mémoire ;
- MEOSI ($Modified - Exclusive - Owned - Shared - Invalid$) : en plus des états cités précédemment, une ligne de cache peut être à l'état O. La ligne de cache contient alors une donnée partagée par plusieurs processeurs, mais la valeur n'est pas la même qu'en mémoire. Cet état a été ajouté afin de réduire le nombre de $write - backs$ non nécessaires ;
- MESIF ($Modified - Exclusive - Shared - Invalid - Forward$) : il est utilisé dans les protocoles de *snooping* afin d'éviter la réception de réponses redondantes de tous les caches partageant une ligne de cache. Seul le cache ayant la ligne de cache à l'état F doit envoyer la réponse et la donnée si nécessaire.

2.3.3 Les protocoles AMBA ACE

Certains *designs* de cache associent plus d'états aux lignes de caches dans le but d'optimiser certaines transitions.

Les systèmes à mémoire partagée les plus récents ont été étendus avec des accélérateurs pouvant avoir des caches. C'est le cas dans les derniers appareils de téléphones mobiles. Cette extension de la cohérence nécessite un protocole qui maintient la cohérence des caches de tous les composants hétérogènes du système. Assurer la cohérence des données au niveau système s'avère l'un des défis majeurs auxquels sont confrontés les architectes des multiprocesseurs à base de mémoire partagée. Afin de gérer la cohérence de cache et en raison de la complexité croissante des logiciels, un support matériel pour la cohérence de cache a été proposé. Comme pour tout autre système, ARM se propose de gèrer la

cohérence de cache par le *hardware*. Dans sa quatrième génération de bus intitulée AMBA, le protocole AMBA ACE a été conçu. Il est alors devenu un standard industriel pour la gestion de la cohérence de cache et plus particulièrement, dans les systèmes hétérogènes sur puces [Miao et al., 2006].

La spécification AMBA ACE a introduit les extensions de cohérence au protocole AXI (*Advanced eXtensible Interface*). Ce protocole permet une flexibilité dans l'implémentation des architectures à base de réseaux d'interconnexion et assure la consistance mémoire et la cohérence de cache entre des composants ayant différentes caractéristiques. Le protocole AXI définit la communication entre deux composants au niveau interface. Une unique interface est définie entre un processeur et le réseau d'interconnexion, un coprocesseur et le réseau d'interconnexion et les mémoires et le réseau d'inteconnexion. Dans les architectures basées sur ce protocole, plusieurs types de canaux de communication existent. La spécification AXI décrit la structure de chaque type de canal et les paramètres qui leur sont associés, comme la taille des canaux de données et des canaux de réponses. L'extension ACE a par ailleurs introduit des modifications au niveau de l'ordonnancement des transactions et des canaux additionnels ont été ajoutés. Toutes ces caractéristiques seront détaillées dans le chapitre 3.
Dans cette thèse, nous étudions l'implémentation du protocole de cohérence AMBA ACE sur l'architecture extensible que nous avons conçue.

2.4 Conclusion

Dans ce chapitre, nous avoir présenté le contexte de la thèse en décrivant une des classes les plus importantes des machines parallèles qui est la classe des multiprocesseurs à mémoire partagée. Nous avons ensuite décrit la consistance mémoire et la cohérence de cache dans ces systèmes. Dans une architecture à mémoire partagée, le choix du protocole de consistance mémoire et de cohérence de cache est important. Dans les systèmes les plus récents, la tâche de gestion de la cohérence des données au niveau du système est particulièrement complexe vu l'hétérogénéité des composants. C'est pourquoi, de plus en plus de protocoles sont proposés pour définir les règles de communication entre les composants hétérogènes. Nous avons introduit le protocole AMBA ACE que nous avons retenu dans cette thèse pour la gestion de la consistance mémoire et la cohérence de cache dans l'architecture générique proposée.

Chapitre 3

Systèmes hybrides à base de multiprocesseurs ARM

Dans ce chapitre, nous décrivons les systèmes multiprocessseurs à base de réseaux *CCN* (*CoreLink Cache Coherent Networks*) proposés par ARM. Nous définissons ensuite la cohérence de cache dans ces systèmes hybrides en décrivant les principaux éléments de la spécification du protocole AMBA ACE qui a été proposé dans le but de gérer la cohérence de cache et d'assurer la consistance mémoire. Le protocole de mémoire partagée AMBA ACE, ainsi que l'architecture associée « *CoreLink CCN* » sont retenus comme solution de départ dans notre approche.

3.1 Les systèmes de ARM

Aujourd'hui, ARM est devenu un acteur dominant dans le domaine de l'informatique embarquée. Il est surtout connu pour ses systèmes sur puces (*SoCs*) regroupant sur une même puce plusieurs composants tels que des microprocesseurs, des processeurs graphiques et des *DSP*. Les architectures ARM existent depuis l'introduction de ARM1 qui a été conçu par *Acorn Computers* à Cambridge en 1985. Avec la création de ARM en 1990, l'objectif principal a été de concevoir des processeurs puissants à faible consommation. Les architectures ont évolué au fil des années. La réalisation de l'architecture ARM7 en 2004 est considérée comme l'une des étapes les plus importantes dans cette évolution. ARM est spécialisé dans la conception de propriétés intellectuelles (*Intellectual Properties*). Les cœurs des processeurs ARM sont très présents dans les systèmes embarqués. Dans ce qui suit, nous décrivons les principales évolutions des produits ARM [Matthews, 1999].

3.1.1 Les processeurs ARM

Dotés d'une architecture relativement simple avec une faible consommation, les processeurs ARM sont massivement utilisés dans les appareils électroniques destinés au grand public pour une utilisation personnelle. Cela inclut les téléphones mobiles, les tablettes numériques, les lecteurs multimédia, etc.
Les processeurs ARM sont basés sur une architecture RISC (*Reduced Instruction Set Computer*) développée par Advanced RISC Machines (ARM). ARM a conçu des processeurs multicœurs RISC de 32 et 64 bits dans le but : (i) d'exécuter des instructions en nombre réduit de sorte à opérer à grande vitesse, et ce en exécutant des millions d'instruc-

tions par seconde (MIPS) et (ii) de réduire la consommation en énergie. Grâce au nombre réduit d'instructions, le nombre de transistors utilisés dans leur fabrication est réduit, ce qui a facilité leur intégration à des composants de très petites tailles.
Le premier processeur commercialisé de ARM a été le ARM2 (sans cache) suivi de ARM 3 qui est la nouvelle version avec cache. Il y a eu ensuite l'apparition des cœurs ARM6 et ARM7 à partir desquels ARM a commencé à viser le marché de l'embarqué [Jeff, 2012].
Afin de satisfaire le besoin croissant survenu avec l'explosion du marché de la téléphonie mobile, ARM a conçu de plus en plus de SoCs avec comme premier modèle de licence IP, les microprocesseurs. En travaillant sur la conception d'un cœur pouvant s'appliquer à différentes technologies, ARM a proposé en 2001, ARM926EJ-S qui est complètement synthétisable avec 5 étages de pipelines, une MMU (*Memory Management Unit*) intégrée et un support *hardware* des accélérations Java et de certaines opérations DSP. De nouvelles technologies sont ensuite apparues dans ARM 10 et ARM 11.
Afin de diversifier les services et les solutions proposées, ARM a conçu plusieurs familles de processeurs que nous citons ci-dessous. Ces familles de processeurs sont destinées à des applications diverses.

Cortex-A-Series

Basés sur l'architecture ARMv7-A, il s'agit de processeurs à haute performance conçus pour des systèmes d'exploitation complexes tels que Linux, Android, Chrome OS et autres. Ce sont principalement des processeurs 32 bits exceptés les nouveaux Cortex-A72, Cortex-A57 et Cortex-A53 qui délivrent une performance en 32 et 64 bits utilisés pour les téléphones $next - generation$, les réseaux et les serveurs. Ces processeurs ont de 1 à 4 cœurs avec une capacité à intégrer une MMU et des FPU. Les processeurs Cortex-A5, Cortex-A7, Cortex-A9, Cortex-A15 et Cortex-A17 sont basés sur l'architecture ARMv7. Les nouveaux processeurs Cortex-A72, Cortex-A57 et Cortex-A53 étendent les performances de la famille Cortex-A series en exécutant des instructions 64 bits tout en maintenant la compatibilité avec les instructions à 32 bits.
En 2011, la technologie $big.LITTLE$ a été introduite à la série Cortex-A pour améliorer les performances en combinant un processeur haute performance avec un processeur efficace en énergie. Les tâches sont migrées entre les deux processeurs en les exécutant sur le processeur le plus adapté. Les configurations considérées sont : Cortex-A7 combiné avec Cortex-A15/Cortex-A17 et Cortex-A53/Cortex-A35 combinés avec Cortex-A72/Cortex-A57. Le Cortex-A53 (big) peut être combiné avec Cortex-A35 ($LITTLE$) [Penton and Jalloq, 2006].
Les caractéristiques principales de ces processeurs sont :

- La mémoire virtuelle : tous les cœurs Cortex-A contiennent une MMU permettant une conversion d'adresses physiques-virtuelles avec un adressage à 40 bits dans le Cortex-A15 et Cortex-A7. L'architecture supportée est nommée $VMSAv7$;
- Plusieurs niveaux de cache : l'architecture permet de supporter jusqu'à 8 niveaux de caches. Cependant, seuls 3 niveaux de caches existent dans les systèmes conçus ;
- Support du multicœurs : à l'exception du Cortex-A8, tous les Cortex-A ont une configuration multicœurs ;
- Sécurité $TrustZone$: une extension introduite à tous les Cortex-A pour créer deux machines virtuelles s'exécutant sur un seul processeur avec un partitionnement entre les deux. Elle est conçue pour des applications sécurisées comme le e-paiement ;

— La virtualisation : une solution de virtualisation est supportée avec un hyperviseur permettant de créer des machines virtuelles et des systèmes d'exploitations multiples ;
— NEON (*Advanced SIMD Extension*) : une extension optionnelle destinée aux programmes multimédia haute performance en fournissant une accélération aux algorithmes de compression des données, traitement d'images, etc ;
— *Floating point Unit* : FPU est une extension optionnelle qui permettant de supporter la virgule flottante à double précision.

Un large choix d'applications est possible avec cette famille de processeurs. On peut citer : les smartphones, les tablettes, les jeux vidéos, les imprimantes, les serveurs, etc.

Cortex-R-Series

Il s'agit d'une famille de processeurs temps réel visant à délivrer un comportement déterministe pour des applications sensibles à la puissance (*power sensitive*) et ce en temps réel [Wenger et al., 2013]. Ils exécutent un système d'exploitation temps réel (RTOS pour *Real-Time Operating System*) et contiennent une MPU (*Memory Protection Unit*) et peuvent avoir jusqu'à 4 cœurs.
Parmi ses applications, on peut citer : les systèmes de contrôle automobile, les contrôleurs de stockage en masse, les imprimantes etc.

Cortex-M-Series

Il s'agit de processeurs de 32 bits destinés aux applications de calcul embarquées. Ils sont délivrés en *black box* avec des applications pré-chargées et ont une capacité limitée à étendre les fonctionnalités *hardware*.
Les principales applications pour cette classe de processeurs sont : les micro-contrôleurs, les compteurs intelligents (*smart meters*), internet des objets connectés (*internet of things*).
La figure 3.1 donne une description d'un exemple de processeur pour chaque série décrite ci-dessus [ARM, 2016].

Les différents modèles de processeurs cités précédemment peuvent être implémentés dans des *clusters* allant jusqu'à 4 cœurs.

3.1.2 Les systèmes intégrant les processeurs ARM

Les systèmes à base de processeurs ARM intègrent des processeurs appartenant à une des familles citées ci-dessus et des composants additionnels. Afin de garantir l'interportabilité et de fournir un modèle de programmation commun à différentes implémentations, ARM a défini des spécifications pour ses architectures décrivant le fonctionnement des produits ARM. Pour différents systèmes basés sur un même type de processeur ARM, différentes configurations peuvent être définies à l'implémentation. Par exemple, un système générique peut inclure : un contrôleur de cache $L2$, un contrôleur mémoire statique et/ou dynamique, un réseau d'interconnexion, un contrôleur d'interruptions, un *timer* et des interfaces externes de bus.
Avec l'intégration massive des composants à un système, l'interaction entre eux est devenue de plus en plus complexe. Il est important de gérer efficacement la communication

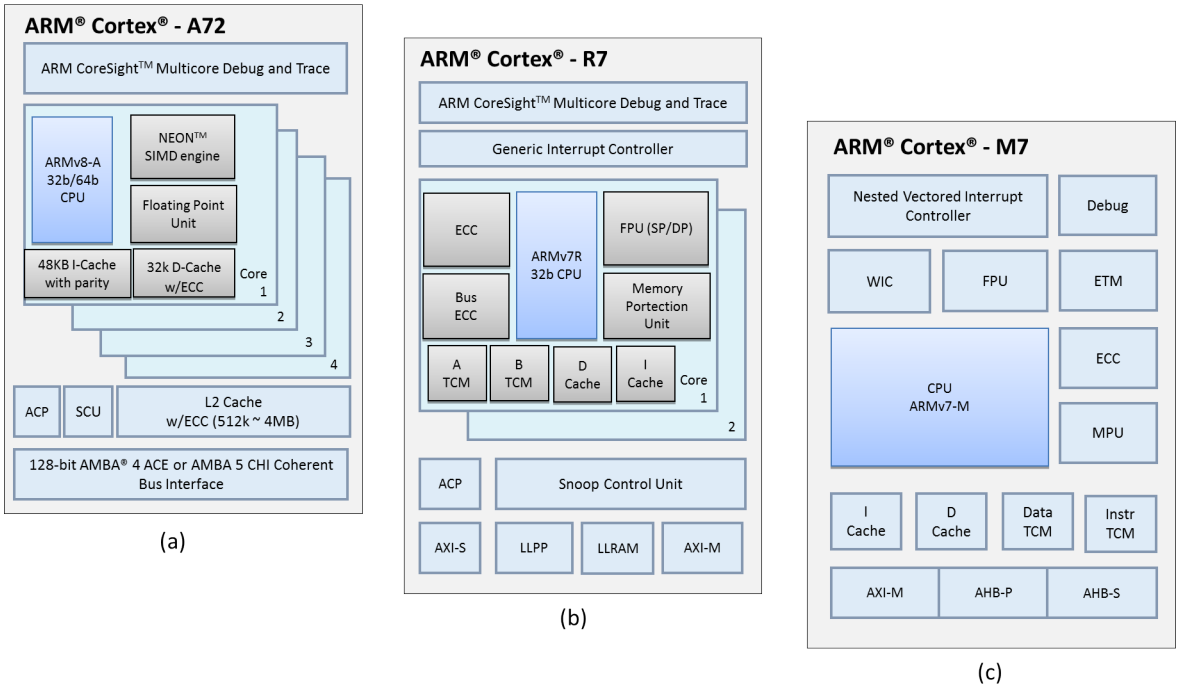

FIGURE 3.1 – Description des processeurs (a) Cortex-A72, (b) Cortex-R7 et Cortex-M7.

pour avoir un système performant. Ils existent divers systèmes fabriqués par différentes entreprises. De nouveaux modèles sortent tous les ans afin de suivre l'évolution du marché. ARM a conçu une famille de systèmes d'interconnexion appelée ARM *CoreLink* intégrant des processeurs Cortex et Mali dans le but de fournir les meilleures performances en termes de latence et flux de données à haut débit. Avec cette famille *CoreLink*, ARM fournit aux designers, les composants et la méthodologie pour construire des systèmes tout en maximisant l'efficacité de stockage et de flux des données. Dans ce qui suit, nous citons les 3 familles de systèmes d'interconnexion de ARM.

Interconnexion des composants

— *CoreLink CCN Cache Coherent Networks* : destinés aux applications infrastructures, ils sont optimisés pour les processeurs ARM-Cortex et supportent la dernière architecture ARMv8-A incluant le Cortex-A72 et le Cortex-A53. Le nombre de cœurs supportés varie de 1 à 48 cœurs. Un cache configurable *L3* peut être intégré pour permettre aux I/O et accélérateurs d'allouer de la mémoire cache, réduisant ainsi les accès à la mémoire externe. Les CCN sont scalables pour des applications infrastructures multiples. La famille CCN est optimisée pour les processeurs Cortex-A72 et Cortex-A53, les contrôleurs mémoires et AMBA 4 ACE/ACE-LITE et AXI4 pour les I/O. La figure 3.2 représente un tableau récapitulatif des caractéristiques des différents *CCNs* [ARM, 2016]. Notre travail est basé sur cette famille d'interconnexion. Son architecture sera détaillée dans le chapitre suivant 4 ;

— *CoreLink* CCI *Cache Coherent Interconnects* : ils supportent les processeurs multicœurs incluant Cortex-A53 et Cortex-A57 et les GPU Mali. Dans le modèle *CoreLink* CCI-400, jusqu'à 2 *clusters* de 4 cœurs chacun sont supportés. Dans le modèle *CoreLink* CCI-500, 4 *clusters* incluant big.LITTLE et des accélérateurs cohérents

Feature	CCN-502	CCN-504	CCN-508	CCN-512
Key Benefits	Smallest CCN interconnect, optional L3	Highest performance 4 cluster interconnect	8 cluster interconnect, high IO bandwidth	12 cluster interconnect, highest compute performance
L3 Cache Support	0 to 8 MB	1 to 16 MB	1 to 32 MB	1 to 32 MB
Processor Clusters	1-4 Cortex-A53/ Cortex-A57	1-4 Cortex- A53/ Cortex- A57	1-8 Cortex- A53/ Cortex- A57	1-12 Cortex-A53/ Cortex-A57
IO (AXI4/ACE-Lite)	Up to 9	Up to 18	Up to 24	Up to 24
DDR3/DDR4	1 to 4 channels with DMC-520	1 to 2 channels with DMC-520	1 to 4 channels with DMC-520	1 to 4 channels with DMC-520

FIGURE 3.2 – Caractéristiques des *CoreLink CCN*.

sont supportés ;

— *CoreLink* NIC *Network Interconnects* : il s'agit d'interfaces réseau optimisant la latence et la consommation. Ils sont configurables avec une connectivité hiérarchique pour AMBA et destinés à plusieurs applications comme les micro-contrôleurs basés sur Cortex-M et les SoCs basés sur Cortex-A.

Afin de gérer l'interconnexion des blocs fonctionnels, des protocoles standards définissant des interfaces communes ont été proposées par ARM comme le protocole AMBA (*Advanced Microcontoller Bus Architecture*) qui est une spécification d'interconnexion facilitant le développement des systèmes multiprocesseurs avec un nombre important de contrôleurs et de périphériques. Il réunit les interfaces CHI, ACE, AXI, AHB, APB et ASB. Dans la section suivante, nous décrivons l'évolution du l'architecture AMBA.

L'architecture de bus AMBA

AMBA a une architecture de bus hiérarchique. Il est organisé en plusieurs bus système et bus périphériques connectés via un *bridge*. La spécification AMBA définit le protocole de communication entre les composants sur chaque type de bus. On distingue plusieurs types de bus AMBA :

— ASB (*Advanced system Bus*) : Il s'agit de la première génération du bus AMBA présentée en 1996 ;

— ATB (*Advanced Trace Bus*) : est le bus de trace avancé destiné à offrir une solution de déboguage et traçage sur les puces appelées CoreSight ;

— APB (*Advanced Peripheral Bus*) : APB est un bus statique qui fournit un adressage simple. Il est utilisé pour connecter des périphériques à faible vitesse et consommation ;

— AHB ($Advanced\ High-performance\ Bus$) : Il s'agit du bus AMBA haute performance. Il fournit une communication à haut débit entre les processeurs (ARM, MIPS, ADP 320xx, etc.), les accélérateurs/périphériques à haute performance (MPEG, LCD, etc), SRAM *on-chip*, l'interface mémoire externe et le bridge APB. AHB se compose de $master-AHB$, de $slave-AHB$, d'un arbitre et d'un décodeur ;

— AXI ($Advanced\ eXtensible\ Interface$) : est basé sur le concept de connexion point à point. Il peut connecter jusqu'à 100 *masters* (processeurs/coprocesseurs) et *slaves*

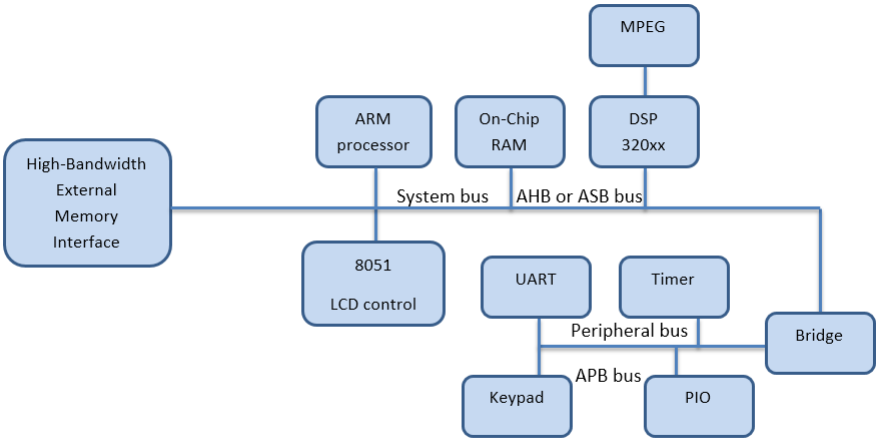

FIGURE 3.3 – Exemple d'un système avec une architecture de bus basée sur AMBA.

(mémoires).
— ACE (AXI *Coherency Extensions*) : est utilisé dans les systèmes big.LITTLE. Nous le détaillons dans la section suivante.
— CHI (*Coherent Hub Interface*) : fournit la plus haute performance. Il est utilisé dans les serveurs et les réseaux.

La figure 3.3 illustre un exemple d'architecture basée sur les bus AMBA.

3.2 La cohérence de cache dans les systèmes à base de réseaux d'interconnexion CCN

Les systèmes hybrides consistent en un ensemble de blocs de processeurs, coprocesseurs, mémoires et I/Os. Certains composants hétérogènes accèdent à une mémoire partagée. Dans le but de réduire le temps d'accès à cette mémoire partagée, certains composants peuvent être dotés de caches locaux contenant des copies locales des lignes mémoire. Une telle configuration engendre un problème de cohérence de cache au niveau du système.
Le maintien de la cohérence de cache et la consistance mémoire peut être assuré par des solutions logicielles ou matérielles. Le partage des ressources comme la mémoire nécessite une interconnexion variant d'un simple bus à un réseau de bus complexe acheminant les messages entre les différents blocs du système. Dans ce contexte, ARM a proposé plusieurs générations de bus AMBA comme présenté dans la 3.1.2 [ARM, 2016].
Dans le cadre de la conception des systèmes de calcul efficaces et flexibles, ARM a proposé une famille de réseaux d'interconnexion appelée *CoreLink CCN Cache Coherent Networks* [ARM, 2015]. Elle connait un large nombre d'applications comme les serveurs et les *data centers*. La cohérence de cache est assurée par le protocole ACE. Les différents composants supportent le protocole $ACE-master$ pour les composants avec caches (les processeurs) et le protocole $ACE-Lite\ master$ pour les composants sans caches (coprocesseurs). Afin de mieux gérer la cohérence de cache, un composant appelé *Snoop filter* est utilisé. Il permet d'optimiser les temps de réponse aux requêtes.
Dans cette famille de *CoreLink*, les derniers processeurs 64 bits $ARMv8$ incluant Cortex-A57 et Cortex-A53 à deux niveaux de cache sont utilisés. Le nombre de cœurs peut aller

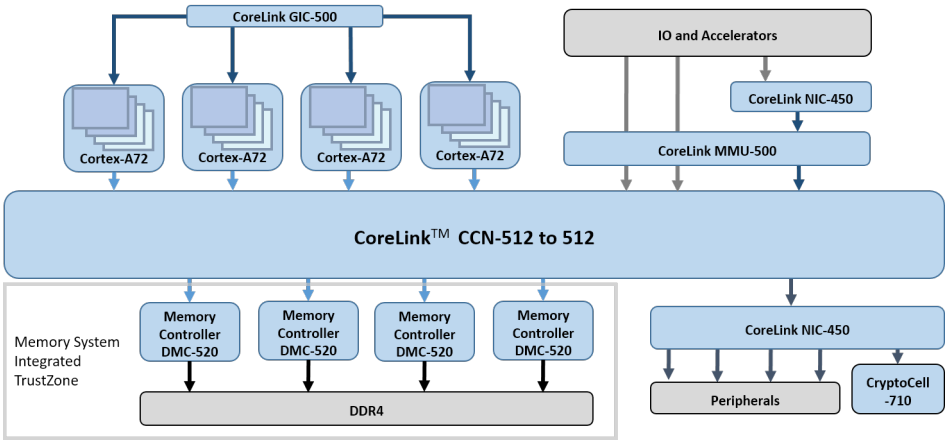

FIGURE 3.4 – Architecture générale d'un *CoreLink CCN*.

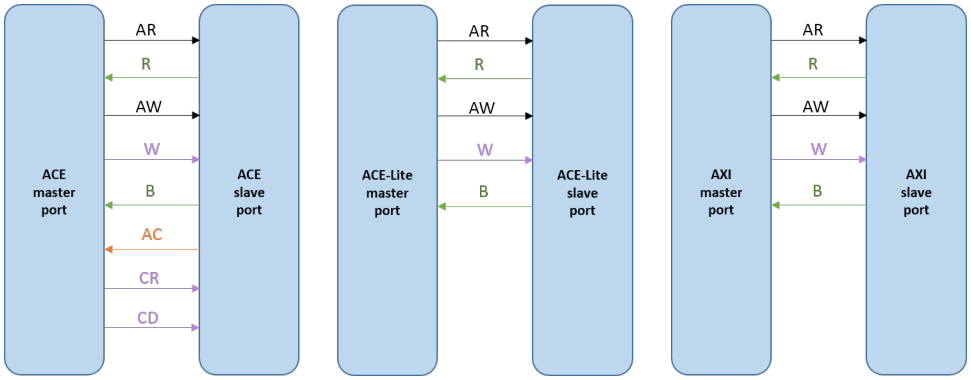

FIGURE 3.5 – Les canaux AXI et ACE.

jusqu'à 48. Un troisième niveau de cache L3 de taille allant jusqu'à 32MB peut être intégré au niveau de l'interconnexion.
La figure 3.4 donne un aperçu général de l'architecture des *CCNs*. Les composants communiquent via des ports. Chaque port consiste en plusieurs canaux. On distingue des canaux pour la lecture, des canaux pour l'écriture et des canaux pour le *snooping*. Ils sont détaillés dans ce qui suit et illustrés dans la figure 3.5.

- *Address read channel* (AR) : utilisé pour envoyer les requêtes de lecture ;
- *Data read channel* (R) : utilisé pour envoyer la donnée ;
- *Address write channel* (AW) : utilisé pour envoyer les requêtes d'écriture ;
- *Data write channel* (W) : utilisé pour envoyer les données à écrire ;
- *Write response channel* (B) : utilisé pour signaler la fin de l'opération d'écriture ;
- *Address coherency channel* (AC) : utilisé pour envoyer les requêtes de *snooping* ;
- *Coherency response channel* (CR) : utilisé pour répondre aux requêtes de *snooping* en indiquant si le transfert de données aura lieu ;
- *Coherency data channel* (CD) : utilisé pour envoyer les données au *CoreLink*.

Exemple : *Cache Coherent Networks CCN-512*

Destiné principalement aux serveurs haute performance et aux centres de calcul (*data centers*), le *CoreLink CCN-512* supporte la plus grande configuration allant jusqu'à 12

clusters de 48 cœurs. Il s'agit d'un système d'interconnexion à haut débit approchant 1.8 TB/sec en utilisant des mémoires DDR4 à une fréquence d'horloge de 3.2GHz. Cette configuration a quatre contrôleurs mémoire DCM-520 supportant des mémoires DDR4. Il supporte le protocole de cohérence AMBA ACE et AMBA 5 CHI et peut intégrer un niveau de cache L3 de taille allant de 1 à 32 MB.
L'architecture du *CoreLink CCN-512* est illustrée dans la figure 3.6 [ARM, 2016]. Dans cette thèse, nous étendons le CCN-512 à une configuration plus large et vérifions le protocole ACE mappé sur cette nouvelle architecture.

3.3 Le protocole de cohérence de cache AMBA ACE

Le protocole ACE étend le protocole AXI4 en fournissant un support pour la cohérence de cache au niveau système. Il est réalisé avec un modèle à 5 états de ligne de cache. Il existe une machine d'états FSM déterminant les actions générées par la réception d'une requête ou d'une réponse donnée selon l'état actuel de la ligne de cache.
Ce protocole assure que tous les *masters* (processeurs et coprocesseurs) observent la valeur correcte de la donnée en forçant l'existence d'une seule copie lors de sa modification. Après la fin de l'écriture, les autres *masters* peuvent obtenir une nouvelle copie de la donnée pour leur cache local. La mémoire n'est pas obligatoirement mise à jour à chaque écriture. Grâce à ce protocole, les composants peuvent déterminer les actions à effectuer sur une ligne de cache donnée. Si elle est unique, sa modification ne nécessite pas la notification des autres composants. Dans le cas contraire, tous les caches partageant cette ligne de cache seront notifiés de cette modification.

3.3.1 Les états

La spécification ACE distingue 5 états de ligne de cache :
— *Invalid* : une ligne de cache est à l'état *Invalid* si elle n'est pas présente dans ce cache ;
— *Unique* : une ligne de cache est *Unique* si toutes les autres copies de la même ligne mémoire sont invalides. La ligne de cache réside dans un seul cache ;
— *Shared* : une ligne de cache est *Shared* si elle réside dans plus d'un cache ;
— *Dirty* : une ligne de cache est en *Dirty* si sa valeur est différente de celle en mémoire. Toutes les copies dans les caches ont la même valeur. Un seul cache est responsable de réécrire la donnée en mémoire ;
— *Clean* : une ligne de cache est en *Clean* si sa valeur est la même que celle en mémoire. Le cache n'a pas la responsabilité de mettre à jour la mémoire.
La figure 3.7 montre les 5 états de cache du modèle ACE.

3.3.2 Les interfaces

Afin de supporter les transactions de cohérence, le protocole AXI a été étendu. Deux types d'interfaces sont définies : les interfaces $ACE-master$ utilisés pour les composants avec cache et les interfaces ACE-Lite pour les composants sans caches. Au niveau du *CoreLink*, un port *ACE slave* est utilisé pour être connecté à un port *ACE master* et un port *AXI master* pour être connecté à la mémoire.

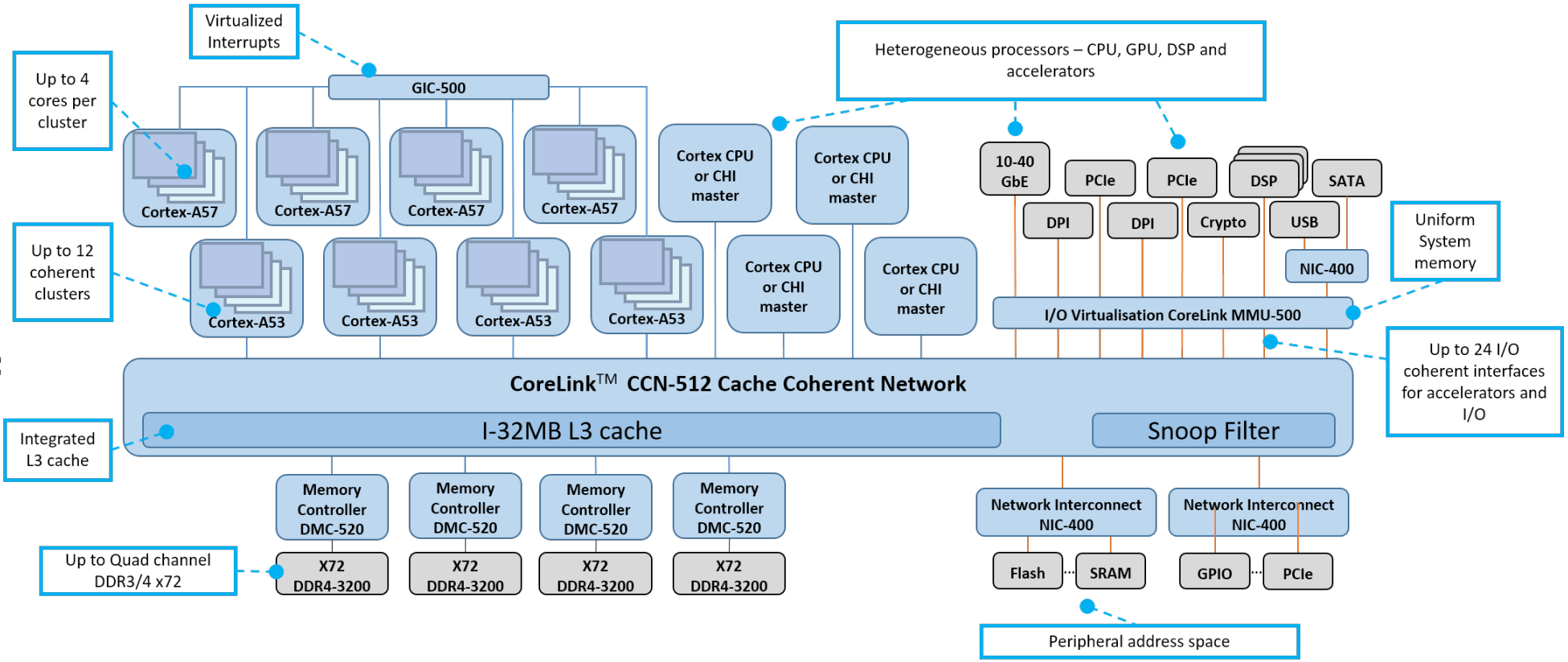

FIGURE 3.6 – Architecture du *CoreLink CCN-512*.

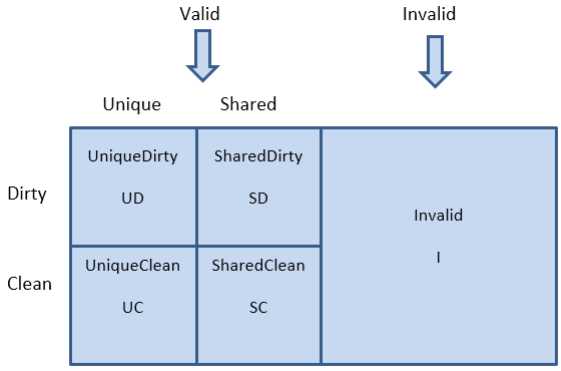

FIGURE 3.7 – Le modèle d'états de cache de ACE.

3.3.3 Rôle du *snoop filter*

L'implémentation la plus simple de la cohérence de cache consiste à diffuser un *snoop* à tous les caches du système afin de localiser la donnée partagée. Quand un cache reçoit une requête de *snooping*, il détermine s'il détient une copie de la donnée et envoie la réponse adéquate. Dans la majorité des applications, le taux de succès de cache lors d'un *snooping* est faible. Plusieurs opérations de *snooping* ne sont pas nécessaires. La solution est d'effectuer des opérations de *snooping* d'une manière optimisée en intégrant un *snoop filter* à l'interconnexion. Ce composant permet de maintenir un répertoire pour le contenu du cache de chaque processeur pour éliminer ainsi le besoin de diffuser le *snoop* à tous les caches du système.
Le principe du *snoop filter* est comme suit :

- L'adresse en mémoire de chaque ligne présentée dans un cache est rangée dans un répertoire. Lors d'un accès mémoire, le *snoop filter* permet de savoir si une donnée est présente dans l'un des caches en la localisant ;
- L'information sur le *master* responsable de la mise à jour de la mémoire est disponible dans le *snoop filter* ;
- (i) *hit* : une copie de la donnée est présente dans les caches, la liste des *caches* ayant la donnée est retournée. Le *snoop filter* répondra aux requêtes de cohérence après avoir reçu la réponse à la requête de *snooping* envoyée aux caches, (ii) *miss* : la donnée est dans la mémoire partagée.

3.3.4 Opération de lecture d'une région de la mémoire partagée

Dans ce qui suit, nous donnons l'exemple d'une opération de lecture depuis un emplacement mémoire partagé, dans le cas où le composant master M ne détient pas une copie de cet emplacement dans son cache local. La procédure est la suivante :

- Le *master* M émet une requête de lecture sur le canal d'adresse de lecture AR à travers son cache local ;
- Le *CoreLink* détermine si un autre cache détient une copie de cet emplacement mémoire pour envoyer l'adresse sur le canal d'adresse de *snooping* AC ;
- Un des deux cas se produisent : (i)si une copie de cet emplacement mémoire est

présente dans l'un des caches du système, le cache *snoopé* répond sur le canal de réponse au *snooping* CR et fournit la donnée sur le canal de données CD. (ii) si aucun cache ne détient une copie de l'emplacement mémoire, le *CoreLink* initie une transaction à la mémoire principale. La donnée est ensuite envoyée au *master* M sur le canal des données R ;
— Le *master* M indique que la transaction est terminée en utilisant le signal $RACK$.

3.3.5 Opération d'écriture dans une région de la mémoire partagée

Quand un *master* écrit dans une ligne de cache ou un emplacement mémoire, il invalide toutes les autres copies de la ligne de cache afin de garantir la présence d'une copie unique dans le système. La nouvelle valeur de la ligne de cache est propagée aux autres caches après qu'ils aient effectué des opérations de lecture. Dans le cas où un *master* M veut effectuer une opération d'écriture et a déjà une copie de la ligne de cache. la procédure est comme suit :

— Le *master* M demande une copie unique de la ligne de cache en envoyant une requête $CleanUnique$ sur le canal AR. Toutes les autres copies de la ligne de cache sont alors supprimées et si une copie $Dirty$ existe, elle est réécrite en mémoire ;
— Le *CoreLink* passe la transaction aux autres caches sur le canal AC qui répondent sur le canal CR en indiquant la suppression de la ligne avec succès. Si la ligne de cache est $Dirty$, elle peut être réécrite en mémoire ;
— Si une donnée $Dirty$ existe, un des *masters* fournit la ligne de cache au *CoreLink* en utilisant le canal CD qui construit la transaction pour écrire la donnée $Dirty$ en mémoire ;
— Une réponse est fournie au *master* M sur le canal R. Aucun transfert de données n'est fait ;
— Le *master* M effectue le *store* et utilise le signal $RACK$ pour indiquer que la transaction est terminée.

Dans le cas où un *master* M veut effectuer une opération d'écriture partielle d'une ligne de cache, la procédure est comme suit :

— Le *master* M obtient une forme de *pre-store* de la ligne de cache et demande que toutes les autres copies soient supprimées en envoyant une transaction $ReadUnique$ sur le canal AR ;
— Le *CoreLink* passe la transaction aux autres caches sur le canal AC ;
— Les autres caches répondent sur le canal CR en fournissent au *CoreLink* la donnée sur le canal CD ;
— Si aucun *master* ne détient une copie de ligne de cache, une lecture de la mémoire partagée est effectuée ;
— Le *CoreLink* envoie la donnée au *master* M sur le canal R ;
— Le *master* M effectue le *store* et utilise le signal $RACK$ pour indiquer que la transaction est terminée.

3.3.6 Les transactions

Dans la spécification du protocole ACE, il existe : des transactions de non-*snooping* utilisées pour accéder aux emplacements mémoire non partagés, des transactions cohérentes utilisées par ACE et ACE-Lite pour accéder à des emplacements mémoire partagées en envoyant la transaction au *CoreLink* qui initiera la transaction de *snooping* correspondante aux autres *masters* si nécessaire, des transactions de mise à jour de la mémoire utilisées pour mettre à jour la mémoire partagée, des transactions de maintenance de cache utilisées par les composants pour accéder et observer l'effet d'opérations de *load* et *store* sur les caches des autres composants, et des transactions de *snooping* initiées par le *CoreLink* pour accéder aux caches d'un ACE *master* lors du traitement d'une transaction cohérente ou de maintenance de cache.
Nous détaillons dans ce qui suit les transactions classifiées en 5 types :

A) Les transactions de lecture : Dans ce type de transactions, l'adresse est envoyée sur le canal AR, la donnée et la réponse sont retournées sur le canal R. 6 types de transactions de lecture existent.

- *ReadNoSnoop* : *ReadNoSnoop* est une requête de lecture utilisée dans une région mémoire non partagée avec d'autres *masters*. Le *CoreLink* doit toujours lire depuis la mémoire, ou le périphérique approprié ;
- *ReadOnce* : *ReadOnce* est une requête de lecture utilisée dans une région mémoire partagée avec d'autres *masters*. Cette requête est utilisée quand un *snapshot* de la donnée est requis. L'emplacement n'est pas sauvegardé localement dans le cache pour une future utilisation ;
- *ReadClean* : *ReadClean* est une requête de lecture utilisée dans une région mémoire partagée avec d'autres *masters*. Elle garantit de ne pas passer la responsabilité de mise à jour de la mémoire au *master* initiant la transaction. Elle est utilisée par un *master* qui veut obtenir une copie propre de la ligne de cache ;
- *ReadNotSharedDirty* : est une requête de lecture utilisée dans une région mémoire partagée avec d'autres *masters*. Elle est utilisée par un *master* avec cache exécutant une opération *load* et pouvant accepter la ligne de cache dans n'importe quel état sauf l'état *SharedDirty* ;
- *ReadShared* : *ReadShared* est une requête de lecture utilisée dans une région mémoire partagée avec d'autres *masters*. Elle est utilisée par un *master* ayant un cache, exécutant une opération *load* et pouvant accepter la ligne de cache dans n'importe quel état ;
- *ReadUnique* : *ReadUnique* est une requête utilisée dans une région mémoire partagée avec d'autres *masters*. Elle obtient une copie de la donnée tout en assurant que la ligne de cache soit l'unique copie de la donnée. La ligne de cache est obtenue avec l'état unique. Ceci permet au *master* d'effectuer une opération *store*. Elle est utilisée quand un *master* veut effectuer un *store* d'une ligne de cache partielle et qu'il n'a pas de copie de la ligne de cache.

B) Les transactions *Clean* : Elles sont envoyées sur le canal AR. Un transfert sur le canal R retourne la réponse. Aucune donnée n'est retournée pour ce type de transactions.

- *MakeUnique* : *MakeUnique* est seulement utilisée par un *master* effectuant une

opération *store* sur une ligne complète de cache. Elle est retenue à l'état *unique*. Ceci permet au *master* de procéder à l'opération de *store* sans avoir à obtenir une copie de la donnée. Toutes les autres copies de la ligne de cache sont supprimées ;
— *CleanShared* : *CleanShared* est une requête de diffusion d'une opération *Clean* utilisée dans des régions mémoire partagées et non partagées. Elle assure que toutes les copies de la mémoire disponibles dans les caches soient propres. Si le *master* effectuant cette opération de maintenance détient une copie de la ligne de cache en *Dirty*, il doit émettre une transaction de *WriteBack* ou *WriteClean* pour avoir la ligne de cache à l'état *Clean* avant d'émettre la transaction *CleanShared* ;
— *CleanInvalid* : *CleanInvalid* est une requête de diffusion d'une opération de *Clean* et d'invalidation utilisée dans des régions mémoire partagées et non partagées. Elle assure que la mémoire soit mise à jour et qu'aucune copie de l'emplacement mémoire ne soit présente dans les caches. Si le *master* effectuant cette opération de maintenance détient une copie de la ligne de cache en *Dirty*, il doit émettre une requête de *WriteBack* ou *WriteClean* pour ensuite invalider sa ligne de cache avant d'émettre la requête *CleanInvalid*.

C) Les transactions *make* : Elles sont envoyées sur le canal AR, un transfert sur le canal R retourne la réponse. Aucune donnée n'est retournée pour ce type de transactions.

— *MakeUnique* : *MakeUnique* peut seulement être utilisée par un *master* effectuant une opération *store* d'une *full* ligne de cache. La ligne de cache est retenue avec l'état *unique*. Aucune donnée n'est obtenue en réponse à cette transaction et tous les autres copies de la ligne de cache sont supprimées ;
— *MakeInvalid* : MakeInvalid est utilisée pour assurer qu'une aucune copie d'une ligne mémoire n'existe dans les caches ; Si le *master* effectuant cette opération de maintenance détient une copie de la ligne de cache à l'état *valid*, il doit invalider sa ligne de cache avant d'émettre la transaction $MakeInvalid$.

D) Les Transactions d'écriture : L'adresse est envoyée sur le canal AW, le transfert de données sur le canal W et la réponse est retournée sur le canal B.

— *WriteNoSnoop* : *WriteNoSnoop* est utilisée dans une région de la mémoire qui n'est pas partagée avec d'autres *masters*. Elle est le résultat de : (i) une action du programme (une opération *store* par exemple), (ii) une mise à jour de la mémoire pour une ligne de cache qui est dans une région mémoire non partagée. La ligne de cache peut seulement passer de l'état *invalid* à l'état *valid* si un *store* d'une *full* cache ligne a été effectué ;
— *WriteUnique* : $WriteUnique$ est utilisée dans une région de la mémoire qui est partagée avec d'autres *masters*. L'écriture est propagée vers la mémoire. Dans le cas où ce cache détient une copie de la ligne de cache à l'état *Clean*, elle doit être mise à jour avec la nouvelle valeur quand la réponse à la transaction $WriteUnique$ est reçue ;
— *WriteLineUnique* : $WriteLineUnique$ est utilisée dans une région de la mémoire qui est partagée avec d'autres *masters*. L'écriture est propagée vers la mémoire. Il s'agit d'une opération de *store* sur une *full* ligne de cache. Tous les *bytes* doivent être mis à jour ; Dans le cas où ce cache détient une copie de la ligne de cache à

l'état *Clean*, elle doit être mise à jour avec la nouvelle valeur quand la réponse à la transaction $WriteLineUnique$ est reçue ;
— *WriteBack* : *WriteBack* est utilisée dans une région de la mémoire qui est partagée ou non-partagée avec d'autres *masters*. Il s'agit d'une écriture d'une ligne de cache *Dirty* pour mettre à jour la mémoire. Après une transaction *WriteBack*, la ligne de cache n'est plus allouée ;
— *WriteClean* : *WriteClean* est utilisée dans une région de la mémoire qui est partagée ou non-partagée avec d'autres *masters*. Il s'agit d'une écriture d'une ligne de cache *Dirty* pour mettre à jour la mémoire. Après une transaction *WriteClean*, la ligne de cache reste allouée.

E) Les transactions *Evict* : Elles sont envoyées sur le canal AW, la réponse est retournée sur le canal B. Aucune donnée n'est retournée pour ce type de transactions.

— *Evict* : *Evict* indique que la ligne de cache a été retirée du cache local d'un *master*. Aucun transfert de donnée n'est associé à cette transaction. Elle est utilisée dans une région de la mémoire qui est partagée.

3.3.7 Affectation des requêtes de cohérence aux requêtes de *snooping*

Lorsqu'un *master* initie une transaction, le *CoreLink* est responsable de l'émission de la transaction de *snooping* correspondante pour compléter la transaction originelle. Les transactions envoyées par les *masters* ne sont pas toutes autorisées sur le canal AC. Le tableau 3.8 représente l'affectation des opérations de cohérence aux opérations de *snooping* [AMBA and Specification-AXI, 2011].

3.3.8 Séquencement des transactions

Plusieurs *masters* peuvent émettre des transactions au même instant. Le *CoreLink* doit assurer un ordre bien défini des requêtes d'accès à la même ligne de cache. En effet, dans le cas où deux *masters* émettent des transactions à la même ligne de cache au même instant, le *CoreLink* détermine l'ordre des transactions en séquencant les réponses aux transactions et les transactions de *snooping*. Les règles suivantes s'appliquent :

— Si un *master* émet une transaction à une ligne de cache et reçoit une transaction de *snooping* vers la même ligne de cache avant d'avoir reçu la réponse à la transaction émise, la transaction de *snooping* est séquencée en première ;
— Si un *master* émet une transaction à une ligne de cache et reçoit une réponse avant de recevoir une transaction de *snooping* à la même ligne de cache, alors la transaction émise est traitée avant la transaction de *snooping* ;
— Si le *CoreLink* fournit une réponse à une transaction émise par un *master*, il ne doit pas envoyer à ce *master* une transaction de *snooping* à cette même ligne de cache avant d'avoir reçu le signal $RACK$ ou $WACK$ de ce *master* ;
— Si le *CoreLink* envoie une transaction de *snooping* à un *master*, il ne doit pas lui envoyer une réponse à une transaction à la même ligne de cache avant d'avoir reçu la réponse $CRRESP$ associée de ce *master*.

Transaction initiée par un master	Transaction de snooping
ReadNoSnoop	Non snoopée
ReadOnce	ReadOnce
ReadClean	ReadClean
ReadNotSharedDirty	ReadNotSharedDirty
ReadShared	ReadShared
ReadUnique	ReadUnique
CleanUnique	CleanUnique
MakeUnique	MakeInvalid
CleanShared	CleanShared
CleanInvalid	CleanInvalid
MakeInvalid	MakeInvalid
WriteNoSnoop	Non snoopée
WriteUnique	CleanInvalid
WriteLineUnique	MakeInvalid
WriteBack	Non snoopée
WriteClean	Non snoopée
Evict	Non snoopée

FIGURE 3.8 – *Mapping* des transactions de cohérence aux opérations de *snooping*.

3.3.9 Complexité du problème de vérification

La vérification de la cohérence dans un système multicœurs est complexe. Dans le protocole ACE, il existe une vingtaine de types d'opérations de lecture et d'écriture. Afin de tester chaque type d'opération, il convient de construire des scenarii de vérification intégrant, pour tout type de requête, tous les états possibles de la ligne de cache concernée. Cette tâche est complexe. La complexité réside dans la taille de l'espace de vérification où chaque permutation de scenario doit être complètement vérifiée. A titre d'exemple, le nombre de combinaisons d'une spécification ACE dépend des types de transactions, des types de réponses et des états des lignes de cache et par conséquent croit vite avec l'augmentation du nombre de processeurs et coprocesseurs dans le système. Dans le protocole ACE, une règle générale définit ce nombre comme étant approximativement 7000^N, où N est le nombre de *masters*. Dans le cas d'une système quadri-cœurs par exemple, 28000 produits doivent être vérifiés.
Ce nombre restreint le nombre de composants de calcul supportés par les produits ARM et plus particulièrement les *CoreLink CCN*. Le défi de vérification est majeur, la complexité ne réside pas seulement dans le design, mais aussi dans l'effort de vérification. La cohérence restreinte est obtenue aux dépends de la flexibilité.
Des méthodes de vérification efficaces sont requises. Une première solution consiste à réduire l'espace de vérification aux seules permutations applicables. En effet, certains produits ne sont pas légaux, comme par exemple le cas où deux *masters* détiennent une ligne de cache en $UniqueDirty$.

3.4 Conclusion

Dans ce chapitre, nous avons présenté les systèmes hétérogènes à base de multiprocesseurs en donnant une description des *CoreLink CCN*. Nous avons ensuite défini la cohérence de cache dans le cas des systèmes hétérogènes en général et dans les cas des systèmes hétérogènes à base de réseaux d'interconnexion *CoreLink CCN*. Une spécification du protocole AMBA ACE a été donnée en décrivant les principaux éléments introduits dans cette spécification. Ce protocole est devenu un standard pour la gestion de la cohérence dans les systèmes hétérogènes. La vérification d'un tel protocole est très intéressante et représente un défi majeur, vu la complexité de la tâche. Dans le chapitre suivant, nous décrirons la modélisation de ce protocole et son implémentation sur l'architecture générique et extensible que nous avons conçue autour de l'interconnexion *CoreLink CCN*.

Chapitre 4

Implémentation du protocole de cohérence de cache AMBA ACE sur une architecture générique et extensible

Dans ce chapitre, nous présentons l'architecture générique et extensible proposée et détaillons la micro-architecture de l'interconnexion *CCN-xx*. Nous décrivons ensuite la modélisation de chaque objet la composant et l'implémentation du protocole AMBA ACE sur cette architecture. En dernier point, nous expliquons le processus d'exécution de 3 types de requêtes sur notre architecture.

4.1 Architecture générique

Il s'agit d'une architecture de systèmes à mémoire partagée. Les principaux composants de cette architecture sont les processeurs, les coprocesseurs, le réseau d'interconnexion et la mémoire partagée.
Dans cette étude, l'interconnexion *CoreLink CCN-xx* regroupe dans un seul et même module la fonction d'interconnexion des composants et celle de la gestion des accès concurrents à la mémoire en utilisant le protocole AMBA ACE. Les 2 types d'interfaces spécifiées dans le protocole, à savoir ACE *master* et ACE-lite *master* sont implémentées.
Afin d'implémenter le protocole AMBA ACE sur une architecture à base d'interconnexion CCN étendue, nous avons modélisé des composants équivalents à chaque composant de l'architecture de base : les processeurs connectés aux $L2$, les coprocesseurs avec l'interface ACE-Lite *master*, la mémoire, les caches locaux aux processeurs avec l'interface ACE *master* et le réseau d'interconnexion *CoreLink CCN-xx*. La figure 4.1 décrit l'architecture générique modélisée.

4.1.1 Modélisation des processeurs et coprocesseurs

La vérification dynamique exige des transacteurs servant à générer des *patterns* de test qui doivent suivre le protocole de cohérence. Pour ce faire, nous avons implémenté deux types de composants : *proc* pour modéliser les processeurs et *coproc* pour modéliser les coprocesseurs n'ayant pas de cache local (GPU, DSP, etc.). Afin de tester les transactions

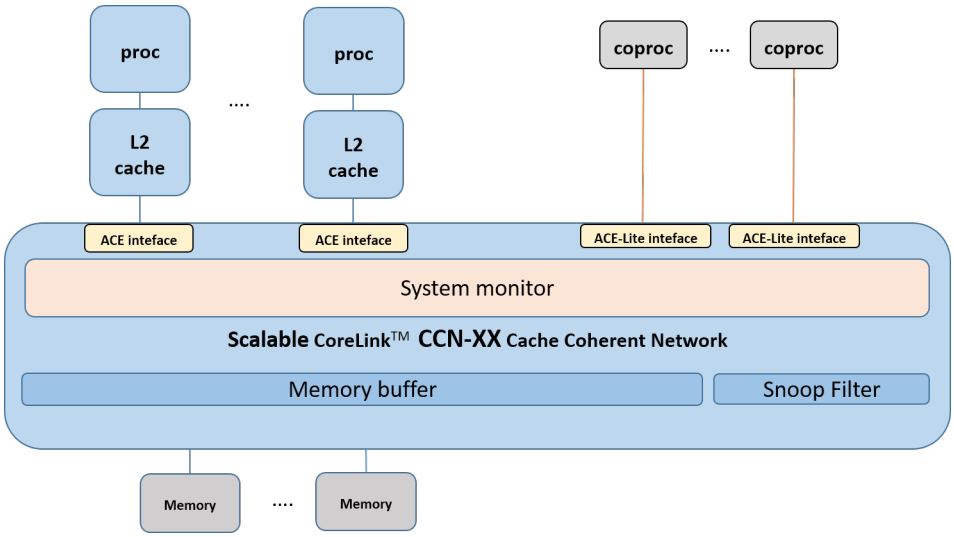

FIGURE 4.1 – L'architecture générique proposée.

ACE sur l'architecture générique, une fonction de génération aléatoire de requêtes est implémentée dans chacun de ces composants : le type de la requête et l'adresse mémoire sont générés aléatoirement.
A chaque génération, une nouvelle transaction identifiée par un identificateur unique TID est créée. Elle contient comme principales informations l'identificateur ID du processeur ou coprocesseur l'ayant générée, le type de la requête et l'adresse d'accès à la mémoire. Cette requête d'accès est envoyée au cache local, sous forme de paquet, dans le cas d'un *proc* et directement envoyée au réseau d'interconnexion dans le cas d'un *coproc*. Le transacteur terminera la transaction en cours après réception de la réponse et pourra ensuite initier une nouvelle transaction. Si la requête n'a pas été traitée, il recevra une requête de renvoi après un délai d. Si la réponse n'est pas reçue après $maxReqTick$ secondes, celle-ci sera ignorée.
La figure 4.2 illustre les groupes de requêtes pour chaque type de *master* (processeur ou coprocesseur).
Ces composants sont reliés aux caches locaux dans le cas de *proc* et au réseau d'interconnexion dans le cas de *coproc*, comme illustrée dans la figure 4.1.

Fonctions de *proc* et *coproc*

Les procédures d'envoi d'une requête, de réception d'un *retry* (pour un renvoi de requête), de renvoi d'une requête, de réception d'une réponse et de fin d'une transaction sont résumées algorithmiquement dans les algorithmes 4.1, 4.2 et 4.3 .

4.1.2 Modélisation des caches

Afin de modéliser un cache local aux composants *proc* de niveau 2, nous avons implémenté le composant $L2$. Cette structure est composée de lignes de cache stockées sous forme d'une liste de blocs de données. A chaque ligne de cache est associé un état parmi les suivants : $Invalid$ (I), $UniqueClean$ (UC), $UniqueDirty$ (UD), $SharedClean$ (SC) et $SharedDirty$ (SD). Ces états sont décrits dans la section 3.3.

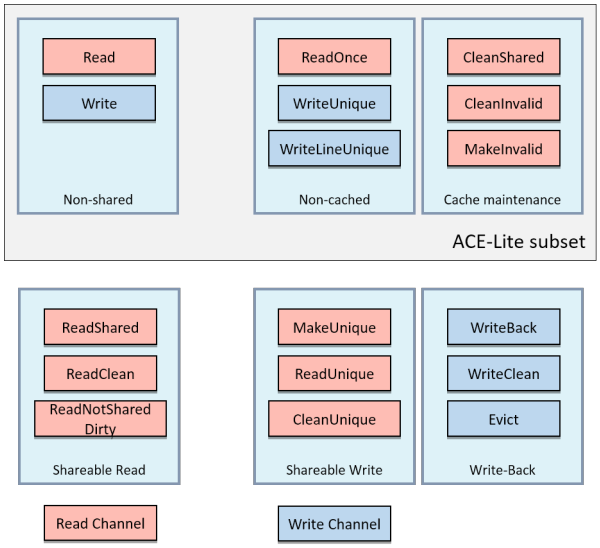

FIGURE 4.2 – Les groupes de transactions selon le protocole AMBA ACE

.

Algorithme 4.1 *tick*()

Générer aléatoirement un type de requête et une adresse.
if requête d'écriture **then**
    Générer aléatoirement une donnée.
end if
Composer un paquet *pkt* contenant les informations sur la requête.
Envoyer *pkt* sur le port en sortie.

Algorithme 4.2 *recvRetry*(*pkt*)

Renvoyer *pkt* sur le port en sortie après un délai d.

Algorithme 4.3 *recvResp*(*pkt*)

if requête de lecture **then**
    Lire la donnée reçue.
end if
Supprimer le paquet.
Générer une nouvelle requête.

A la réception d'une requête, le cache réagit suivant l'état initial de la ligne de cache correspondant à l'adresse d'accès. Des fonctions d'allocation et de dés-allocation des lignes du cache sont implémentées.

Dans le but d'injecter des requêtes suivant le protocole de cohérence AMBA ACE, une fonction *preChecking* est implémentée, afin de rejeter les requêtes incompatibles suivant l'état initial de la ligne de cache concernée par la requête (exemple : une requête *write-back* ne peut pas être générée sur une ligne de cache invalide).

Une fonction d'accès est implémentée pour vérifier la validité de la ligne de cache. Les 3 scénarii possibles sont :

— Requête de lecture : Dans le cas d'un *hit* (succès), la donnée de la ligne de cache est fournie au processeur (*proc*) si son état actuel le permet (tous les cas possibles seront détaillés dans la section 4.3.2), sinon, la requête est envoyée au port en entrée du réseau d'interconnexion. Une fois la réponse à cette requête reçue, une copie est stockée localement et une réponse au processeur ayant initié la requête est envoyée ;
— Requête d'écriture : Dans le cas d'un *hit* en état unique et *Clean*, l'état de la ligne de cache est mis à jour sinon, une copie exclusive de la donnée est demandée. La requête générée dépend de la requête initiale (comme définie dans la spécification ACE que nous détaillerons dans la section 4.3.2). Une fois la réponse reçue, la nouvelle donnée est stockée localement si le protocole de cohérence le permet et la réponse est envoyée au processeur initiateur de la requête ;
— Requête de *snooping* : A la réception d'une requête du *CCN-xx*, le cache envoie une réponse suivant le protocole ACE. Pour une requête de lecture par exemple, la donnée est envoyée et l'état de la ligne de cache est mis à jour. Nous n'étudions pas le cas d'un *miss*. Le composant $snoopFilter$ permet de *snooper* seulement les caches détenant la ligne de cache.

Le remplacement des lignes de cache lors de l'allocation de nouveaux blocs suit un algorithme LRU (*Least Recently Used*) [Puzak, 1985b].

Fonctions de *L2*

Les algorithmes 4.4, 4.5, 4.6 et 4.7 résument les fonctions suivantes : envoi d'une réponse à *proc*, réception d'une requête de *proc*, accès à un bloc d'adresse x, réception d'une réponse à une requête envoyée au *CCN-xx*, allocation d'un bloc et envoi d'une requête au *CCN-xx*. Le déroulement de ces fonctions est détaillé dans la section 4.3.1.

4.1.3 Modélisation de la mémoire partagée

Le composant *mem* modélise la mémoire principale partagée dont la taille est paramétrisable. Le tampon du port en entrée de la mémoire contient les requêtes envoyées pour traitement par le *CCN-xx*. Si la requête nécessite une réponse, elle sera envoyée sur le port de sortie.

Algorithme 4.4 *L2 : recvReq(pkt)*

accéder au bloc d'adresse x.
if Bloc est invalide **then**
    envoyer la requête correspondante au *CCN-xx* sur le port en sortie.
else
    if requête de *snooping* en cours sur l'adresse x **then**
        envoyer un *retry* à *proc*.
    else
        switch type de la requête **do**
            case *ReadShared*
                répondre en envoyant la donnée à *proc* et mettre à jour l'état de ligne de cache.
            case *ReadUnique*
                if l'état initial de la ligne de cache est UC ou UD **then**
                    répondre en envoyant la donnée à *proc* et mettre à jour l'état de ligne de cache.
                else
                    envoyer une requête au *CCN-xx* sur le port en sortie.
                end if
            case *ReadNotSharedDirty*
                if l'état initial de la ligne de cache est UC, UD ou SC **then**
                    envoyer la donnée à *proc* et mettre à jour l'état de ligne de cache.
                else
                    envoyer une requête *CleanShared* au *CCN-xx* sur le port en sortie.
                end if
            case *ReadClean*
                répondre en envoyant la donnée à *proc* et mettre à jour l'état de ligne de cache.
            case $WriteUnique$ ou $WriteLineUnique$
                if l'état de la ligne de cache est UD **then**
                    envoyer une requête *WriteClean* au *CCN-xx*.
                else
                    if $WriteUnique$ **then**
                        envoyer une requête *CleanInvalid* au *CCN-xx*.
                    else
                        envoyer une requête $MakeInvalid$ au *CCN-xx*.
                    end if
                end if
            case $CleanUnique$
                if l'état de la ligne de cache est SC ou SD **then**
                    envoyer une requête $MakeInvalid$ au *CCN-xx*.
                end if

case *CleanInvalid*
if la donnée est *Dirty* **then**
envoyer la requête *CleanInvalid* au *CCN-xx* avec la donnée en lui passant la responsabilité de mise à jour de la mémoire.
else
envoyer une requête *CleanInvalid* au *CCN-xx*.
end if
case *MakeUnique*
if l'état de la ligne de cache est UC ou UD **then**
effectuer l'écriture localement, mettre à jour l'état de la ligne de cache et envoyer une réponse à *proc*.
else
envoyer une requête *MakeUnique* au *CCN-xx*.
end if
end if
end if

fonctions de *mem*

L'algorithme général de la réception d'une requête et des actions engendrées est décrit dans l'algorithme 4.8. Le déroulement des fonctions de réception d'une requête, d'accès à un emplacement mémoire, d'envoi d'une réponse au CCN est détaillé dans la section 4.3.1.

4.1.4 Modélisation du réseau d'interconnexion

Nous avons étendu le réseau d'interconnexion *CoreLink CCN* pour supporter des configurations plus grandes. Le nouveau réseau d'interconnexion est noté *CCN-xx*. L'utilisation efficace des systèmes à base d'interconnexion *CCN-xx* repose sur la qualité de la gestion du parallélisme. Pour une bonne gestion des accès parallèles et concurrents à la mémoire, nous définissons un composant, appelé moniteur système, qui permettra de gérer le réseau d'interconnexion. Pour optimiser le nombre d'accès à la mémoire extérieure, un *buffer* mémoire est ajouté en plus du *snoop filter* déjà présent dans les *CoreLink CCN*.

Modélisation du moniteur système

Ce composant permet de gérer le réseau d'interconnexion et les traitements parallèles des transactions dans le système. Il est nommé $systemMonitor$.
Il est composé d'un tableau de taille limitée $maxS$, qui enregistre les transactions en cours de traitement dans le système d'interconnexion *CCN-xx*. Chaque ligne de ce tableau contient l'information sur l'état de la transaction. La figure illustre la structure de ce tableau appelé $requestList$.

A) Modélisation des règles de séquencement
La spécification ACE définit un ordre strict entre les opérations simultanées d'écriture sur une même adresse. Pour ce faire, un pré-traitement sur les requêtes est fait par

Algorithme 4.5 *L2 : recvResp(pkt)*

Vérifier si la réponse reçue ne contient pas d'erreur.
switch type de la réponse **do**
case *ReadShared*
allouer une ligne de cache avec l'état correspondant, copier la donnée reçue, et envoyer la réponse à *proc*.
case *ReadUnique*
if l'état initial de la ligne de cache est *I* **then**
allouer une ligne de cache avec l'état correspondant, copier la donnée reçue, et envoyer la réponse à *proc*.
else
mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.
end if
case *ReadNotSharedDirty* ou *CleanShared*
if l'état initial de la ligne de cache est *I* **then**
allouer une ligne de cache, copier la donnée reçue, spécifier l'état de ligne de cache et envoyer la réponse à *proc*.
else
mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.
end if
case *ReadClean*
if l'état initial de la ligne de cache est *I* **then**
allouer une ligne de cache avec l'état correspondant, copier la donnée reçue, et envoyer la réponse à *proc*.
else
mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.
end if
case *WriteUnique* ou *WriteLineUnique*
if l'état initial de la ligne de cache est I **then**
envoyer la réponse à *proc*.
else
écrire la nouvelle donnée dans la ligne de cache, mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.
end if
case *CleanUnique*
if l'état initial de la ligne de cache est I **then**
envoyer la réponse à *proc*.
else
mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.
end if
case *MakeUnique* ou *MakeInvalid*
écrire la donnée dans la ligne de cache, mettre à jour l'état de ligne de cache et envoyer la réponse à *proc*.

Algorithme 4.6 *L2 : recvRetry(pkt)*

Envoyer une requête *retry* à *proc*.

Algorithme 4.7 *L2 : recvSnoopingReq(pkt)*

if une requête en cours sur la ligne de cache demandée **then**
mettre la requête en attente.
else
switch type de la requête **do**
case *ReadOnce*
répondre en envoyant la donnée au *CCN-xx*.
case *ReadShared*
répondre en envoyant la donnée au *CCN-xx* et mettre à jour l'état de ligne de cache.
case *ReadUnique*
if la donnée est *Dirty* **then**
répondre en envoyant la donnée au *CCN-xx* en envoyant la donnée et en passant la responsabilité de la mise à jour de la mémoire, invalider la ligne de cache et mettre à jour son état.
else
répondre en envoyant la donnée au *CCN-xx*, invalider la ligne de cache et mettre à jour son état.
end if
case *ReadNotSharedDirty* ou *ReadClean*
if la donnée est *Dirty* **then**
répondre en envoyant la donnée et la responsabilité de mise à jour de la mémoire au *CCN-xx* et mettre à jour l'état de ligne de cache.
else
répondre en envoyant la donnée au *CCN-xx* et mettre à jour l'état de ligne de cache.
end if
case *CleanUnique*
if la donnée est *Dirty* **then**
répondre en envoyant la donnée au *CCN-xx* en envoyant la donnée et en passant la responsabilité de la mise à jour de la mémoire, invalider la ligne de cache et mettre à jour son état.
else
invalider la ligne de cache et mettre à jour son état.
end if
case *CleanInvalid*
if la donnée est *Dirty* **then**
envoyer la donnée au *CCN-xx* en lui passant la responsabilité de mise à jour de la mémoire, invalider sa ligne de cache et mettre à jour son état.
else
invalider sa ligne de cache et mettre à jour son état.
end if

case *MakeInvalid*
invalider sa ligne de cache et mettre à jour son état.
case *CleanShared*
if la donnée est *Dirty* **then**
envoyer la donnée au *CCN-xx* en lui passant la responsabilité de mise à jour de la mémoire et mettre à jour son état.
else
mettre à jour l'état de la ligne de cache.
end if
end if

Algorithme 4.8 *mem* : *recvReq*(*pkt*)

if requête de lecture ou d'écriture en cours **then**
mettre en attente la requête dans le *buffer* en entrée.
else
if requête de lecture **then**
accéder au bloc de l'adresse, copier la donnée dans le *pkt* reçu et envoyer la réponse sur le port en sortie.
else
écrire la donnée à l'adresse correspondante.
end if
end if

Transaction	Etat de la transaction			
	Ready	ReadyForSnoopFilter	ReadyForMemoryBuffer	completed

FIGURE 4.3 – Structure de reqList.

systemMonitor. Une requête ne peut pas être envoyée pour traitement dans le *CCN-xx* tant que cette règle n'est pas respectée. Ceci permet de garantir la cohérence des données dans les caches et la consistance par rapport à la mémoire partagée. Les règles vérifiées sont les suivantes :

— Toute requête sur une adresse dont la donnée est en cours de modification doit être mise en attente. Une requête *retry* est envoyée au *master* initiateur de la requête ;
— Toute requête d'accès exclusif à une adresse doit être mise en attente si une opération est en cours sur cette adresse. Une requête *retry* est envoyée au *master* initiateur de la requête.

B) Les fonctions de *systemMonitor* :
Une tache appelée *trackRequests* se charge d'identifier les nouvelles requêtes entrant au *CCN-xx* en parcourant tous les ports en entrée (port ACE et port ACE-Lite) et de suivre leurs évolutions en parcourant :

— Les *buffers* en sortie des éléments le composant, à savoir $snoopFilter$ et $memoryBuffer$;
— Les ports sur lesquels sont reçues les réponses au *snooping* (canaux de snooping) ;
— Les ports en entrée connectés à la mémoire.

Dans le but d'accélérer la simulation, un arbitre est placé tous les x processeurs ou coprocesseurs pour en sélectionner qu'un seul lors de la phase de parcours des ports en entrée. L'algorithme utilisé dans la sélection est *round robin*.

A chaque *tick* (unité de temps), la tâche *trackRequests* prend en compte soit une nouvelle requête d'accès envoyée sur le réseau, soit une mise à jour d'une requête existante. Dans le premier cas, un pré-traitement est effectué selon le type de la transaction et l'adresse demandée. Si les règles d'accès simultané du protocole ACE sont vérifiées, la requête est ajoutée au tableau avec comme état : *Ready*. Dans le second cas (réponse à une requête), la réponse à une requête par un des composants déterminera l'emplacement suivant de la requête : la réponse est envoyée au $snoopFilter$ ou au $memoryBuffer$. Le composant se chargera d'effectuer le traitement nécessaire. Ce comportement est illustré dans l'algorithme 4.9.

Une tâche appelée *trackUpdates* se charge de parcourir les *buffers* en sortie des composants $snoopFilter$ et $memoryBuffer$. Cette fonction est décrite dans l'algorithme 4.10.

La troisième tâche *processRequests* consiste à déclencher le traitement parallèle de toutes les requêtes prêtes à s'exécuter et de les envoyer au composant correspondant pour être traitées. Ce comportement est illustré dans l'algorithme 4.11.

Il existe deux types de requêtes : les requêtes cohérentes et les requêtes non cohérentes. Ces dernières seront directement envoyées au *buffer* mémoire après leur introduction dans le tableau *reqList*. Les trois tâches *trackRequests*, *trackUpdates* et *processRequests* s'exécutent en parallèle.

Algorithme 4.9 *systemMonitor* : *trackRequests*

while simulation non terminée **do**
  Parcourir les ports en entrée du *CCN-xx*.
  for pour chaque canal des requêtes et des réponses, tous les x ports **do**
    Sélectionner un port suivant l'algorithme round robin.
    if requête **then**
      if une des règles spécifiées dans ACE n'est pas respectée ou la taille maximale de *reqList* atteinte **then**
        Envoyer un *retry*.
      else
        Insérer la requête dans la table *reqList* avec le statut *Ready*.
      end if
    else
      if réponse à une requête de non *snooping* **then**
        Mettre à jour l'état de la requête $ReadyForMemoryBuffer$.
      else
        if une mise à jour de la mémoire est nécessaire **then**
          Mettre à jour l'état de la requête $ReadyForMemoryBuffer$.
        else
          Mettre à jour l'état de la requête $ReadyForSnoopFilter$.
        end if
      end if
    end if
  end for
end while

Algorithme 4.10 *systemMonitor* : *trackUpdates*

while simulation non terminée **do**
  Parcourir les *buffers* en sortie de $snoopFilter$ et $memoryBuffer$.
  if Réponse du $snoopFilter$ **then**
    if un *miss* pour une requête **then**
      Mettre à jour l'état de la requête à $ReadyForMemoryBuffer$.
    else
      Terminer la transaction et mettre à jour son état à *completed*.
    end if
  end if
  if réponse du $memoryBuffer$ **then**
    une donnée est disponible.
    if réponse à une requête de non *snooping* **then**
      Envoyer une réponse à *coproc* initiateur de la requête et mettre à jour l'état de la requête à *completed*.
    else
      Mettre à jour l'état de la requête à $ReadyForSnoopFilter$
    end if
  end if
end while

Algorithme 4.11 *systemMonitor* : *processRequests*()

while simulation non terminée **do**
  Parcourir *reqList*.
  if Transaction *ReadyForSnoopFilter* **then**
    Envoyer la transaction au *snoopFilter* pour traitement.
  else
    Envoyer la transaction au *memoryBuffer* pour traitement.
  end if
end while

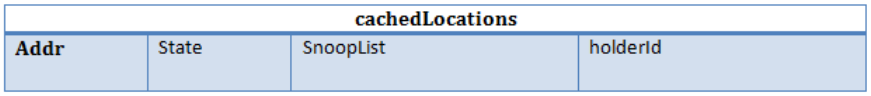

FIGURE 4.4 – Structure d'une entrée du *snoopfilter*.

Modélisation du *snoopFilter*

Dans le but d'optimiser la temps de réponse aux requêtes, un composant appelé *snoopFilter* a été ajouté dans l'architecture des *CCNs*. Le modèle que nous proposons permet de garder trace de l'emplacement des lignes de caches dans le système en détenant une liste *snoopList* de tous les caches où réside une ligne de cache donnée, son état et l'identifiant du *master* ayant pour rôle la mise à jour de la mémoire (*holder*). La figure 4.4 illustre la structure d'une entrée du *snoopFilter*. Dans le cas d'une requête lecture *ReadShared* par exemple, si la donnée réside dans plus d'un cache, un seul cache répondra en envoyant la donnée au *CCN-xx*. L'état de la ligne de cache est mis à jour à chaque modification qui survient dans le système.
Le traitement d'une requête est déclenché par *systemMonitor*. Si la ligne demandée existe dans le système, le *snoopFilter* retourne la liste des caches détenant la ligne de cache demandée. Selon le type de la transaction, une requête sera envoyée à l'un de ces caches ou à tous les caches (cas d'une opération de diffusion). Dans le cas où la ligne de cache n'existe pas, une requête d'accès au *buffer* mémoire sera signalée.

Le déroulement des fonctions du *snoop filter* sera détaillé dans la section 4.3.1. Les algorithmes de réception d'une requête et de mise à jour sont décrits dans 4.12 et 4.13.

Modélisation du *buffer* mémoire

Ce composant a été ajouté afin de garder une copie des données transférées de la mémoire aux processeurs et coprocesseurs. Il permet d'optimiser la durée d'accès. Toute donnée transmise de la mémoire au *CCN-xx* résidera dans le *buffer* mémoire. Ceci permet de lire la donnée depuis le *buffer* mémoire, sans avoir à accéder à la mémoire extérieure. Dans notre implémentation, il est relié à la mémoire par un port *memSide*.
Le déroulement des fonctions du *buffer* mémoire sera détaillé dans la section 4.3.1. L'algorithme décrit la fonction de réception d'une requête.

Algorithme 4.12 *snoopFilter* : *recvReq*(*pkt*)

Vérifier pour l'adresse *addr*, si une ligne de cache est allouée dans le système.
if *snoopList* retourne une liste vide pour cette *addr* **then**
    Envoyer la réponse sur le *buffer* de sortie (*miss*).
else
    if requête de diffusion **then**
        Envoyer la requête de *snooping* à tous les *L*2 de *snoopList*(*addr*).
    else
        if Si la donnée dans les caches est *Dirty* **then**
            Envoyer la requête de *snooping* au *L*2 correspondant au *holderId*
        else
            Envoyer la requête de *snooping* au premier *L*2.
        end if
    end if
end if

Algorithme 4.13 *snoopFilter* : *updateSnoopFilter*()

if La réponse reçue contient une donnée **then**
    Mettre à jour l'état de *snoopList* et envoyer la réponse à *L*2 initiateur de la requête.
else
    Mettre à jour l'état de *snoopList* et envoyer la réponse à *L*2 initiateur de la requête.
end if

Algorithme 4.14 *memoryBuffer* : *recvReq*()

if requête de lecture **then**
    Vérifier si la donnée est disponible dans le *buffer*.
    if donnée disponible **then**
        Répondre avec la donnée sur le *buffer* de sortie.
    else
        Envoyer la requête à la mémoire.
    end if
else
    if requête d'écriture **then**
        Écrire la donnée dans le *buffer* et envoyer une requête d'écriture à la mémoire.
    else
        Écrire la donnée dans le *buffer*.
    end if
end if

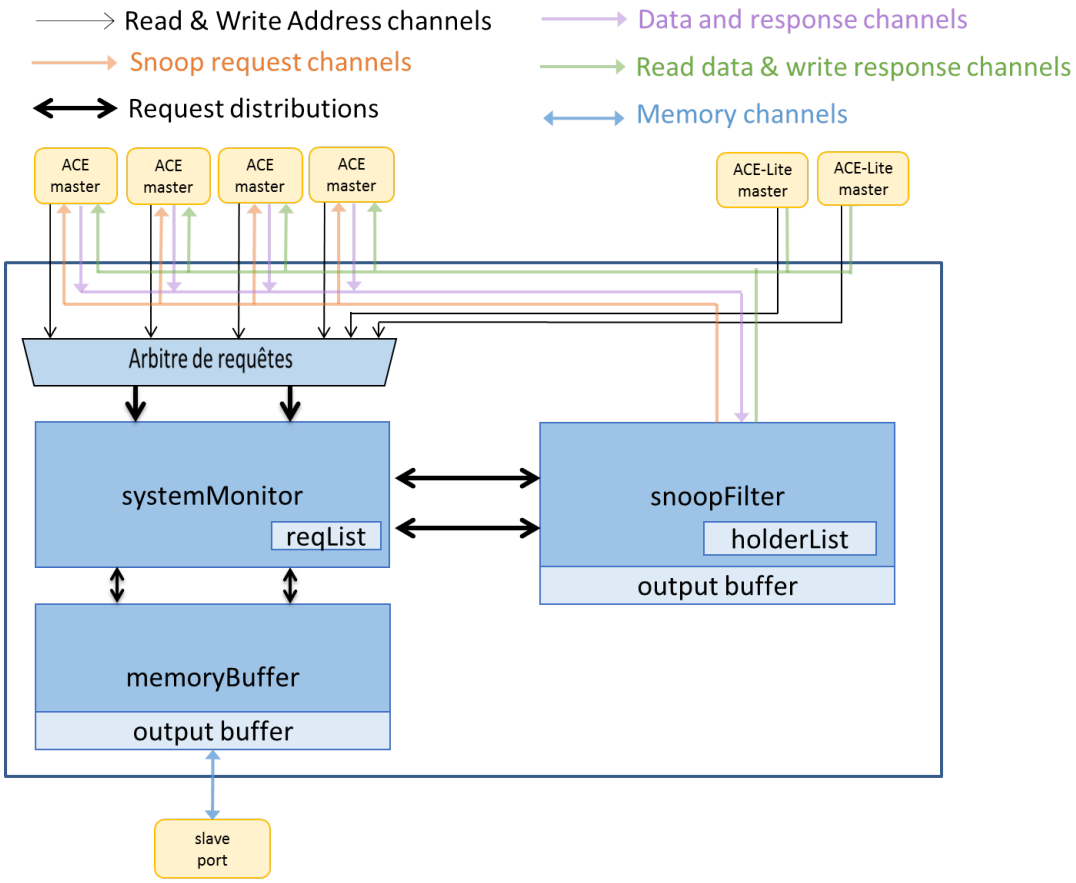

FIGURE 4.5 – Micro-architecture de l'interconnexion *CCN-xx*.

4.2 Micro-architecture de l'interconnexion *CCN-xx*

4.2.1 L'architecture interne de l'interconnexion CCN étendue

Le schéma bloc de l'interconnexion CCN étendue est illustré dans la figure 4.5. Le nombre de ports en entrée et en sortie du *CCN-xx* est paramétrisable. La liaison entre le *CCN-xx* et les *L2/coproc* comporte plusieurs canaux virtuels.

4.2.2 Échange des données

Dans notre implémentation, les composants communiquent par envoi de paquets via des ports. A chaque port est associé un tampon de profondeur limitée où seront stockées les requêtes ou réponses reçues des composants externes. La figure 4.6 (a) décrit l'interconnexion L2(*master*)/*CCN-xx* (*slave*), la figure 4.6 (b) décrit l'interconnexion *coproc* (*master*)/*CCN-xx* (*slave*) et la figure 4.7 décrit les canaux entre le *CoreLink CCN-xx* et la mémoire.

4.2.3 Les options de configuration

— Le nombre de ports en entrée et en sortie du *CCN-xx* est paramétrisable. Il dépend du nombre de processeurs, coprocesseurs et mémoires auxquels il est connecté ;

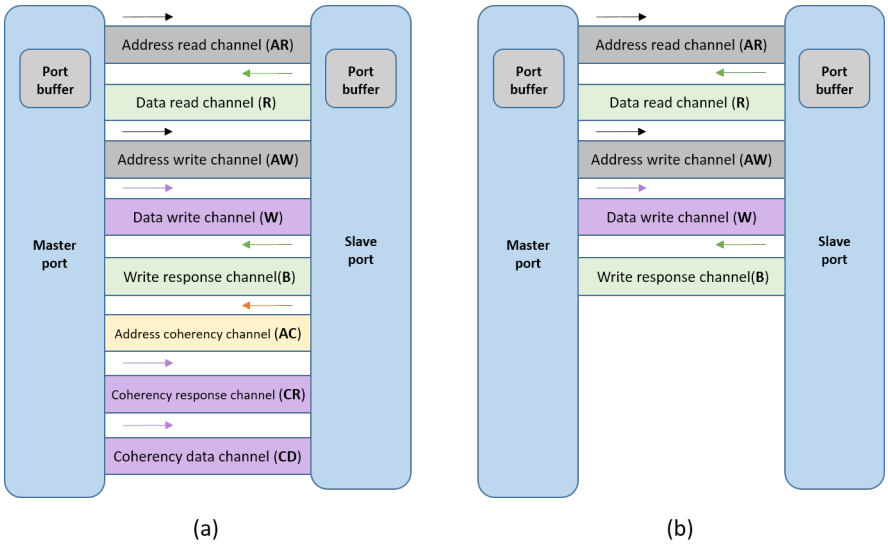

FIGURE 4.6 – Architecture des canaux : (a) pour un ACE *master*, (b) pour un ACE-Lite *master*.

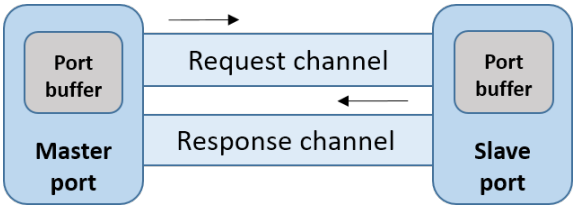

FIGURE 4.7 – Architecture des canaux entre le *CCN-xx* et la mémoire.

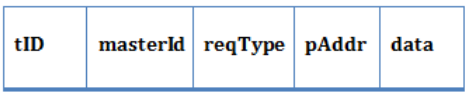

FIGURE 4.8 – Structure d'une transaction.

— La taille des tampons dans chaque port ;
— la taille de *reqList* et de *holderList* ;
— La nombre maximal de requêtes traitées à chaque cycle par le *CCN-xx* ;
— Les latences de réponse de chaque composant.

4.3 Implémentation du protocole ACE

Le protocole ACE assure que tous les processeurs et coprocesseurs d'un système observent la valeur correcte de la donnée à n'importe quelle adresse mémoire en forçant l'existence d'une seule copie après une opération d'écriture. L'implémentation du protocole ACE consiste à : (1) modéliser les états des lignes des caches *L2* selon le modèle d'états défini dans la section 3.3.1, (2) modéliser les différents types de requêtes, ainsi que les actions générées par chacune (4.3.2), (3) modéliser les canaux sur lesquels seront envoyées les requêtes et les réponses et (4) modéliser les règles de séquencement des transactions.

4.3.1 Description d'une transaction

Dans notre modélisation transactionnelle, quand une transaction est initiée, un paquet est envoyé sur le réseau. Il contient un ensemble d'informations définies comme étant : l'identifiant de la transaction tID qui est unique, l'identifiant du *master* $masterId$ ayant initié la transaction, le type de la requête $reqType$ qui est détaillé dans la section 3.3.6, l'adresse physique $pAddr$ à laquelle on veut accéder et la donnée, dans le cas d'une requête d'écriture ou d'une réponse avec une donnée. La figure 4.8 illustre la structure d'une transaction.

4.3.2 Exécution complète des transactions

Description du flot général d'une transaction dans le système étudié

Quand une transaction est envoyée par un *master*, elle peut être soit une requête cohérente soit une requête non cohérente. Cette information est extraite du type de la transaction. Le flot général d'une transaction est comme suit :

— Un *master* (processeur ou coprocesseur) envoie une transaction ;
— Le *L2* répond directement au *master proc* ou renvoie la requête au *CCN-xx* ;
— La contrainte de séquencement des transactions sur la même adresse est vérifiée pour cette requête par le $systemMonitor$. Une fois le test validée, elle est prête pour le traitement dans le *CCN-xx* ;
— Selon que la requête soit cohérente ou non : (i) La requête va être directement passée par le $systemMonitor$ au $memoryBuffer$ et par la suite au composant

Tableau 4.1 – Actions et transitions générées pour une requête *ReadShared*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
ReadShared	I	*ReadShared*	SC/UC	UC	SC/I
	I	*ReadShared*	$SD/UD/$	UD*	SD/I
	I	*ReadShared*	SC/UC	SC	SC/I
	I	*ReadShared*	SD/UD	SD*	SD/I
	UC	-	UC	-	-
	UD	-	UD	-	-
	SC	-	SC/UC	SC	SC/I
	SD	-	SD/UD	SD*	SD/I

mem si nécessaire, (ii) c'est une requête cohérente qui sera passée au $snoopFilter$ par le $systemMonitor$;
— Pour une requête cohérente, le $snoopFilter$ déterminera les transactions de *snooping* nécessaires. Si aucune transaction de *snooping* n'est envoyée par le $snoopFilter$, le $systemMonitor$ se charge d'envoyer cette requête au $memoryBuffer$;
— Chaque *master* qui reçoit la requête de *snooping* devra envoyer une réponse au *CCN-xx* ;
— Si un accès au $memoryBuffer$ est nécessaire, une fois la requête reçue par le $memoryBuffer$, ce dernier détermine s'il peut répondre à la requête. Dans le cas contraire, il envoie la requête à la mémoire *mem* ;
— la mémoire traite la requête en répondant avec la donnée, si nécessaire.

Actions et transitions pour chaque type de requête

Cette section présente, pour chaque type de requête, les actions et transitions définies dans le protocole ACE sous forme de tableaux. Les transitions sont déterminées pour le $L2$ ayant envoyé la requête et le $L2$ répondant à la requête. Nous utilisons les notations suivantes pour les états des lignes de cache : I (*Invalid*), UC (*Unique Clean*), UD (*Unique Dirty*), SC (*Shared Clean*) et SD (*Shared Dirty*).
Les tableaux sont structurés comme suit : le type de la requête, pour un $L2$ initiateur de la requête (l'état initial de la ligne de cache, la transaction de *snooping* correspondante, le nouvel état de la ligne de cache) et pour un $L2$ *snoopé* (l'état initial de la ligne de cache, le nouvel état de la ligne de cache).

A) *ReadShared* :
Pour une requête *ReadShared*, le $L2$ répond directement au *proc* si la ligne de cache correspondant à son adresse est valide, sinon une requête de *snooping ReadShared* est envoyée au *CCN-xx*. Dans ce dernier cas, si aucun des $L2$ ne peut répondre à la requête, la donnée sera cherchée au $memoryBuffer$ ou à la mémoire partagée *mem*, sinon une réponse sera envoyée par le *master snoopé* avec la donnée et une transition est effectuée comme indiqué dans le tableau 4.1.

B) *ReadUnique* :

Tableau 4.2 – Actions et transitions générées pour une requête *ReadUnique*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
ReadUnique	I	*ReadUnique*	*UC*	*UC*	I
	I	*ReadUnique*	*UD*/	*UD**	I
	I	*ReadUnique*	*UC*/*UC*	*SC*	I
	I	*ReadUnique*	*UD*	*SD**	I
	UC	-	*UC*	-	-
	UD	-	*UD*	-	-
	SC	*ReadUnique*/MakeInvalid	*UC*	*SC*	I
	SD	*ReadUnique*/MakeInvalid	*UD*	*SD**	I

Tableau 4.3 – Actions et transitions générées pour une requête *ReadNotSharedDirty*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
ReadNotSD	I	ReadNotSD	*SC*/*UC*	*UC*	*SC*/I
	I	ReadNotSD	*SC*/*UC*	*UD**	*SC*/I
	I	ReadNotSD	*SC*/*UC*	*SC*	*SC*/I
	I	ReadNotSD	*SC*/*UC*	*SD**	*SC*/I
	UC	-	*UC*	-	-
	UD	-	*UD*	-	-
	SC	-	*SC*/*UC*	*SC*	*SC*/I
	SD	*CleanShared*	*SC*/*UC*	*SD**	*SC*/I

Pour une requête *ReadUnique*, le *L2* répond directement au *proc* si la ligne de cache correspondant à son adresse est unique (*Dirty* ou *Clean*) sinon une requête de *snooping* *ReadUnique* est envoyée au *CCN-xx* pour effectuer l'opération de lecture et invalider toutes les autres copies du système. Les transitions effectuées sont indiqués dans le tableau 4.2. Si la *master* qui répond détient une copie en *Dirty*, il répond en envoyant la donnée et en passant le rôle de la mise à jour de la mémoire au *master* initiateur avant d'invalider.

C) *ReadNotSharedDirty* :
Si la ligne de cache est détenue en *Shared*, elle devra être *Clean* à la fin de l'opération *ReadNotSharedDirty*. (*) Si elle est détenue en *SharedDirty*, une requête *WriteClean* est envoyée au *CCN-xx* pour mettre à jour la mémoire. Le *CCN-xx* envoie une requête *CleanShared* aux autres *masters* pour nettoyer la donnée. Le tableau 4.3 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

D) *ReadClean* :
Une copie *Clean* de la donnée est demandée. (*) S'il existe en *Dirty* dans le système, la donnée est réécrite en mémoire immédiatement en envoyant une requête *WriteClean* ou *WriteBack*. Le *CCN-xx* envoie une requête *CleanShared* aux autres *masters* pour nettoyer la donnée si elle est partagée. Le tableau 4.4 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

Tableau 4.4 – Actions et transitions générées pour une requête *ReadClean*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
ReadClean	I	*ReadClean*	*SC*/*UC*	*UC*	*SC*/I
	I	*ReadClean*	*SC*/*UC*	*UD**	*SC*/I
	I	*ReadClean*	*SC*/*UC*	*SC*	*SC*/I
	I	*ReadClean*	*SC*/*UC*	*SD**	*SC*/I
	*UD**	-	*UC*	-	-
	SC	-	*SC*/*UC*	*SC*	*SC*/I
	SD	-*ReadClean*	*SC*/*UC*	*SD**	*SC*/I

Tableau 4.5 – Actions et transitions générées pour une requête *WriteUnique*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
WriteUnique	I	*CleanInvalid*	I	*UC*	I
	I	*CleanInvalid*	I	*UD**	I
	I	*CleanInvalid*	I	*SC*	I
	I	*CleanInvalid*	I	*SD**	I
	UC	-	*UC*	-	-
	SC	*CleanInvalid*	*UC*	*SC*	I

E) *WriteUnique* :
(*) Cette requête assure que toute ligne de cache dans l'état *Dirty* soit réécrite en mémoire avant d'effectuer l'opération d'écriture. Quand la réponse est reçue, la ligne de cache est mise à jour avec la nouvelle valeur. Le *snoopFilter* envoie une requête *CleanInvalid* à tous les *masters* détenant une copie de la ligne de cache. Si un des *masters* détient la donnée en *Dirty*, le *snoopFilter* initie une requête *WriteBack* avec la donnée reçue par un des *masters* snoopés. Le tableau 4.5 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

F) *WriteLineUnique* :
Cette requête informe tous les *masters* qui détiennent la ligne de cache que la ligne entière sera réécrite. Aucun *WriteBack* n'est effectué. Ils devront invalider leurs copies.

Tableau 4.6 – Actions et transitions générées pour une requête *WriteLineUnique*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
WriteLineU	I	MakeInvalid	I	*UC*	I
	I	MakeInvalid	I	*UD*	I
	I	MakeInvalid	I	*SC*	I
	I	MakeInvalid	I	*SD*	I
	UC	-	*UC*	-	-
	SC	MakeInvalid	*UC*	*SC*	I

Tableau 4.7 – Actions et transitions générées pour une requête *CleanUnique*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
CleanUnique	I	*CleanInvalid*	I	*UC*	I
	I	*CleanInvalid*	I	*UD**	I
	I	*CleanInvalid*	I	*SC*	I
	I	*CleanInvalid*	I	*SD**	I
	UC	-	*UC*	-	-
	UD	-	*UD*	-	-
	SC	*CleanInvalid*	*UC*	*SC*	I
	SD	*CleanInvalid*	*UC*	*SD**	I

Tableau 4.8 – Actions et transitions générées pour une requête *CleanInvalid*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
CleanInvalid	I	*CleanInvalid*	I	*UC*	I
	I	*CleanInvalid*	I	*UD**	I
	I	*CleanInvalid*	I	*SC*	I
	I	*CleanInvalid*	I	*SD**	I

snoopFilter initie une requête *MakeInvalid* pour invalider toutes les copies des autres caches. L'écriture est propagée à la mémoire et la ligne de cache est mise à jour avec la nouvelle valeur après avoir reçue la réponse pour la requête *WriteLineUnique*. Le tableau 4.6 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

G) *ReadOnce* :
La requête *ReadOnce* est une requête générée par les coprocesseurs. Elle est directement envoyée par le *systemMonitor* au *snoopMonitor* qui l'enverra, si besoin, aux *L2*. Aucune modification des états des lignes de cache n'est nécessaire. Le *master* détenant la ligne de cache répond en envoyant la donnée au *CCN-xx* qui répondra par la suite au coprocesseur.

H) *CleanUnique* :
Le *master* initiateur demande une copie unique de la ligne de cache en envoyant cette requête sur le canal *Read address channel*. Toutes les autres copies seront supprimées et toute donnée *Dirty* sera copiée en mémoire. Le tableau 4.7 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

I) *CleanInvalid* :
Cette requête est une opération de diffusion pour invalider et mettre à jour la mémoire. Après avoir invalidé sa copie, un *L2* initie cette opération dans le but d'invalider toutes les autres copies présentes dans les autres caches. Si la donnée existe en *Dirty*, un *WriteBack* est initié avant d'invalider la ligne de cache. Le tableau 4.8 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

J) *CleanShared* :

Tableau 4.9 – Actions et transitions générées pour une requête *CleanShared*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
CleanShared	I	*CleanShared*	I	*UC*/*SD**	*UC*/*SC*
	UC	*CleanShared*	*UC*	-	-
	SC	*CleanShared*	*SC*/*UC*	*SD**	*SC*/I

Tableau 4.10 – Actions et transitions générées pour une requête *MakeUnique with full store*.

Transaction	*master* initiateur			*master snoopé*	
	Etat initial	Requête de *snooping*	Etat final	Etat initial	Etat final
MakeUnique	I	*MakeUnique* and *store*	*UD*	*UC*	I
	I	*MakeUnique* and *store*	*UD*	*UD**	I
	I	*MakeUnique* and *store*	*UD*	*SC*	I
	I	*MakeUnique* and *store*	*UD*	*SD**	I
	UC	*store*	*UD*	-	-
	UD	*store*	*UD*	-	-
	SC	*MakeUnique* and *store*	*UD*	*SC*	I
	SD	*MakeUnique* and *store*	*UD*	*SD**	I

Cette requête est une opération de diffusion pour nettoyer les copies d'une ligne de cache partagée. Si le *master* initiateur détient une copie *Dirty*, une opération *WriteClean* est envoyée au *CCN-xx* de sorte que la ligne de cache soit en état *Clean* avant d'envoyer la transaction *CleanShared*. (**) Si le *master snoopé* détient une copie *Dirty* après avoir reçu la transaction *CleanShared*, la donnée est envoyée au *CCN-xx* pour être réécrite en mémoire. Le tableau 4.9 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

K) *MakeUnique* :
La transaction n'obtient pas une copie de la ligne de cache. Cette transaction est couplée avec une opération d'écriture. L'écriture n'est pas propagée à la mémoire. Si le *master snoopé* détient une donnée *Dirty*, il envoie la donnée pour réécriture en mémoire et invalide sa copie locale de ligne de cache. Le tableau 4.10 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

L) *WriteBack* :
L'opération *WriteBack* ne génère pas de requête de *snooping*. Après une mise à jour de la mémoire, l'état de la ligne de cache est invalide. Le tableau 4.11 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

M) *WriteClean* :
L'opération *WriteClean* ne génère pas de requête de *snooping*. Après une mise à jour de la mémoire, la ligne de cache reste allouée. Le tableau 4.12 illustre les différentes transitions suivant les états initiaux de la ligne de cache.
N) *Evict* :

Tableau 4.11 – Actions et transitions générées pour une requête *WriteClean*.

Transaction	*master* initiateur	
	Etat initial	Etat final
WriteBack	*UD*	I
	SD	I

Tableau 4.12 – Actions et transitions générées pour une requête *WriteClean*.

Transaction	*master* initiateur	
	Etat initial	Etat final
WriteClean	*UD*	*UC*
	SD	*SC*

Cette transaction indique la suppression d'une ligne de cache du cache local quand aucune mise à jour de la mémoire n'est nécessaire. Le tableau 4.13 illustre les différentes transitions suivant les états initiaux de la ligne de cache.

4.3.3 Flot de quelques transactions dans l'architecture générique proposée

ReadOnce

Considérons l'exemple décrit dans la figure 4.9.

(1) *coproc*0 émet une transaction *ReadOnce* vers l'adresse 0 sur le canal *AR* (*ReadAddressChannel*). La requête est envoyée au *CCN-xx*.

(2) *systemMonitor* sélectionne la requête et vérifie les règles de séquencement sur l'adresse 0. Deux cas se présentent :

— Aucune opération de modification de la donnée de l'adresse 0 n'est en cours : la requête est ajoutée à *reqList* ;

— Une opération de modification sur la donnée à l'adresse 0 est en cours : la requête ne sera pas prise en compte, une requête *retry* est envoyée à *coproc* qui va renvoyer la requête après un délai *d*.

(3) La requête est envoyée au *snoopFilter* qui vérifie si une copie de la donnée est présente dans l'un des caches du système. Deux cas possibles :

— Une copie existe : (4a) *snoopFilter* émet une requête de *snooping ReadOnce* au

Tableau 4.13 – Actions et transitions générées pour une requête *Evict*.

Transaction	*master* initiateur	
	Etat initial	Etat final
Evict	*UC*	I
	SC	I

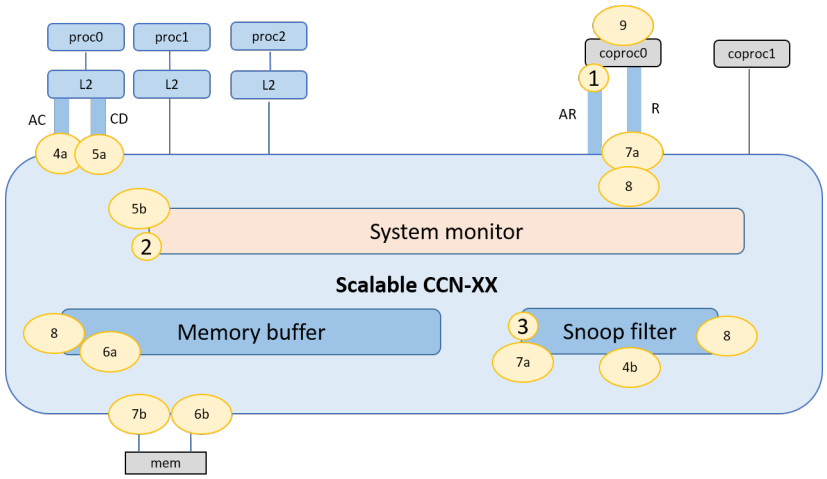

FIGURE 4.9 – Déroulement d'une transaction *ReadOnce* dans le système à base de *CCN-xx*.

cache $L2_0$ ayant le rôle de la mise à jour de la mémoire (seul un des caches répondra en envoyant la donnée si plusieurs ont cette donnée) ;
— (4b) Aucune copie n'existe dans les caches : $snoopFilter$ le signale au $systemMonitor$.

(5a) En réponse à (4a), $L2_0$ envoie la réponse au *CoreLink* sur le canal CD (*Coherency Data Channel*) avec la donnée sans modifier l'état de la ligne de cache.

(5b) $systemMonitor$ envoie la requête au $bufferMemory$ qui vérifiera l'existence de cette donnée. Deux cas possible :
— (6a) la donnée existe, elle est alors envoyée sur le *buffer* en sortie ;
— (6b) La donnée n'existe pas, une requête *ReadOnce* est est envoyée à la mémoire *mem*.

(7a) La donnée est reçue du $L2_0$, une réponse est initiée à *coproc* par $systemMonitor$ et l'état de la requête est mis à jour dans $reqList$.

(7b) La mémoire envoie la réponse.

(8) La réponse est reçue de la mémoire, dans ce cas là, $systemMonitor$ notifie $memoryBuffer$ pour sauvegarder une copie de la donnée, envoie la réponse au *coproc* et met à jour l'état de la requête dans $reqList$.

(9) *coproc* reçoit la réponse et termine la transaction. Une nouvelle requête peut alors être envoyée.

ReadUnique

Considérons l'exemple d'une requête $ReadUnique$ envoyée par *proc*0.

(1) *proc*0 émet une transaction $ReadUnique$ vers l'adresse 0 sur le canal AR (ReadAddressChannel). La requête est envoyée au $L2_0$.

(2) Le $L2_0$ reçoit la requête et vérifie s'il a la ligne de cache correspondant à l'adresse 0. Deux cas se présentent :

— Dans le cas d'un *hit* avec l'état UC ou UD, le $L2_0$ répond directement à *proc*0 sans modifier l'état de la ligne de cache et met en attente toute requête reçue sur cette ligne de cache jusqu'à la fin de traitement. Dans le cas d'un *hit* avec comme état de ligne de cache SC et SD, une copie unique de la donnée est demandée, les autres copies dans le système doivent être invalidées. Le cache envoie la requête $ReadUnique$ au *CCN-xx* ;
— Dans le cas d'un *miss*, une requête $ReadUnique$ est envoyée au *CCN-xx*.

(3) $systemMonitor$ sélectionne la requête et vérifie les règles de séquencement sur l'adresse 0. Deux cas se présentent :
— Aucune opération sur la donnée de l'adresse 0 n'est en cours : la requête est ajoutée à $reqList$;
— Une opération sur la donnée à l'adresse 0 est en cours : la requête ne sera pas prise en compte, une requête *retry* est envoyée à *proc*0 qui renverra la requête après un délai d.

(4) La requête est envoyée au $snoopFilter$ qui vérifie si une copie de la donnée est présente dans l'un des caches du système. Deux cas possibles :
— Une copie existe : (5a) $snoopFilter$ émet une requête de *snooping* $MakeInvalid$ à tous les caches détenant une copie de la donnée relative à l'adresse 0 ;
— (5b) Aucune copie n'existe dans les caches : $snoopFilter$ le signale au $systemMonitor$.

(6a) En réponse à (5a), considérons que les deux caches détiennent une copie de ligne de cache. $L2_0$ envoie la réponse au *CoreLink* sur le canal CD (*Coherency Data Channel*) avec la donnée, passe la responsabilité de mise à jour de la mémoire et invalide sa ligne de cache, le cache $L2_1$ et $L2_2$ invalident leur lignes de cache.

(6b) $systemMonitor$ envoie la requête au $bufferMemory$ qui vérifiera l'existence de cette donnée. Deux cas possibles :
— La donnée existe, elle est alors envoyée sur le *buffer* en sortie ;
— La donnée n'existe pas, une requête $ReadUnique$ est est envoyée à la mémoire *mem*.

(7a) La donnée est reçue du $L2_0$, une réponse est initiée à *proc* par $systemMonitor$, l'état de la requête est mis à jour dans $reqList$ et $snoopFilter$ est mis à jour.

(7b) La réponse est reçue de la mémoire, dans ce cas là, $systemMonitor$ notifie $memoryBuffer$ pour sauvegarder une copie de la donnée, envoie la réponse au *proc*0, met à jour l'état de la requête dans $reqList$ et met à jour $snoopFilter$.

(8)$L2_0$ alloue un bloc pour la donnée reçue si elle était invalide, sinon l'état de la ligne de cache est mis à jour à l'état $Unique$ et une réponse est envoyée à *proc*0.

(9) *proc*0 reçoit la réponse et termine la transaction. Une nouvelle requête peut alors être envoyée.

WriteLineUnique

Considérons l'exemple d'une requête d'écriture $WriteLineUnique$ où l'écriture est propagée vers la mémoire. L'utilisation de cette transaction est restreinte aux caches ne détenant pas la ligne en *Dirty*.

(1) *proc*0 émet une transaction $WriteLineUnique$ vers l'adresse 0 sur le canal W (*Write Address Channel*). La requête est envoyée au $L2_0$.

(2) Le $L2_0$ reçoit la requête et vérifie s'il a la ligne de cache correspondant à l'adresse 0. Plusieurs cas se présentent :

- Dans le cas d'un *hit* avec l'état UD ou SD, la transaction est rejetée comme indiquée dans la spécification ACE ;
- Dans le cas d'un *hit* avec l'état UC, le $L2_0$ répondra directement à *proc*0 en indiquant que la ligne de cache est valide et indiquera une requête en cours sur la ligne de cache ;
- Dans le cas d'un *hit* avec comme état de ligne de cache SC, les autres copies dans le système doivent être invalidées. Le cache envoie la requête $WriteLineUnique$ au *CCN-xx* ;
- Dans le cas d'un *miss*, une requête $WriteLineUnique$ est envoyée au *CCN-xx*.

(3) $systemMonitor$ sélectionne la requête et vérifie les règles de séquencement sur l'adresse 0. Deux cas se présentent :

- Aucune opération sur la donnée de l'adresse 0 n'est en cours : la requête est ajoutée à $reqList$;
- Une opération sur la donnée à l'adresse 0 est en cours : la requête ne sera pas prise en compte, une requête *retry* est envoyée à *proc*0, qui renverra la requête après un délai d.

(4) La requête est envoyée au $snoopFilter$, qui vérifie si une copie de la donnée est présente dans l'un des caches du système. Deux cas possibles :

- Une copie existe : (5a) $snoopFilter$ émet une requête de *snooping* $MakeInvalid$ à tous les caches détenant une copie de la donnée relative à l'adresse 0 ;
- (5b) Aucune copie n'existe dans les caches : $snoopFilter$ le signale au *systemMonitor*.

(6a) En réponse à (5a), considérons que les deux caches détiennent une copie de ligne de cache. Ils invalident leurs lignes de cache. (6b) $systemMonitor$ envoie la requête au $bufferMemory$ avec la donnée à écrire en mémoire. Une requête d'écriture est envoyée à la mémoire mem.

(7) en réponse à (6a) après invalidation des copies dans le cache et mise à jour de $snoopFilter$, $systemMonitor$ envoie la requête au $bufferMemory$ avec la donnée à écrire

en mémoire. Une requête d'écriture est envoyée à la mémoire *mem*.

(8) La donnée est écrite en mémoire.

(9) La réponse est envoyée au $L2_0$, qui mettra à jour sa ligne de cache avec la dernière valeur, et le $snoopFilter$ est mis à jour.

(10) *proc*0 reçoit la réponse et termine la transaction. Une nouvelle requête peut alors être envoyée.

4.4 Conclusion

Dans ce chapitre, nous avons présenté l'architecture générique et extensible conçue dans cette thèse. Nous avons montré l'importance de l'introduction du composant appelé moniteur système dans la gestion efficace du parallélisme dans les systèmes basés sur une telle architecture. Nous avons ensuite proposé une méthodologie pour la modélisation de chaque composant de notre système.
Pour une modélisation efficace et suffisante du protocole ACE sur l'architecture proposée, nous avons modélisé les caches comme entités comportant tous les états des données spécifiés dans le protocole ACE (I, UC, UD, SC et SD). Tous les cas de transitions présentés dans 4.3.2 ont été modélisés. Nous avons ensuite conclu par la description générale de l'exécution des requêtes ACE sur notre système suivi d'un exemple du processus d'exécution détaillée pour 3 types de requêtes, à savoir *ReadOnce*, *ReadUnique* et *WriteLineUnique*. Afin de vérifier l'efficacité de nos modèles, nous présentons la méthodologie suivie dans le chapitre 5.

Chapitre 5

Méthodologie de vérification

5.1 Modélisation et validation des systèmes informatiques

La modélisation est une étape importante dans la conception d'un système, vu la complexité et contraintes temps/développement de sa conception matérielle. Une conception incorrecte d'un système *hardware* complexe est très coûteuse, c'est pourquoi, les concepteurs ont recours à des techniques de vérification dès les premières phases du flot de conception. Plusieurs techniques existent. On retrouve des techniques basées sur les méthodes analytiques, la vérification formelle ou la simulation. Cette dernière a été retenue pour les travaux de cette thèse.
La modélisation d'un système consiste à spécifier un modèle qui reproduit son comportement. Ce modèle est une abstraction plus ou moins précise d'un système réel. En fonction du degré d'abstraction, on obtient un modèle plus ou moins fidèle. Dans ce qui suit, nous présentons les trois techniques de vérification citées ci-dessus.

5.1.1 Validation par méthodes analytiques

Les méthodes analytiques proposent d'étudier le comportement stationnaire d'un système en résolvant son modèle mathématique, ou plus précisément, en résolvant les équations mathématiques sous-jacentes à son modèle mathématique [Maria, 1997]. L'intérêt des méthodes analytiques réside principalement dans la résolution des équations généralement peu coûteuse en temps de calcul. Leur inconvénient est qu'il est généralement nécessaire de faire des hypothèses restrictives sur le système réel pour pouvoir obtenir des modèles exploitables.

5.1.2 Validation par vérification formelle

Il s'agit de méthodes basées sur des preuves mathématiques sur la validité de la conception d'un système [Swan, 2006]. La vérification formelle d'un système nécessite les descriptions formelles de la conception et de la spécification. En pratique, ces méthodes sont efficaces sur des systèmes à états finis de tailles moyennes. Dans ce qui suit, nous citons deux méthodes de vérification formelle.

Vérification formelle par l'algèbre des processus

Il s'agit d'une famille de formalismes proposée par la théorie de la concurrence. Un processus représente le comportement d'un système qui regroupe un ensemble d'évènements ou d'actions que ce système peut réaliser et l'ordre de leur exécution [Biemans and Blonk, 1986].
L'algèbre des processus fournit une description des interactions, des communications et des synchronisations entre un ensemble de composants indépendants du système concurrent. Des lois sont définies pour décrire, manipuler et analyser les différents processus.
Le langage LNT (*LOTOS New Technology*) est un langage de spécification moderne qui regroupe les caractéristiques de l'algèbre des processus et un ensemble de définitions des processus (*process* par exemple) et de définitions de types (*type* ou *endtype*) [Dwyer et al., 1999]. Dans [Park, 2007], plusieurs exemples sont donnés sur la vérification des systèmes numériques en utilisant l'algèbre des processus.

Vérification formelle par la preuve des propriétés

Cette technique consiste à construire un modèle M d'un système S, et à spécifier les exigences de performabilité que ce système doit satisfaire sous la forme de propriétés écrites en logique temporelle, avant de vérifier automatiquement, à l'aide d'un outil appelé un *model checker*, que ce modèle respecte ces propriétés [Clarke et al., 1999]. Lorsqu'on souhaite vérifier les propriétés du système, il est nécessaire de pouvoir les exprimer dans un langage adéquat. Il s'agit d'un outil $software$ qui, étant donnée une description d'un modèle M et une propriété P, décide si M satisfait P. Il retourne *true* si la propriété est satisfaite, sinon $false$ est retourné avec un contre exemple [Hu et al., 1997].
Dans ce qui suit, nous citons un exemple d'outil très utilisé dans la vérification des protocoles de cohérence de caches :

— $Murphi$ [Milne, 1993] est un langage de description formelle. Les spécifications en $Murphi$ sont des propriétés de $safety$ qui sont vérifiées par une exploration explicite de l'espace d'états. Une vue dite *white box* du système est utilisée. Les états de toutes les variables sont visibles durant la vérification. $Murphi$ est utilisé pour vérifier les protocoles de cohérence de caches dans les systèmes multiprocesseurs, où le but est de prouver la conformité de la conception du système par rapport à sa spécification, en termes de cohérence de cache, comme décrit dans [Clarke and Wing, 1996].

Ci-dessous, nous détaillons un exemple donné dans [Hachtel and Somenzi, 2006] : les machines à états finis (FSMs) sont établies pour chaque bloc du système. Les états sont déclarés en HDL comme des registres et les transitions sont représentées par des conditions entrainant l'évolution du système vers des états spécifiques, si elles sont validées. Par exemple, la mémoire partagée fonctionne en deux états : $busy = 0$ et $busy = 1$. Dans le premier état, la mémoire est libre et disponible à toute requête d'accès. Dans le second état, elle est occupée à servir un processeur et toute autre requête reçue à ce moment est mise en attente. Dans le cas des caches, deux FSMs sont associées. Elles retournent les états du cache et du module d'écoute (pour le protocole de *snooping*). Le rôle de cette dernière est de maintenir la cohérence des caches. Ce module d'écoute met à la disposition des autres modules l'état actuel du bloc (*valid*, *shared* ou *owned*).
L'état global du système regroupe toutes les instantiations des modules (processeurs,

mémoire partagée, bus,etc.). Sa FSM est donc une composition de leurs machines FSMs. Pour la vérification, des propriétés sont examinées selon un modèle donné. La technique employée dans [Hachtel and Somenzi, 2006] est *VSI-model checking*, les propriétés sont exprimées en CTL (*Computation Tree Logic*) et vérifiées avec l'outil VIS (*Verification Interacting with Synthesis tool*).

5.1.3 Validation par simulation

La simulation est une méthode de vérification dynamique. Grâce à la puissance de calcul des systèmes actuels, elle est considérée comme la solution priviligiée [Navaridas et al., 2011]. En simulation, les modèles peuvent être décrits avec des précisions variables, allant du niveau bit et du cycle d'horloge, qui est souvent un mécanisme lent, à des modèles plus abstraits, qui sont très souvent difficiles à définir. Cette difficulté est due à la prise en compte de tous les détails de l'architecture et de ses implications [Singhal and Engineer, 2008].
L'avantage de la simulation est d'offrir une approche très générale permettant d'étudier n'importe quel modèle, du moment que l'outil de simulation est adapté au modèle considéré. Néanmoins, les performances peuvent se dégrader en fonction de la qualité de l'implantation. L'implantation réalisée pour une modélisation d'un protocole de cohérence de cache, par exemple, doit être détaillée de sorte à gérer toutes les actions du protocole. Le nombre d'actions et de messages générés pour un même protocole peut différer d'une implémentation à une autre.
Dans ce travail, nous proposons une méthode d'évaluation haut niveau des protocoles de cohérence de cache de ARM ACE au niveau transactionnel.

5.1.4 Evaluation des systèmes : application aux systèmes hybrides et aux protocoles de cohérences de cache

Dans l'évaluation des protocoles de cohérence de caches et afin d'obtenir des résultats précis, il convient de modéliser entièrement tous les composants d'un système et leurs interconnexions. Les modélisations peuvent être à différents niveaux d'abstraction.
Ils existent plusieurs modélisations, que nous détaillerons dans ce qui suit.

Modélisation au niveau RTL (*Register-Tranfer Level*)

Il s'agit d'un modèle au niveau transfert de registre. La plupart du temps, il représente le modèle de référence pour la synthèse des circuits. Ces modèles constituent des points d'entrées du *design flow* d'un système et sont utilisés pour spécifier la logique utilisée pour la fabrication des *chips* physiques. Un modèle RTL peut être décrit dans un langage de description de matériel, comme VHDL ou Verilog.

Modélisation au niveau TLM (*Transaction-level Modeling*)

La notion de modélisation transactionnelle a récemment été introduite afin de désigner une représentation de l'architecture à un niveau de précision au dessus du niveau classique RTL. L'introduction du TLM dans le *flow* de conception a permis des détecter au plus tôt les problèmes relatifs à l'architecture et l'implémentation des systèmes. Des réductions

significatives des durées de conception des systèmes ont été réalisées. Nous utilisons cette technique dans nos développements.

Les langages de modélisation et environnements de simulation

Plusieurs langages peuvent être utilisés dans la modélisation des composants des systèmes et leur interconnexion. Nous en citons quelques uns sans être exhaustif.

— VHDL (*VHSIC Hardware Description Language*) : Le langage VHDL est un standard de description du matériel utilisé en électronique [Navabi, 1997]. Il couvre les niveaux d'abstraction allant du comportemental au logique. Contrairement aux descripteurs HDL auxquels il a succédé, il autorise plusieurs méthodologies de conception tout en étant d'un très haut niveau d'abstraction électronique Des bibliothèques de modèles VHDL aident à la modélisation et aux tests de systèmes complexes tels que les systèmes processeurs. De nombreux modèles sont développés et mis à disposition par la communauté VHDL ou par les vendeurs de circuits intégrés ;
— Lisa (*Description Language for hardware and Software*) : Il a été introduit en 1996 [Pees et al., 1999] et présenté comme un langage de description machine donnant des descriptions formelles des architectures programmables, des interfaces et des périphériques. Il permet la modélisation structurelle et comportementale généraliste du *hardware* et est utilisé pour décrire $l'ISA$ d'un processeur ;
— SystemC [Varga et al., 2001] : SystemC est un standard IEEE généraliste qui permet de développer des prototypages virtuels. Il consiste en une librairie de C++ qui fournit un ensemble de macros, de classes et un noyau de simulation à évènements discrets. Il a été développé en 1999 par ARM, Synopsys et Forte Design Systems en se basant sur Verilog, VHDL, SpecC et Scenic. L'un des avantages de SystemC est sa construction sous C++, ce qui permet de l'utiliser pour modéliser le *software* et le *hardware* des systèmes hybrides. Il est souvent utilisé pour les prototypage virtuel comme proposé, comme décrit dans [Dally and Towles, 2004b] ;
— OMNET++ [Rodrigues et al., 2011] : il s'agit d'un simulateur d'évènements discrets basé sur le langage C++. Ce simulateur est principalement destiné à la simulation des protocoles réseau et des systèmes distribués ;
— gem5 [GEM, 2015] : gem5 est une plate-forme de simulation à évènements discrets destinée à simuler des architectures de systèmes complexes. La simulation événementielle est basée sur la gestion d'une liste ordonnancée dans le temps. Elle comprend des évènements qui doivent intervenir dans le futur. Cette plate-forme de simulation est particulièrement intéressante. En effet, sa communauté ne cesse de s'agrandir et de plus en plus de travaux basés sur cet environnement sont proposés. Nous donnerons plus de détails sur gem5 dans 5.2. Pour une raison d'efficacité, nous avons choisi cet environnement pour simuler l'architecture proposée dans cette thèse.

5.2 Environnement de simulation gem5

gem5 est un environnement de simulation à évènements discrets. Il est basé sur les langages C++ et Python. Initialement, deux simulateurs appelés *GemS* et *M5* ont été

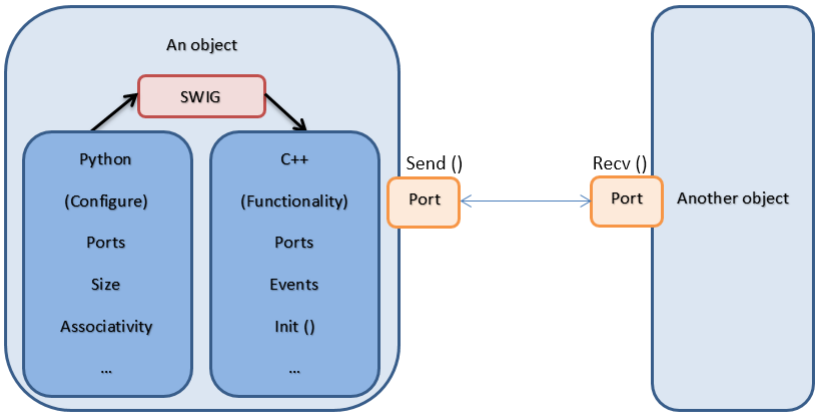

FIGURE 5.1 – Composition d'un modèle gem5.

conçus. Le simulateur *GEMS* (*General Execution-driven Multiprocessor Simulator*) permet la modélisation et la simulation détaillée des systèmes à base de multiprocesseurs [Martin et al., 2005]. Le simulateur *M5* permet la simulation de systèmes à base de réseaux d'interconnexion [Binkert et al., 2003]. Leur fusion a donné le simulateur gem5 [GEM, 2015].
Grâce à sa licence libre (*BSD*), plusieurs organismes de recherche et d'industrie l'ont utilisé dans leur travaux de recherche. Certains travaux, comme ceux présentés dans [Van Laer et al., 2013] et [Yang et al., 2012], permettent de faire évoluer cet outil en enrichissant sa bibliothèque par de nouveaux modèles de CPU, de mémoires, de bus, etc.

5.2.1 Structures du modèle gem5

gem5 accepte une combinaison de modèles au niveau instruction, ainsi que des modèles au niveau cycle. Un modèle dans *gem5* est une collection d'objets appelés *SimObject*. Les points d'entrée/sortie des objets sont les ports. Ils dérivent de la classe *Port* qui elle même dérive de la classe *SimObject*. Deux types de ports indiquant la direction des transferts sont définis : *MasterPort* qui génère une requête et *SlavePort* qui répond à une requête. Les objets *SimObjet* communiquent par l'envoi de paquets à travers des objets *Port*. Le paquet est une structure contenant des données abstraites définies par l'utilisateur.
Le comportement du système est décrit en C++. La configuration des objets est décrite en Python.
La figure 5.1 illustre un exemple d'un système de 2 objets modélisés sous gem5.

5.2.2 Modèle d'exécution

Un temps logique et global noté *tick* est utilisé dans gem5.
Les objets *SimObject* ordonnancent leurs propres évènements. Par exemple, un *CPU* ordonnance un évènement à des intervalles réguliers comme suit : à chaque cycle ou tous les n *picosecondes*. Quand un *SimObject* envoie un paquet par la fonction *send*(*pkt*), le noyau de simulation de gem5 prend en charge l'évènement qui lui est associé, il calcule le

temps d'arrivée de ce paquet dans le module destinataire en se basant sur la latence du lien et ordonnance les actions associées à ce paquet. En cas d'égalité de temps, il prend en compte la priorité du paquet. Si la priorité est la même il prend en compte l'instant où le paquet a été envoyé. Les premiers évènements sont produits lors de la phase d'instanciation des objets du système.
Les actions associées à un évènement sont décrites par des fonctions écrites en C++. L'action associée à un évènement noté e peut être la génération d'une nouvelle requête, par exemple.

5.2.3 Modes de simulation

Les modèles de *CPU* dans gem5 opèrent selon deux modes :
— FS *Full simulation* : où en plus de l'exécution des instructions de l'utilisateur, ils exécutent les instructions du noyau du système d'exploitation. Il permet également la modélisation des périphériques systèmes ;
— SE *System-call Emulation* : où toutes les fonctionnalités du système d'exploitation sont émulées en appelant *l'OS host*. Ce mode a été utilisé dans notre simulation.

La simulation en mode *FS* est très lente. C'est pourquoi, notre choix s'est porté sur une simulation en mode *SE*.

5.2.4 Quelques modèles prédéfinis

Certains modèles sont déjà prédéfinis dans gem5, on trouve :
— 4 modèles de CPU : *AtomicSimple* (un modèle à IPC unique), *TimingSimple* (qui inclut le temps des accès mémoire), *InOrder* (un CPU *pipeliné* et *in-order*), et *O3* (un CPU *pipeliné* *out-of-order*) ;
— Deux systèmes de mémoires : un système classique, qui est rapide et facilement configurable, et Ruby, pour l'étude de la cohérence de cache dans les systèmes à base de réseaux ;
— Un support des jeux d'intructions (ISAs) les plus populaires incluant ARM, ALPHA, MIPS, Power, SPARC and x86.

5.3 Implémentations dans gem5

Dans cette section, nous décrivons notre implémentation dans l'environnement de simulation gem5. Tout composant est modélisé comme étant un *SimObject*. Les classes que nous avons implémentées dérivent de cette classe. Les fonctions associées à chaque évènement sont décrites en C++ et les paramètres sont spécifiés en Python. Le rôle de l'outil *SWIG* utilisé dans gem5 est d'encapsuler du code C++ en Python. Il permet de configurer et d'exécuter des simulations par des scripts Python.

5.3.1 Interconnexion des composants

Les objets communiquent via des ports. Les ports sont de deux types : les ports *master* qui génèrent des requêtes et les ports *slave* qui répondent aux requêtes. Ils sont reliés par

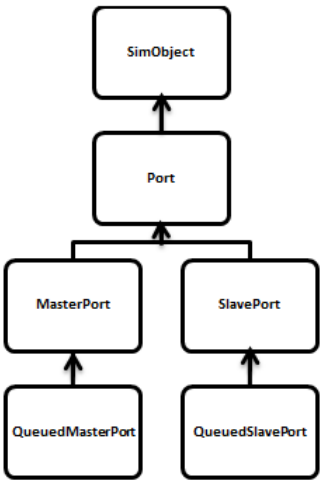

FIGURE 5.2 – Diagramme d'héritage de la classe *QueuedPort*.

la fonction prédéfinie *bind*(). La classe des ports que nous avons utilisée dérive de la classe *SlavePort* pour les ports *slave* et *MasterPort* pour les ports *master*. Une FIFO leur est associée. Le diagramme d'héritage est décrit dans la figure 5.2.
La construction des ports se fait comme suit :

QueuedMasterPort(const std : :string &name, SimObject owner, MasterPacketQueue &queue, PortID id = InvalidPortID) : MasterPort(name, owner, id), queue(queue)*

QueuedSlavePort(const std : :string& name, SimObject owner, SlavePacketQueue &queue, PortID id = InvalidPortID) : SlavePort(name, owner, id), queue(queue)*

Les canaux entre chaque paire de composants sont construits comme suit :

Layer(DstType& dport, SrcType& sport, const std : :string& name);

5.3.2 Echange des données

Chaque accès à la mémoire est un paquet envoyé entre les composants. La construction du paquet se fait comme suit :

*Packet(Request *req, MemCmd cmd, int blkSize) : cmd(cmd), req(req), data(nullptr), addr(0), src(PortID), dest(PortID)*

Quand la fonction *send* est appelée au niveau du port *master*, une fonction *recv* est appelée au niveau du port *slave* auquel il est connecté. Ces fonctions seront détaillées dans la suite de cette section, et ce pour chaque objet. La figure 5.3 illustre le flot d'une transaction via les ports de notre système.

La plateforme de simulation gem5 est *event driven*. Un évènement est associé à chaque envoi de paquet (requête ou réponse) et une fonction *process*() décrit les actions de cet

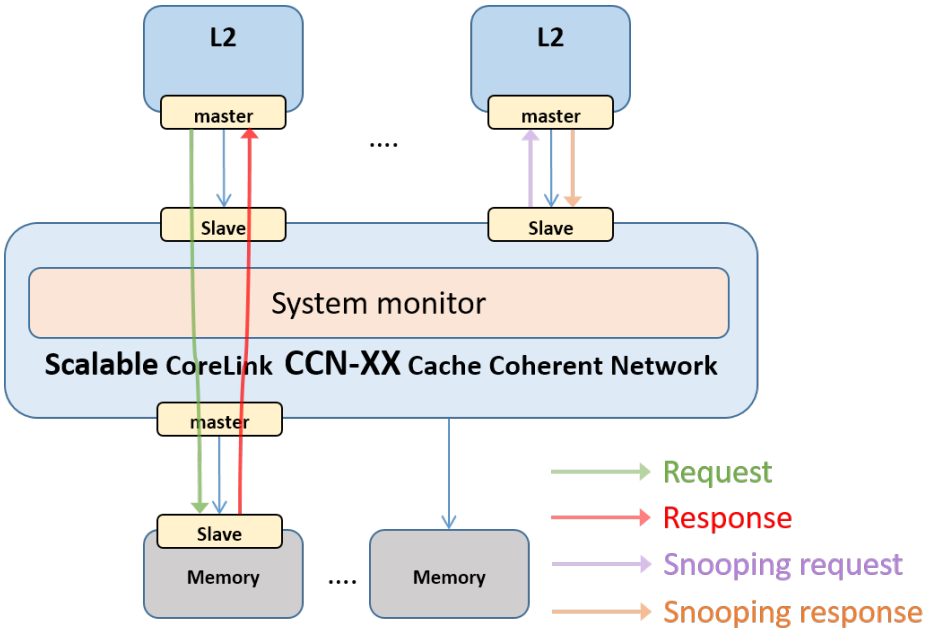

FIGURE 5.3 – Flot des transactions via les ports.

évènement une fois déclenché. Le simulateur détient une liste des évènements appelée *eventQueue*, qui seront déclenchés au cours de la simulation. Les premiers évènements sont générés lors de la phase d'instantiation des objets du système modélisé.
La configuration de chaque objet est décrite dans un fichier Python auquel est associé un fichier en C++ pour la description des fonctions de chaque objet. Le lien est fait entre ces deux fichiers avec l'outil *SWIG*. Le schéma 5.4 décrit un diagramme des composants dans gem5.

5.3.3 Les composants *proc* et *coproc*

Comme décrit dans la section 4.1.1, le rôle du composant *proc* est d'imiter le comportement d'un processeur. Les fonctions implémentées décrivent l'envoi aléatoire des requêtes avec la fonction *sendReq*, le renvoi des requêtes avec la fonction *doRetry* et l'envoi de l'accusé de réception *ack* pour terminer une transaction avec la fonction *completeRequest*. Les requêtes de ce composant sont envoyées à son cache local *L2*.
Le composant *coproc* renvoie quant à lui les requêtes directement au réseau d'interconnexion. Le diagramme d'héritage est décrit dans la figure 5.5.

a) Les structures :
Deux classes *proc* et *coproc* sont implémentées. Elles dérivent de l'objet de base de gem5 qui est *simObject*. Dans cette classe sont définis les ports de ce composant, la taille de la mémoire à laquelle accéder, les évènements générés par chacune de ses fonctions et la fonction *process*() de chaque évènement. Cette fonction décrit les actions qui lui sont associées.

b) Les connexions :
Ces composants sont reliés aux caches locaux dans le cas de *proc* et au *CCN-xx* dans le

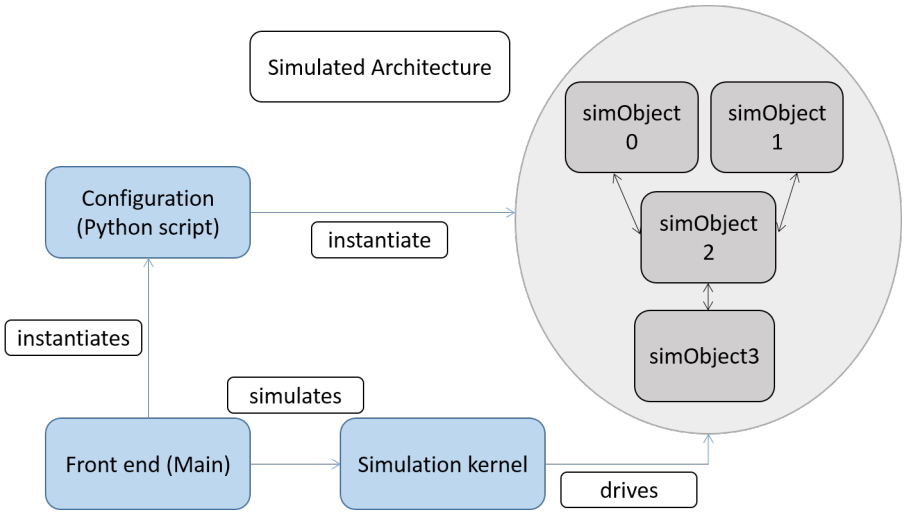

FIGURE 5.4 – Diagramme des composants de gem5.

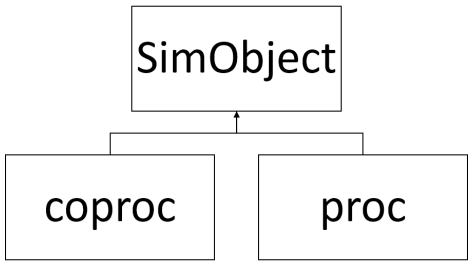

FIGURE 5.5 – Diagramme d'héritage des classes *proc* et *coproc*.

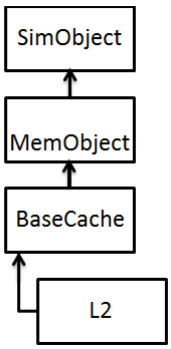

FIGURE 5.6 – Diagramme d'héritage de la classe *L2*.

cas de *coproc*, comme illustré dans la figure 4.1. Nous avons nommé les ports comme suit.

— *proc* : il est relié au cache avec le port *master cachePort* ;
— *coproc* : il est relié au *CoreLink* avec le port *master memSidePort*. Ce port se compose d'un ensemble de canaux virtuels (*AR*, *R*, *AW*, *W* et *B*).

c) Les actions :
Une fonction, notée *init*(), initialisant tous les paramètres des composants *proc* et *coproc* est d'abord définie. Elle génère le premier évènement de l'objet.
Une fonction itérative appelée *tick*() génère aléatoirement des requêtes, de type dépendant du pourcentage défini pour chaque type de requête ACE. L'adresse d'accès est calculée aléatoirement dans une plage d'adresses fixée par l'utilisateur. Le type de la transaction est généré selon le type de *master* (processeur ou coprocesseur)). L'évènement déclenchant cette fonction itérative est noté *TickEvent*.
Les fonctions implémentées pour décrire les différentes actions de *proc* et *coproc* sont *sendReq*, *recvResp recvRetry*, *doRetry* et *completeReq*. Ces actions sont détaillées dans 4.1.1.

5.3.4 Le composant *L2*

Le composant *L2* hérite de la classe *baseCache* de gem5. Il reprend les fonctions de base d'un cache comme les fonctions de remplacement, d'allocation, d'invalidation d'un bloc, etc.
Le diagramme d'héritage est décrit dans la figure 5.6.

a) Les structures :
Le cache *L2* est défini comme étant un ensemble de blocs correspondant (lignes de cache). Chaque ligne de cache contient les informations décrites ci-dessous :
structure(bloc, desc="Cache entry")
Address Address, desc="Physical address for this bloc";
State blkState, desc="state";
DataBlock DataBlk, desc=" buffer for the data bloc";
bool inProgress, default="false", desc="a request is in progress for this bloc";
bool Dirty, default="false", desc="indicator to update memory";

b) Les connexions :
Deux types de ports sont définis pour cette classe :

- *procSidePort* : un port *slave* pour se connecter à un *proc* ;
- *memSidePort* : un port *master* pour se connecter au *CCN-xx*. Il contient une file des requêtes envoyées en sortie et se compose des canaux virtuels suivants : AR, R, AW, W, B, AC, CR et CD.

c) Les actions :
Le comportement du composant $L2$ est décrit algorithmiquement dans 4.1.2. La classe définie reprend les fonctions de base du composant *baseCache* implémenté dans gem5. A titre d'exemple, l'algorithme LRU (*Least Recently Used*) est utilisé pour le remplacement des lignes de caches. Les fonctions implémentées du $L2$ sont les suivantes :

- *recvReq* pour la réception d'une requête venant d'un *master* ;
- *recvSnoopingReq* pour la réception d'une requête venant du *CCN-xx* (requête de *snooping*) ;
- *recvResp* pour la réception d'une réponse venant du *CCN-xx* ;
- *recvRetry* pour la réception d'une requête de renvoi ;
- *sendReq* pour l'envoi d'une requête au *CCN-xx* ;
- *sendSnoopingResp* pour la réception d'une requête. Le cache accède au bloc correspondant à l'adresse.

Lors de la réception d'une réponse, un bloc est alloué s'il est invalide, son état est mis à jour.

5.3.5 Le composant *snoopFilter*

SnoopFilter fait partie du composant CCN qui est un *simObject* de gem5.

a) Les structures :
Ce composant est une classe contenant : une liste de ports *slave* par ligne de cache, l'état de la ligne de cache et son adresse. Un entrée de table implémentée dans *snoopFilter* est déclarée comme suit :

struct SnoopItem { Addr addr, SnoopMask SnoopList, SnoopMask holderId, State status}

b) Les connexions :
Nous avons défini les ports suivants : Un vecteur de ports *snoopSidePorts* pour se connecter aux caches $L2$ et un port noté *monitorSidePort* pour se connecter au *systemMonitor*.

c) Les actions :
Nous avons implémenté les algorithmes décrit dans 4.1.4 comme suit :

- Une fonction *lookup* pour vérifier si une copie d'une ligne de cache donnée existe dans le système. Elle permet de la localiser ;
- Une fonction *snoopSelected* pour *snooper* le cache sélectionné (envoi d'une requête de *snooping*) ;

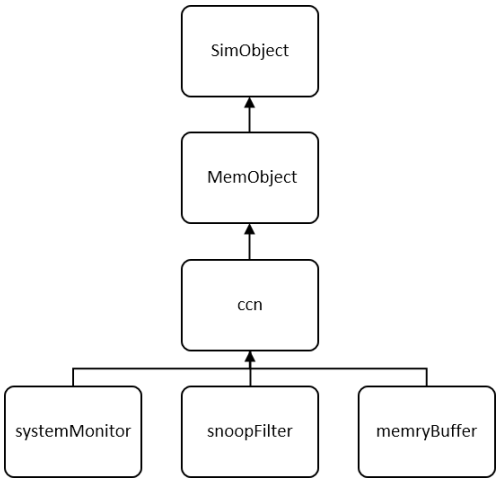

FIGURE 5.7 – Diagramme d'héritage de la classe *CCN-xx*.

— Une fonction *snoopAll* pour diffuser une requête donnée ;
— Une fonction *updateCacheLine* pour mettre à jour l'état d'une ligne de cache et sa localisation ;
— Une fonction pour envoyer la réponse au *master* initiateur après une demande du $systemMonitor$.

5.3.6 Le composant *systemMonitor*

systemMonitor permet de gérer les accès au *CCN-xx*. Il fait partie de *CCN-xx* tout comme $snoopFilter$ et $memoryBuffer$.

a) Les structures :
Il contient un tableau *reqList* qui représente une liste contenant les requêtes d'accès entrant au *CCN-xx* comme décrit dans la section 4.1.4.

struct transItem {PacketPtr pkt, TransState state}

b) Les connexions :
Il est interconnecté au $snoopFilter$ via le port $snoopFilterSidePort$ et au $memoryBuffer$ via le port $bufferSidePort$.

c) Les actions :
Comme détaillé dans la section 4.1.4, les deux tâches principales sont *trackRequests* qui a pour rôle de suivre les requêtes et les réponses entrant au *CCN-xx* et *processRequests* qui a pour rôle d'envoyer les requêtes pour exécution au $snoopFilter$ et $memoryBuffer$. Il se charge d'envoyer la réponse au *master* initiateur de la requête via $snoopFilter$ en utilisant la fonction *sendResp*. La fonction $updateSnoopFilter$ met à jour le $snoopFilter$. La fonction $updateMemoryBuffer$ met à jour le $memoryBuffer$.

Le diagramme d'héritage est décrit dans la figure 5.7.

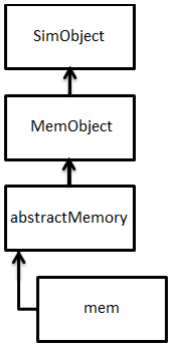

FIGURE 5.8 – Diagramme d'héritage de la classe *mem*.

5.3.7 Le composant *memoryBuffer*

memoryBuffer permet de sauvegarder les données lues ou écrites dans le système. Il fait partie de *CCN-xx*.

a) Les structures :
La classe $memoryBuffer$ est un $SimObject$. Il s'agit d'une classe contenant une liste de blocs (adresse, donnée) et implémentant plusieurs fonctions, comme décrit dans 4.1.4.

b) Les connexions :
Deux ports sont définis : $monitorSidePort$ pour se connecter au $systemMonitor$ et un vecteur $memSidePorts$ pour se connecter aux mémoires.

c) Les actions :
Les fonctions suivantes sont implémentées :
- $access$ pour accéder au bloc d'une adresse donnée ;
- $sendReq$ pour envoyer une requête à la mémoire ;
- $allocateBlock$ pour allouer un bloc ;
- $victimBlock$ pour sélectionner un bloc à remplacer, sa taille limitée étant limitée.

5.3.8 Le composant *mem*

Le composant *mem* hérite d'une classe $abstractMemory$ de gem5. Le diagramme d'héritage est décrit dans la figure 5.8.
La définition des paramètres *latency* et *bandwidth* se fait dans le fichier de configuration python $Mem.py$.

a) Les structures :
mem contient une file en entrée, notée $packetQueue$, qui contiendra les requêtes en attente de traitement. Elle se compose d'un ensemble de blocs de données. Sa taille est définie par l'utilisateur.

b) Les connexions :
Un port *slave* $memoryPort$ relie la mémoire au *CCN-xx*.

c) Les actions :
La réception d'une requête se fait par la fonction *recvReq*. Une fonction d'accès *access* décrit l'accès à l'adresse demandée. Si c'est une requête d'écriture, la donnée est copiée à l'adresse indiquée dans la requête. Si c'est une requête de lecture, la donnée de l'adresse d'accès est copiée dans le champs *data* du paquet à envoyer. Si la requête nécessite une réponse, elle sera envoyée en utilisant la fonction *makeResponse*. Le comportement implémenté est décrit algorithmiquement dans 4.1.3.

5.4 Conclusion

Dans la première partie de ce chapitre, nous avons introduit la vérification des systèmes informatiques en présentant les trois techniques les plus connues pour leur validation, à savoir la validation par méthodes analytiques, par vérification formelle et par simulation. Après analyse des différentes méthodes, nous avons retenu la simulation comme méthodologie de vérification de nos modèles. Cette approche a été adoptée pour des considérations d'efficacité et de performance.
La vérification étant une étape importante dans la conception et la validation des architectures, plusieurs outils de modélisation existent. Dans la deuxième partie, nous en avons décrit quelques uns, dont l'environnement de simulation gem5. Après analyse des différents environnements existants pour la modélisation et la simulation, nous avons retenu gem5 pour des raisons d'efficacité.
La troisième partie a été consacrée aux détails des implémentations de nos différents modèles dans gem5. Les résultats de la vérification de notre architecture sont présentés dans le chapitre 6.

Chapitre 6

Description des tests et analyse des résultats

Ce chapitre décrit le processus de vérification et d'évaluation du protocole AMBA ACE sur le système générique proposé. En premier lieu, nous avons effectué des tests pour vérifier l'aspect fonctionnel de l'architecture. La démarche suivie consiste entre autres à analyser les états des lignes des caches à tout instant pour vérifier la cohérence des données dans ces caches et leur consistance par rapport à la mémoire. En second lieu, quelques expérimentations sont présentées pour mieux comprendre le comportement de nos simulations et analyser l'efficacité de notre approche de modélisation.

6.1 Génération des *patterns* de test

Dans le processus de validation de l'implémentation du protocole AMBA ACE sur l'architecture générique proposée, des testeurs aléatoires ont été utilisés, comme décrit dans la section 4.1.1. Ces testeurs génèrent des requêtes d'accès (en lecture/écriture) à des adresses aléatoires de l'espace d'adressage avec un taux de recouvrement variable. Afin de vérifier tout le protocole, des requêtes d'accès parallèle de tous les types de requêtes définies dans le protocole AMBA ACE sont générées.

6.2 Paramétrage des tests

Dans les tests effectués, nous faisons varier certains paramètres afin d'étudier l'impact de chacun sur l'exécution des requêtes et le temps de simulation. Les paramètres de configuration sont les suivants :

- Le nombre de *proc* et *coproc* : il est paramétrisable. Nous avons effectué de tests de 4 à 650 processeurs et coprocesseurs ;
- Le nombre de caches $L2$: il est égal au nombre de processeurs. Chaque processeur est attaché à un cache $L2$;
- La taille des caches : nous l'avons fait varier de 16KB à 8MB. Les caches de petite taille permettent de mieux tester certains types de requêtes, comme la requête *Evict* ;
- L'associativité de $L2$: elle est fixée à 4 ;

— La taille d'une ligne de cache est fixée à 64 *bytes* ;
— Le pourcentage de génération de chaque type de requête est variable ;
— Le nombre maximal de requêtes à générer par processeur/coprocesseur ;
— Le délai de *retry* est fixé à 5ns ;
— Le temps maximal de réponse à une requête noté $tmax$. Au delà de cette valeur, la requête est ignorée ;
— La taille maximale de la table $reqList$ est fixée à 400 transactions ;
— La taille de $memoryBuffer$ est fixée à 16MB ;
— Les latences de réponse des $L2$ et du $snoopFilter$ sont fixées à 1 ns ;
— La latencede réponse du $memoryBuffer$ est fixée à 8 ns et celle de mem à 100 ns ;
— La latence des *snooping* varie entre 1 et 8 ns ;
— La taille de la mémoire principale mem varie entre 32 et 256MB et le débit est fixé à 12GB/s.
— La durée d'un cycle de $systemMonitor$ est fixée à 1ns.

6.3 Métriques d'évaluation

Pour analyser les performances de notre modélisation, nous analysons les paramètres suivants :

— Évolution du temps moyen d'exécution d'une transaction en fonction du taux de recouvrement des opérations ;
— La variation du temps moyen d'exécution d'une transaction en fonction du nombre de processeurs et coprocesseurs ;
— La variation de la durée de simulation en fonction de la taille du système simulé ;
— La variation du nombre de *retry* en fonction de la taille de $reqList$ et en fonction de la taille du système ;
— La variation de l'occupation de la table $reqList$ en fonction de la taille du système.

6.4 Résultats

6.4.1 Vérification de l'implémentation du protocole ACE sur l'architecture générique proposée

Une trace de simulation est présentée dans dans ce qui suit. La configuration du test effectué consiste en un système de 2 processeurs et un coprocesseur. Le tableau 6.1 résume la configuration du système simulé.

```
gem5 Simulator System.  http://gem5.org
gem5 compiled Oct  12 2016 14:28:47
gem5 started Oct  12 2016 14:30:45
Global frequency set at 1000000000 ticks per second
info: Entering event queue @ 0.  Starting simulation...
      0: system.physmem.cache0.proc: id 0 initiating readShared at
addr 401480 (blk 401480) expecting 0
      0: system.physmem.coproc: id 1  initiating readOnce at addr
80b5c0 (blk 80b5c0) expecting 0
      0: system.physmem.coproc.mem_side: getTimingPacket ReadOnce
uncacheable for address 80b5c0 size 1
      1: system.physmem.cache0: access for ReadSharedReq address
401480 size 1
      1: system.physmem.cache0: ReadSharedReq 401480 (ns) miss
      1: system.physmem.cache0.mem_side: Asserting bus request for
cause 0
      1: system.physmem.cpu_side_ccn.system_monitor: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[1] ReadOnceReq expr 0 0x80b5c0
      1: system.physmem.cpu_side_ccn.system_monitor: copying for ccn
pkt to global pkt: pkt ReadOnceReq reqList ReadOnceReq state 4
(1)   1: system.physmem.cache2.proc: id 2 initiating readUnique at
addr 401480 (blk 401480) expecting 0
      2: system.physmem.cpu_side_ccn.system_monitor: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[0] ReadSharedReq expr 0
0x401480
      2: system.physmem.cpu_side_ccn.system_monitor: copying for ccn
pkt to global pkt: pkt ReadSharedReq reqList ReadSharedReq state 4
      2: system.physmem.cpu_side_ccn.snoop_filter: recvTimingReq: src
system.physmem.cpu_side_ccn.slave[1] ReadOnceReq 0x80b5c0 SF size: 0
lat: 1
      2: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
packet src system.physmem.cpu_side_ccn.slave[1] addr 0x80b5c0 cmd
ReadOnceReq
      2: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
SF value 0.0
      2: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
new SF value 1.0
      2: system.physmem.cpu_side_ccn.snoop_filter: Resp:
forwardTiming for ReadOnceReq address 0x80b5c0 size 64
(2)   2: system.physmem.cache2: access for ReadUniqueReq address
401480 size 1
      2: system.physmem.cache2: ReadUniqueReq 401480 (ns) miss
      2: system.physmem.cache2.mem_side: Asserting bus request for
cause 0
      3: system.physmem.cpu_side_ccn.snoop_filter: recvTimingReq: src
system.physmem.cpu_side_ccn.slave[0] ReadSharedReq 0x401480 SF size:
0 lat: 1
      3: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
packet src system.physmem.cpu_side_ccn.slave[0] addr 0x401480 cmd
ReadSharedReq
      3: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
SF value 0.0
      3: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
new SF value 1.0
      3: system.physmem.cpu_side_ccn.snoop_filter: Resp:
forwardTiming for ReadSharedReq address 401480 size 64
        3: system.physmem.cpu_side_ccn.system_monitor: ReadOnceReq
address 80b5c0 size 64 state 2
(3)   3: system.physmem.cpu_side_ccn.system_monitor: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[2] ReadUniqueReq expr 0
0x401480
      3: system.physmem.cpu_side_ccn.system_monitor: ReadUniqueReq
address 401480 busy send a retryReq
      4: system.physmem.cpu_side_ccn.system_monitor: ReadSharedReq
address 401480 size 64 state 2
      4: system.physmem.cpu_side_ccn.memory_buffer: recvTimingReq:
```

```
src system.physmem.cpu_side_ccn.slave[1] ReadOnceReq 0x80b5c0 SF
size: 0 lat: 1
      4: system.physmem.cpu_side_ccn.memory_buffer: access for
ReadOnceReq address 80b5c0 size 1
      4: system.physmem.cpu_side_ccn.memory_buffer: ReadOnceReq
80b5c0 (ns) invalid
(4)   4: system.physmem.cpu_side_ccn.cache2: received a retry
      4: system.physmem.cpu_side_ccn.cache2: forwarding retryReq: src
system.physmem.cpu_side_ccn.slave[2] ReadUniqueReq 0x401480 SF size:
0 lat: 1
      5: system.physmem.cpu_side_ccn.memory_buffer: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[0] ReadSharedReq 0x401480 SF
size: 0 lat: 1
      5: system.physmem.cpu_side_ccn.memory_buffer: access for
ReadSharedReq address 401480 size 1
      5: system.physmem.cpu_side_ccn.memory_buffer: ReadSharedReq
401480 (ns) invalid
      5: system.physmem.cpu_side_ccn.memoryBuffer: forwardTiming
memory for ReadOnceReq address 80b5c0 size 64
(5)   5: system.physmem.cache2.proc: received a retry for readUnique
at addr 401480 (blk 401480) value cb
      6: system.physmem.cpu_side_ccn.memoryBuffer: forwardTiming
memory for ReadSharedReq address 401480 size 64
      6: system.mem: ReadOnce from physmem.coproc of size 64 on
address 0x80b5c0 C
      6: system.mem: 00000000  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      6: system.mem: 00000010  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      6: system.mem: 00000020  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      6: system.mem: 00000030  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      7: system.mem: ReadShared from physmem.cache0.proc of size 64
on address 0x401480 C
      7: system.mem: 00000000  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      7: system.mem: 00000010  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      7: system.mem: 00000020  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      7: system.mem: 00000030  00 00 00 00 00 00 00 00  00 00 00 00
00 00 00 00
      106: system.physmem.cpu_side_ccn.system_monitor: ReadOnceReq
address 80b5c0 size 64 state 2
      107: system.physmem.cpu_side_ccn.system_monitor: ReadSharedReq
address 401480 size 64 state 6
      107: system.physmem.cpu_side_ccn.memory_buffer: updateResponse:
Block for addr 80b5c0 being updated
      107: system.physmem.cpu_side_ccn.system_monitor: updateReqList:
ReadOnceReq 0x80b5c0 completed state 1
      108: system.physmem.cpu_side_ccn.snoop_filter: updateResponse:
old SF value 1.0
      108: system.physmem.cpu_side_ccn.snoop_filter: updateResponse:
new SF value 0.1 state UC id 0
      108: system.physmem.cpu_side_ccn.memory_buffer: updateResponse:
Block for addr 401480 being updated
        108: system.physmem.coproc: completing read at address 80b5c0
(blk 80b5c0) success
      109: system.physmem.cpu_side_ccn.system_monitor: updateReqList:
ReadSharedReq 0x401480 completed state 1
      109: system.physmem.cache0: Handling response to ReadResp for
address 401480 (ns)
      109: system.physmem.cache0: Block for addr 401480 being updated
in Cache
      109: system.physmem.cache0: Block addr 401480 (ns) moving from
```

```
state 0 to state: 7 (UC) valid: 1 writable: 1 readable: 1 dirty: 0
tag: 401
      109: system.physmem.cache0.proc_side: SendTiming for
ReadSharedResp address 401480 size 1
      109: system.physmem.cache0: Leaving recvTimingResp with
ReadResp for address 401480
(6)   109: system.physmem.cache2.proc: sendRetryReq readUnique at
addr 401480 (blk 401480) value cb
        110: system.physmem.cache0.proc: completing read at address
401480 (blk 401480) success
      110: system.physmem.cache2: access for ReadUniqueReq address
401480 size 1
      110: system.physmem.cache2: ReadUniqueReq 401480 (ns) miss
      110: system.physmem.cache2.mem_side: Asserting bus request for
cause 0
      111: system.physmem.cpu_side_ccn.system_monitor: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[2] ReadUniqueReq expr 0
0x401480
      111: system.physmem.cpu_side_ccn.system_monitor: copying for
ccn pkt to global pkt: pkt ReadUniqueReq reqList ReadUniqueReq state
4
      112: system.physmem.cpu_side_ccn.snoop_filter: recvTimingReq:
src system.physmem.cpu_side_ccn.slave[2] ReadUniqueReq 0x401480 SF
size: 0 lat: 1
      112: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
SF value 0.1
      112: system.physmem.cpu_side_ccn.snoop_filter: lookupRequest:
new SF value 1.1
      112: system.physmem.cpu_side_ccn.snoop_filter: Resp :
forwardTiming for ReadUniqueReq address 401480 size 64
      114: system.physmem.cache0: recvTimingSnoopReq for
ReadUniqueReq address 401480 size 64
      114: system.physmem.cache0: handleSnoop for ReadUniqueReq
address 401480 size 64
      114: system.physmem.cache0: Block for addr 401480 being updated
in Cache
      114: system.physmem.cache0: Block addr 401480 (ns) moving from
state 7 to state: 0 (I) valid: 0 writable: 0 readable: 0 dirty: 0
tag: 400
      116: system.physmem.cache0: sendTimingResp with ReadUniqueResp
for address 401480
      117: system.physmem.cpu_side_ccn.system_monitor:
recvTimingResp: src system.physmem.cpu_side_ccn.master[0]
ReadUniqueResp 0x401480
      117: system.physmem.cpu_side_ccn.system_monitor: ReadUniqueReq
address 401480 size 64 state 4
      117: system.physmem.cpu_side_ccn.snoop_filter: updateResponse:
old SF value 1.0
      117: system.physmem.cpu_side_ccn.snoop_filter: updateResponse:
new SF value 0.2 state UC id 2
      118: system.physmem.cpu_side_ccn.system_monitor: updateReqList:
ReadUniqueReq 0x401480 completed state 1
      118: system.physmem.cache2: Handling response to ReadUniqueResp
for address 401480 (ns)
      118: system.physmem.cache2: Block for addr 401480 being updated
in Cache
      118: system.physmem.cache2: Block addr 401480 (ns) moving from
state 0 to state: 7 (UC) valid: 1 writable: 1 readable: 1 dirty: 0
tag: 401
      118: system.physmem.cache2.proc_side: SendTiming for
ReadUniqueResp address 401480 size 1
      118: system.physmem.cache2: Leaving recvTimingResp with
ReadUniqueResp for address 401480
      119: system.physmem.cache2.proc: completing readUnique at
address 401480 (blk 401480) success
Exiting @ tick 18446744073709551615 because simulate() limit reached
```

Tableau 6.1 – Paramètres de configuration du test 1.

Paramètre	valeur
Nombre de *masters*	3
Taille des caches	256KB
Taille d'une ligne de cache	64 *bytes*
Nombre de requêtes/*master*	1
Délai de *retry*	109ns.
Temps maximal de réponse	800ns
Taille de $memoryBuffer$	1MB
Latence de réponse de $L2$	1ns
Latence d'un *hit* de $L2$	1ns
Latence de réponse de $snoopFilter$	1ns
Latence de réponse de $memoryBuffer$	1ns
Latence de réponse de mem	100ns
Latence des *snooping*	1ns
Taille de la mémoire	64MB

Le processus d'exécution illustré dans la trace est comme suit : le processeur *proc*0 émet une requête d'accès *ReadShared* vers l'adresse 401480 au *tick*= 0, le coprocesseur *coproc*1 émet une requête d'accès *ReadOnce* vers l'adresse 80b5c0 au *tick*=0 et le processeur *proc*2 émet une requête d'accès *ReadUnique* vers l'adresse 401480 au *tick*=1.
La trace de simulation est structurée comme suit : l'instant auquel survient un évènement, le composant concerné par l'opération et le type d'opération.
Nous expliquons, dans ce qui suit, le déroulement de la simulation pour le *proc*2. Nous avons noté en rouge les étapes détaillées ci-dessous :

— (1) *proc*2 envoie une requête *ReadUnique* à *tick*= 1 ;
— (2) Le cache local $L2$ reçoit la requête qu'il transfère au *CCN-xx* après un *miss* à *tick*= 2 ;
— (3) $systemMonitor$ constate qu'une requête est en cours sur l'adresse 401480. Il envoie alors une requête de renvoi de la requête à *proc*2 via son cache local à *tick*= 3 ;
— (4) $L2$ reçoit la requête de renvoi et la transfère à *proc*2 à *tick*= 4 ;
— (5) *proc*2 reçoit la requête de renvoi à *tick*= 5 ;
— (6) Après la fin de la requête *ReadShared* à *tick*= 109, la requête *ReadUnique* est renvoyée au *CCN-xx* via $L2$. La réponse est reçue par le cache $L2$ du *proc*0. Il invalide sa ligne de cache à l'adresse 401480 en réponse à la requête. *proc*2 reçoit la donnée à *tick*=121.

Nous avons illustré en vert les transitions de la ligne de cache d'adresse 401480. Elle est d'abord allouée par le cache local à *proc*0 et passe de l'état *invalid* à l'état UC, puis elle est invalidée après la réponse à la requête de *snooping* reçue pour une requête *ReadUnique*. Son état passe de UC à I. Elle est ensuite allouée par le cache local de *proc*2 et passe de l'état *invalid* à l'état UC. Les résultats illustrés dans la trace de simulation montrent que la requête *ReadUnique* et la requête *ReadShared* ne peuvent pas s'exécuter dans le système en parallèle, vu que *ReadUnique* demande un accès exclusif à l'adresse

Tableau 6.2 – Paramètres de configuration du test 2.

Paramètre	valeur
Taille des caches	256KB
Taille d'une ligne de cache	64 *bytes*
Nombre de requêtes/*master*	20
Délai de *retry*	2ns.
Temps maximal de réponse	400ns
Taille de $memoryBuffer$	16MB
Latence de réponse de $L2$	1ns
Latence d'un *hit* de $L2$	1ns
Latence de réponse $snoopFilter$	1ns
Latence de réponse du $memoryBuffer$	1ns
Latence de réponse de mem	100ns
Latence des *snooping*	2ns
La durée d'un cycle de $systemMonitor$	1ns

401480. Ce n'est pas le cas pour la requête *readOnce* qui accède à une adresse différente dans l'espace d'adressage commun. L'exécution de la requête *ReadUnique* est retardée, mais elle obtiendra la donnée du cache local à *proc*0. L'accès à la mémoire extérieure est évité.
L'analyse de la trace de simulation précédente permet de vérifier le bon fonctionnement de notre architecture. Le comportement défini dans la spécification ACE et implémenté sur notre architecture à base d'interconnexion *CCN-xx* est vérifié à tout instant. Les résultats des différents tests effectués sur plusieurs configurations montrent que la cohérence des données est assurée dans tous les caches du système.

6.4.2 Évolution de la latence moyenne d'accès en fonction du taux de recouvrement des opérations

Le temps de réponse t_{rep} à une requête correspond à la durée entre l'instant d'envoi de la requête et l'instant de la réception de la réponse. La latence moyenne d'accès dans un système composé de N processeurs et coprocesseurs est donnée par la formule suivante :

$$t_{rep}^{moyen} = \sum_{p \in \mathcal{P}} t_{rep}^{p}/N$$

où P représente l'ensemble des processeurs et coprocesseurs du système simulé.
On définit le taux de recouvrement des opérations par le nombre de requêtes d'accès simultanés, envoyées sur la même adresse par différents processeurs et coprocesseurs.

Le tableau 6.2 donne un aperçu des différents paramètres de configuration du système simulé.

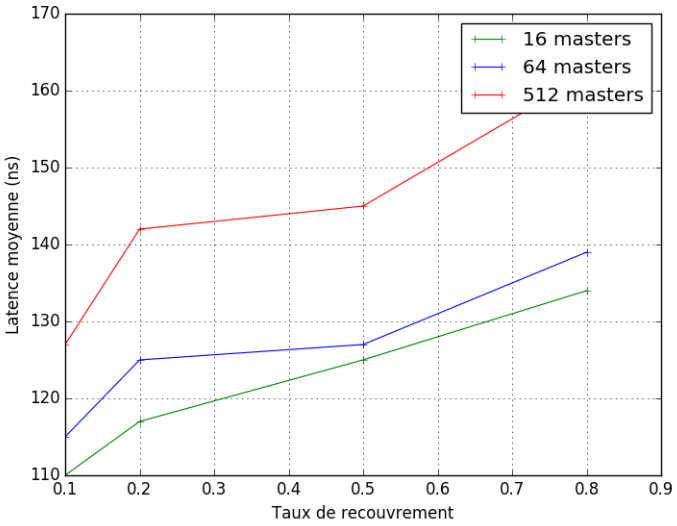

FIGURE 6.1 – Variation de la latence moyenne d'accès en fonction du taux de recouvrement des opérations.

Test 2.a et résultats

Le premier test consiste à analyser la variation du temps moyen de réponse pour la même configuration en fonction du taux de recouvrement. Les pourcentages de génération des requêtes sont générés aléatoirement. Les résultats sont illustrés dans le graphique 6.1

Après analyse du graphique, nous constatons que la latence moyenne d'accès varie en fonction du taux de recouvrement. Plus le taux augmente, plus le temps moyen de réponse augmente et ce pour les 3 configurations testées. Néanmoins, cette augmentation est limitée. Le nombre d'accès simultanés aux mêmes adresses par plusieurs processeurs et co-processeurs croît, entrainant la mise en attente de certaines requêtes par $systemMonitor$. L'augmentation est limitée pour deux raisons : (i) certaines requêtes mises en attente reçoivent la réponse lors de leur traitement par un des caches locaux ayant la donnée, évitant ainsi l'accès à la mémoire, (ii) les accès simultanés à la même adresse peuvent être des opérations de lecture partagée, donc plusieurs composants peuvent lire simultanément à la même adresse.

Test 2.b et résultats

Pour mieux analyser ce cas, nous avons effectué un deuxième test en fixant le taux de génération des requêtes *ReadShared* à 50% et le taux de génération des requêtes $ReadNotSharedDirty$ à 30%.
Comme illustré dans le graphique 6.2, la latence moyenne varie légèrement avec l'augmentation de la valeur du taux de recouvrement. Quand le nombre d'opérations de lecture partagée est grand, le taux de recouvrement n'a pas une forte influence sur la performance mesurée. Ceci s'explique par le nombre élevé de requêtes s'exécutant en parallèle dans le système. L'accès à la même adresse ne retarde pas toutes les requêtes qui sont majoritairement des requêtes de lecture partagée.

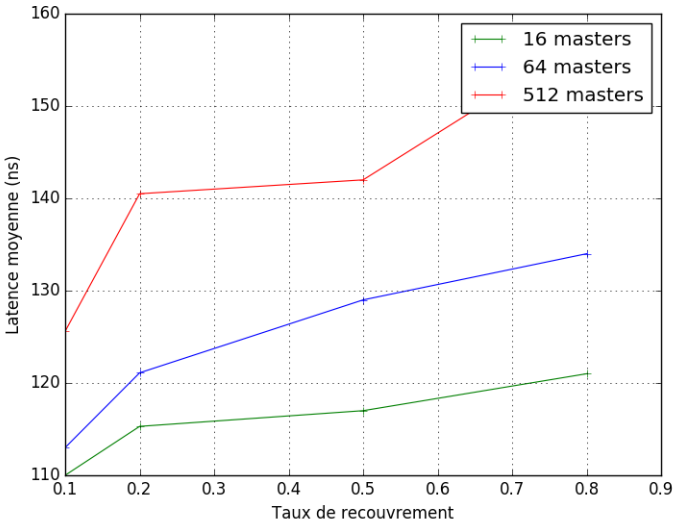

FIGURE 6.2 – Variation de la latence moyenne d'accès en fonction du taux de recouvrement avec un pourcentage de taux de génération des requêtes *ReadShared* élevé.

6.4.3 Évolution de la latence moyenne d'accès en fonction du nombre de processeurs et coprocesseurs dans le système

Cette étude vise à analyser l'impact de la taille du système sur le temps moyen de réponse à une requête.
Le tableau 6.3 résume les différents paramètres de configuration utilisés dans ce test.

Test 3 et résultats

Dans ce test, nous faisons varier le nombre de processeurs et coprocesseurs de 32 à 600 et étudions l'impact de cette variation sur l'exécution des requêtes.
L'analyse du graphique 6.3 permet de voir que le temps moyen de réponse augmente avec la taille du système. Cette augmentation est due au nombre élevé d'accès simultanés à une même adresse et donc au taux élevé d'occupation du réseau. Ce taux peut augmenter avec l'augmentation de la taille du système. En revanche, l'augmentation n'est pas considérable, car même si le pourcentage d'accès simultanés à la même adresse augmente, le nombre de réponses aux requêtes de *snooping* augmente. Par conséquent, le traitement est retardé, mais la durée d'exécution est optimisée par la suite pour certaines requêtes.
La latence moyenne varie avec l'augmentation de la taille du système. Cependant, ce n'est pas le seul facteur ayant un impact sur cette augmentation. Le taux de recouvrement élevé y contribue aussi.

6.4.4 Évolution de la durée de simulation en fonction de la taille du système simulé

Dans cette étude, nous analysons les performances en termes de temps de notre simulation. Nous rappelons que la simulation est une simulation à évènements discrets et la modélisation est au niveau transactionnel.

Tableau 6.3 – Paramètres de configuration du test 3 et 4.

Paramètre	valeur
Taille de la mémoire	64MB
Taille des caches	256KB
Taille d'une ligne de cache	64 *bytes*
Nombre de requêtes/*master*	20
Délai de *retry*	2ns.
Temps maximal de réponse	400ns
Taille de $memoryBuffer$	16MB
Latence de réponse de $L2$	1ns
Latence de *hit* de $L2$	1ns
Latence de réponse $snoopFilter$	1ns
Latence de réponse du $memoryBuffer$	8ns
Latence de réponse de mem	100ns
Latence des *snooping*	6ns
La durée d'un cycle de $systemMonitor$	1ns

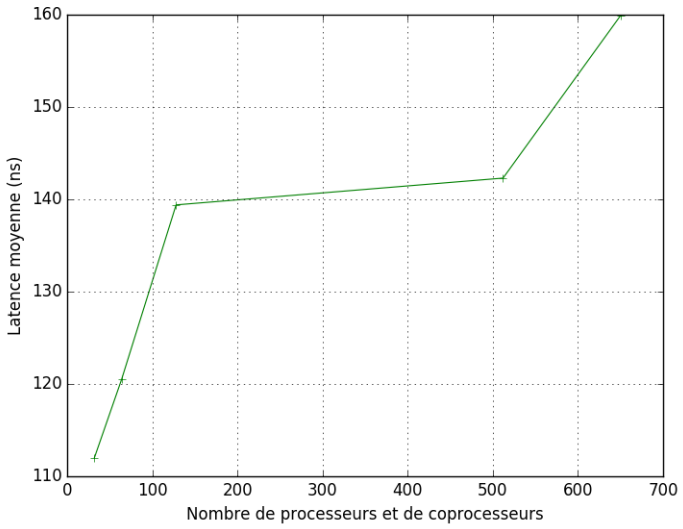

FIGURE 6.3 – Variation de la latence moyenne d'accès en fonction de la taille du système.

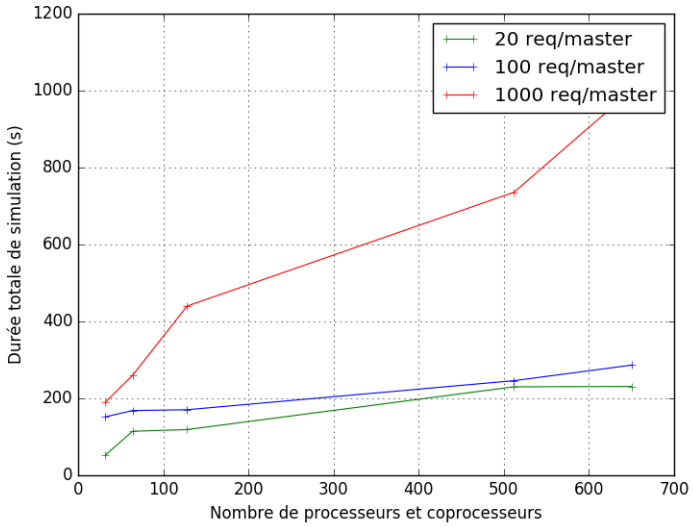

FIGURE 6.4 – Variation du temps de simulation en fonction de la taille du système.

La configuration du système simulé est celle résumée dans le tableau 6.3. Nous faisons varier le nombre de requêtes générées par un *master*.

Test 4 et résultats

Pour analyser le temps de simulation, nous faisons varier le nombre de processeurs et coprocesseurs de 16 à 650 et le nombre de requêtes générées par *master* de 20 à 1000. Le but est de donner une estimation sur les performances de notre implémentation et de la simulation sur gem5 dans ce cas d'étude.
Les résultats présentés dans le graphique 6.4 montrent que le temps de simulation (en ns) augmente considérablement avec la taille du système. Pour 650 processeurs et coprocesseurs générant 1000 requêtes chacun, la simulation a pris jusqu'à 1000 s. Ces tests ont été effectués sur une machine à un processeur *Intel Core i7* ayant une fréquence de 2.50GHz. 2 *cores* sont alloués à la simulation.
Les indices de performance obtenus sont cohérents par rapport à la variation de la taille du système, ils augmentent quand le nombre de traitements augmente. Le trafic sur le réseau augmente et génère plus de requêtes d'invalidation et plus de requêtes de *retry*. Pour le cas de 650 processeurs et coprocesseurs, en particulier, 650 requêtes doivent être exécutées en parallèle dans le système. La taille de la table *reqList* arrive vite à saturation et donc le nombre de requêtes *retry* augmente. Une des solutions consiste à augmenter cette taille mais nous nous limitons à des valeurs compatibles avec les contraintes physiques.
Le coût des simulations en temps de calcul pour de grandes configurations est important, mais reste acceptable.

6.4.5 Variation du taux de requêtes *retry* en fonction de la taille de la table *reqList*

Dans ce test, nous nous intéressons à l'impact de la taille de la table *reqList*, implémentée dans le composant *systemMonitor* sur le nombre de requêtes *retry*. Certaines

Tableau 6.4 – Paramètres de configuration du test 5.

Paramètre	valeur
Taille d'une ligne de cache	64 *bytes*
Nombre de requêtes/*master*	20
Délai de *retry*	2ns.
Temps maximal de réponse	400ns
Taille de $memoryBuffer$	16MB
Taille des caches	256KB
Latence de réponse de $L2$	1ns
Latence d'un *hit* de $L2$	1ns
Latence de réponse $snoopFilter$	1ns
Latence de réponse du $memoryBuffer$	8ns
Latence de réponse de mem	100ns
Latence des *snooping*	2ns
La durée d'un cycle de $systemMonitor$	1ns
Taille de la mémoire	64MB

requêtes *retry* ne sont pas renvoyées par le processeur ou le coprocesseur. En effet, lors de la réception d'une requête *retry*, le *master* peut décider de renvoyer cette requête ou d'en envoyer une nouvelle. Le taux de requêtes *retry* représente le nombre de requêtes *retry* par rapport au nombre de requêtes arrivant à l'entrée du *CCN-xx*.

Test 5 et résultats

Nous faisons varier la taille de la table en terme de nombre de requêtes présentes. Le tableau 6.4 résume la configuration étudiée.

En analysant la figure 6.5, nous remarquons que, pour la configuration de 16 *masters*, le taux de *retry* atteint la valeur 0.625 pour une taille de *reqList* égale à 10 requêtes et décroît ensuite pour se stabiliser à la valeur 0.187. Pour la valeur 10, le nombre maximal de requêtes s'exécutant en parallèle à tout instant est limité à 10, c'est pourquoi le nombre de requêtes rejetées par $systemMonitor$ croît. Pour une valeur supérieure à 10, les rejets du $systemMonitor$ sont dus au respect de la contrainte sur le séquencement des requêtes d'accès simultané aux mêmes adresses.
Le même comportement est observé pour la configuration de 64 *masters* avec une stabilisation du taux de *retry* à la valeur 100 de la taille de *reqList*. Cependant, pour la configuration de 512 *masters*, la valeur décroît pour chaque augmentation de la taille *reqList*, mais se stabilisera ensuite pour la valeur 600. Au delà de cette valeur, les rejets sont dus à la contrainte de séquencement des requêtes sur la même adresse. Les requêtes de lecture partagée ne sont pas concernées.

La taille de le table *reqList* a un impact important sur le nombre de requêtes rejetées à l'entrée du *CCN-xx*. Elle détermine le nombre de requêtes rejetées et envoyées pour *retry*. Dans notre implémentation, nous nous limitons à des valeurs compatibles avec les contraintes physiques.

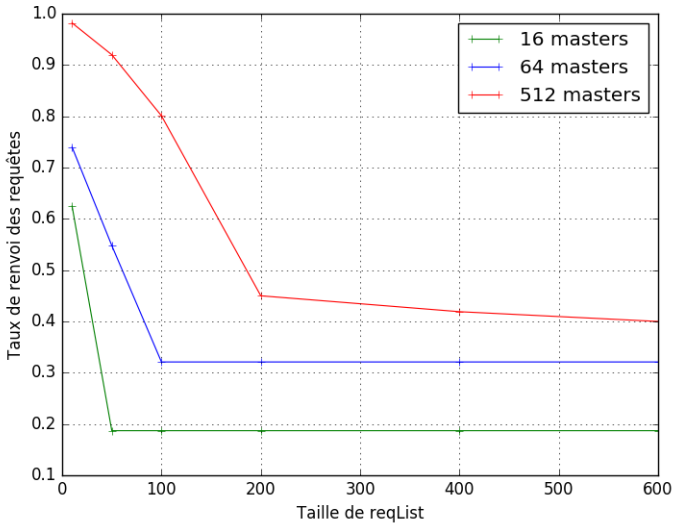

FIGURE 6.5 – Variation du taux de *retry* en fonction de la taille de la table *reqList*.

6.4.6 Variation du taux d'occupation de la table *reqList* en fonction de la taille du système

Nous définissons le taux d'occupation de la table *reqList* comme étant le pourcentage de temps pendant lequel elle est complètement occupée.

Test 6 et résultats

Nous analysons la variation du taux d'occupation de la table *reqList* pour 3 configurations de système, en fixant la taille de la table *reqList* à 10, 50 et 600 requêtes.
L'analyse du graphique 6.6 montre bien la corrélation entre le taux d'occupation et la taille du système. Cette corrélation apparaît dans toutes les évaluations. Plus la taille du système est grande, plus le nombre de requêtes envoyées est grand et donc dépasse dans certains cas le nombre de places disponibles dans *reqList*. L'augmentation est plus importante dans la configuration où la taille est fixée à 10 requêtes. En effet, la table *reqList* arrive vite à saturation.
La taille du système a une influence considérable sur le taux d'occupation de la table *reqList* et sera occupée en permanence. Pour avoir de bonnes performances, nous la fixons à 400 requêtes.

6.5 Conclusion

Les résultats des tests présentés dans ce chapitre montrent les points suivants :

— Nos modèles de *transacteurs* permettent de tester, d'une part, tous les types de requêtes définies dans le protocole ACE, et ce d'une manière complète, et d'autre part, le niveau de stress du *CCN-xx*, à travers le paramètre taux de recouvrement, qui détermine le taux des communications injectées en parallèle dans le réseau ;

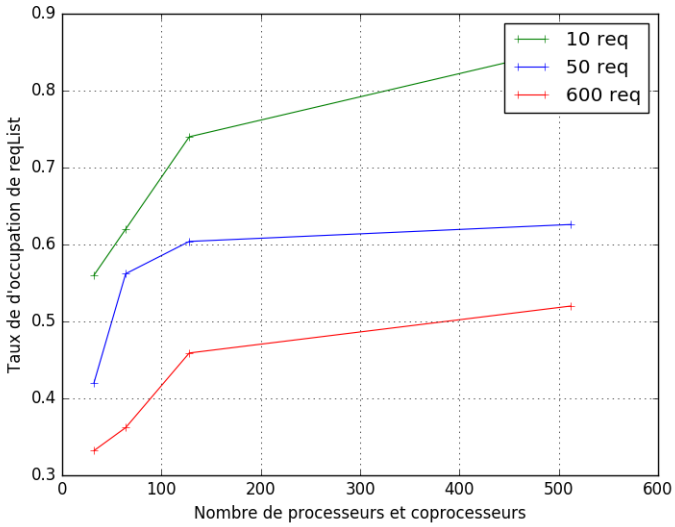

FIGURE 6.6 – Variation du taux d'occupation de la table *reqList* en fonction de la taille du système.

— La gestion du parallélisme par le composant ajouté *systemMonitor* à *CCN-xx* s'avère efficace, même pour de grandes configurations ;
— L'analyse du contenu des caches et des lignes de cache en termes d'états et de transitions, de la mémoire partagée, des requêtes et des réponses transitant dans le système a permis de vérifier la validité de l'implémentation du protocole ACE sur notre architecture. En effet, les données sont cohérentes dans les caches et consistantes par rapport à la mémoire, à tout moment ;
— Le nombre de processeurs et coprocesseurs n'est pas le seul facteur impactant la latence d'accès, on note également l'impact du taux de recouvrement, ainsi que la taille de *reqList* ;
— Le temps de simulation est de l'ordre de quelques milliers de secondes pour des traces d'accès complexes, ce qui reste raisonnable.

Notre méthode de vérification a permis de vérifier au niveau transactionnel l'aspect fonctionnel de l'architecture générique proposée. Les résultats obtenus montrent la pertinence de l'approche proposée.

Conclusions et perspectives

L'architecture des systèmes informatiques évolue vers une hétérogénéité des technologies de calcul et des modèles de programmation. Pour une utilisation efficace de ces systèmes, une gestion harmonieuse de cette hétérogénéité et une optimisation des ressources sont exigées. Avec l'augmentation du nombre d'unités de calcul hétérogènes entraînant la complexification de ces systèmes, ces tâches constituent des défis techniques majeurs dans la conception des futurs systèmes informatiques.
Cette thèse apporte deux principales contributions à la conception et validation d'architectures de calcul massivement parallèle.

En premier lieu, nous avons proposé une architecture à base de réseaux d'interconnexion pouvant supporter de manière générique un ensemble paramétrisable de processeurs hétérogènes, tout en assurant la cohérence de cache et la consistance mémoire. La gestion de la mémoire partagée est un facteur clé de performance qui peut dégrader la vitesse de traitement des données. L'obtention de hautes performances dépend entre autres, de la bonne utilisation de la mémoire partagée et de l'efficacité de gestion de la cohérence des données, face à la concurrence d'accès mémoire. Afin de gérer la cohérence des données dans les caches et la consistance par rapport à la mémoire, nous avons implémenté le protocole AMBA ACE de ARM sur l'architecture proposée.

En second lieu, pour valider cette conception, la démarche suivie consiste en deux étapes : la modélisation et la simulation. L'architecture a été modélisée au niveau transactionnel et simulée par des requêtes générées automatiquement et aléatoirement. Cette approche a été adoptée pour des considérations d'efficacité et de performance.
Après une analyse des différents environnements existants de modélisation et de simulation des systèmes informatiques, nous avons retenu l'environnement de simulation gem5, pour des raisons de facilité de modélisation et d'efficacité.

Les résultats obtenus montrent la pertinence de l'approche proposée. En effet, la valeur ajoutée de cette thèse est de réaliser les deux tâches de conception et de validation sur des configurations comportant un grand nombre de processeurs hétérogènes. Les mesures sur les tests démontrent la possibilité de supporter d'une manière scalable de grandes configurations, excédant quelques centaines d'unités de calcul.

Dans la continuité directe de notre travail de thèse, nous pouvons dégager deux principales directions de recherche. D'une part, cette méthode de vérification permet de vérifier au niveau transactionnel l'aspect fonctionnel de l'architecture générique, mais pourrait être étendue pour effectuer des mesures de performance et de consommation énergétique sur le système étudié. D'autre part, la performance d'un tel système permettrait d'exé-

cuter du logiciel et donc serait un support précieux pour le développement du *co-design* matériel et logiciel. En effet, la démarche de *co-design* n'est pas une activité isolée de la conception de l'ensemble d'un système. Un des problèmes qui pourrait être étudié est le problème du partitionnement matériel/logiciel qui est au cœur de l'activité de *co-design*. Ce dernier est un problème NP-complet et nécessite de mettre en œuvre des méthodes d'optimisation combinatoires avancées pour résoudre efficacement le problème.

Références bibliographiques

[ARM, 2015] (2015). ARM website. https://www.arm.com/products/system-ip/interconnect/corelink-ccn-family.php,. Accessed : 2015-01-30.

[GEM, 2015] (2015). GEM5 website. http://gem5.org/Main_Page,. Accessed : 2015-01-30.

[ARM, 2016] (2016). ARM website. http://arm.com. Accessed : 2016-09-30.

[BUL, 2016] (2016). BULL website. http://www.bull.com/fr/serveurs-bullion. Accessed : 2016-05-30.

[Agarwal et al., 1988] Agarwal, A., Simoni, R., Hennessy, J., and Horowitz, M. (1988). An evaluation of directory schemes for cache coherence. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 280–298. IEEE Computer Society Press.

[Alpern et al., 1994] Alpern, B., Carter, L., Feig, E., and Selker, T. (1994). The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3) :72–109.

[AMBA and Specification-AXI, 2011] AMBA, A. and Specification-AXI, A. P. (2011). Axi4, and axi4-lite. *ACE and ACE-Lite, ARM IHI D*, 22.

[Archibald and Baer, 1986] Archibald, J. and Baer, J.-L. (1986). Cache coherence protocols : Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems (TOCS)*, 4(4) :273–298.

[Baxter et al., 1999] Baxter, W. F., Gelinas, R. G., Guyer, J. M., Huck, D. R., Hunt, M. F., Keating, D. L., Kimmell, J. S., Roux, P. J., Truebenbach, L. M., Valentine, R. P., et al. (1999). Symmetric multiprocessing computer with non-uniform memory access architecture. US Patent 5,887,146.

[Biemans and Blonk, 1986] Biemans, F. and Blonk, P. (1986). On the formal specification and verification of cim architectures using lotos. *Computers in Industry*, 7(6) :491–504.

[Binkert et al., 2003] Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K. (2003). Network-oriented full-system simulation using m5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43. Citeseer.

[Censier and Feautrier, 1978] Censier, L. M. and Feautrier, P. (1978). A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 100(12) :1112–1118.

[Chan et al., 1993] Chan, K., Alexander, T., Hu, C., Larson, D., Noordeen, N., VanAtta, Y., Wylegala, T., and Ziai, S. (1993). Multiprocessor features of the hp corporate business servers. In *Compcon Spring'93, Digest of Papers.*, pages 330–337. IEEE.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods : State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4) :626–643.

[Cohen and Schirmer, 2010] Cohen, E. and Schirmer, B. (2010). From total store order to sequential consistency : A practical reduction theorem. In *International Conference on Interactive Theorem Proving*, pages 403–418. Springer.

[Correale Jr, 1991a] Correale Jr, A. (1991a). Memory by-pass for write through read operations. US Patent 4,998,221.

[Correale Jr, 1991b] Correale Jr, A. (1991b). Memory by-pass for write through read operations. US Patent 4,998,221.

[Culler et al., 1999] Culler, D. E., Singh, J. P., and Gupta, A. (1999). *Parallel computer architecture : a hardware/software approach*. Gulf Professional Publishing.

[Dally and Towles, 2004a] Dally, W. J. and Towles, B. P. (2004a). *Principles and practices of interconnection networks*. Elsevier.

[Dally and Towles, 2004b] Dally, W. J. and Towles, B. P. (2004b). *Principles and practices of interconnection networks*. Elsevier.

[Devietti et al., 2011] Devietti, J., Nelson, J., Bergan, T., Ceze, L., and Grossman, D. (2011). Rcdc : a relaxed consistency deterministic computer. In *ACM SIGPLAN Notices*, volume 46, pages 67–78. ACM.

[Dwyer et al., 1999] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE.

[Gharachorloo et al., 1990a] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. (1990a). *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM.

[Gharachorloo et al., 1990b] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. (1990b). *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM.

[Goodman, 1983a] Goodman, J. R. (1983a). Using cache memory to reduce processor-memory traffic. *ACM SIGARCH Computer Architecture News*, 11(3) :124–131.

[Goodman, 1983b] Goodman, J. R. (1983b). Using cache memory to reduce processor-memory traffic. *ACM SIGARCH Computer Architecture News*, 11(3) :124–131.

[Goodman, 1991] Goodman, J. R. (1991). *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department.

[Hachtel and Somenzi, 2006] Hachtel, G. D. and Somenzi, F. (2006). *Logic synthesis and verification algorithms*. Springer Science & Business Media.

[Hagersten and Hill, 1999] Hagersten, E. E. and Hill, M. D. (1999). Hierarchical smp computer system. US Patent 5,862,357.

[Hu et al., 1997] Hu, A. J., Fujita, M., and Wilson, C. (1997). Formal verification of the hal s1 system cache coherence protocol. In *Computer Design : VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*, pages 438–444. IEEE.

[Jeff, 2012] Jeff, B. (2012). Big. little system architecture from arm : saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1143–1146. ACM.

[Kumar et al., 2005] Kumar, R., Zyuban, V., and Tullsen, D. M. (2005). Interconnections in multi-core architectures : Understanding mechanisms, overheads and scaling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 408–419. IEEE.

[Kumar et al., 1994] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to parallel computing : design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, CA.

[Lamport, 1979] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9) :690–691.

[Lenoski and Weber, 2014] Lenoski, D. E. and Weber, W.-D. (2014). *Scalable shared-memory multiprocessing*. Elsevier.

[Li and Hudak, 1989] Li, K. and Hudak, P. (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4) :321–359.

[Manchanda and Anand, 2010] Manchanda, N. and Anand, K. (2010). Non-uniform memory access (numa). *New York University*.

[Maria, 1997] Maria, A. (1997). Introduction to modeling and simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 7–13. IEEE Computer Society.

[Martin et al., 2005] Martin, M. M., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. (2005). Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4) :92–99.

[Matthews, 1999] Matthews, R. A. (1999). The science of murphy's law. In *PROCEEDINGS-ROYAL INSTITUTION OF GREAT BRITAIN*, volume 70, pages 75–96. Oxford University Press.

[Miao et al., 2006] Miao, Y.-l., Miao, X.-l., Bian, Z.-Z., and Zhang, Y.-j. (2006). Design and application of embedded system based on arm7 lpc2104 processor in telemedicine. In *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*, pages 2187–2190. IEEE.

[Milne, 1993] Milne, G. J. (1993). *Formal specification and verification of digital systems*. McGraw-Hill, Inc.

[Navabi, 1997] Navabi, Z. (1997). *VHDL : Analysis and modeling of digital systems*. McGraw-Hill, Inc.

[Navaridas et al., 2011] Navaridas, J., Miguel-Alonso, J., Pascual, J. A., and Ridruejo, F. J. (2011). Simulating and evaluating interconnection networks with insee. *Simulation Modelling Practice and Theory*, 19(1) :494–515.

[Nguyen et al., 2014] Nguyen, H.-N., Nguyen, T.-A., Nakib, A., and Petit, E. (2014). Transaction level simulation of network-based computing systems. In *2014 International Conference on Computational Science and Computational Intelligence*.

[Oury et al., 2015] Oury, P., Heaton, N., and Penman, S. (2015). Methodology to verify, debug and evaluate performances of noc based interconnects. In *Proceedings of the 8th International Workshop on Network on Chip Architectures*, pages 39–42. ACM.

[Park, 2007] Park, G.-J. (2007). *Analytic methods for design practice*. Springer Science & Business Media.

[Pees et al., 1999] Pees, S., Hoffmann, A., Zivojnovic, V., and Meyr, H. (1999). Lisa—machine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 933–938. ACM.

[Penton and Jalloq, 2006] Penton, J. and Jalloq, S. (2006). Cortex-r4 : A mid-range processor for deeply-embedded applications. *ARM white paper, May*.

[Protic et al., 1998] Protic, J., Tomasevic, M., and Milutinović, V. (1998). *Distributed shared memory : Concepts and systems*, volume 21. John Wiley & Sons.

[Prybylski et al., 1988] Prybylski, S., Horowitz, M., and Hennessy, J. (1988). Performance tradeoffs in cache design. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 290–298. IEEE Computer Society Press.

[Purcell and Cheng, 2004] Purcell, S. C. and Cheng, C. T. (2004). Layered crossbar for interconnection of multiple processors and shared memories. US Patent 6,836,815.

[Puzak, 1985a] Puzak, T. R. (1985a). Analysis of cache replacement-algorithms.

[Puzak, 1985b] Puzak, T. R. (1985b). Analysis of cache replacement-algorithms.

[Rixner et al., 2000] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. (2000). Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 128–138. ACM.

[Rodgers, 1985] Rodgers, D. P. (1985). Improvements in multiprocessor system design. In *ACM SIGARCH Computer Architecture News*, volume 13, pages 225–231. IEEE Computer Society Press.

[Rodrigues et al., 2011] Rodrigues, A. F., Hemmert, K. S., Barrett, B. W., Kersey, C., Oldfield, R., Weston, M., Risen, R., Cook, J., Rosenfeld, P., CooperBalls, E., et al. (2011). The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4) :37–42.

[Salminen et al., 2002] Salminen, E., Lahtinen, V., Kuusilinna, K., and Hamalainen, T. (2002). Overview of bus-based system-on-chip interconnections. In *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, volume 2, pages II–372. IEEE.

[Schweizer and Carroll, 1990] Schweizer, P. T. and Carroll, M. L. (1990). Hierarchical multiple bus computer architecture. US Patent 4,912,633.

[Scott, 1996] Scott, S. L. (1996). Synchronization and communication in the t3e multiprocessor. In *ACM SIGPLAN Notices*, volume 31, pages 26–36. ACM.

[Singhal and Engineer, 2008] Singhal, R. and Engineer, S. P. (2008). Inside intel core microarchitecture (nehalem). In *A Symposium on High Performance Chips*, volume 20.

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

[Swan, 2006] Swan, S. (2006). Systemc transaction level models and rtl verification. In *Proceedings of the 43rd annual Design Automation Conference*, pages 90–92. ACM.

[Ulfsnes, 2013] Ulfsnes, R. (2013). Design of a snoop filter for snoop based cache coherency protocols.

[Van Laer et al., 2013] Van Laer, A., Jones, T., and Watts, P. M. (2013). Full system simulation of optically interconnected chip multiprocessors using gem5. In *Optical Fiber Communication Conference*, pages OTh1A–2. Optical Society of America.

[Varga et al., 2001] Varga, A. et al. (2001). The omnet++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM'2001)*, volume 9, page 65. sn.

[Vashi and Strickland, 1996] Vashi, A. D. and Strickland, T. S. (1996). Computer memory data merging technique for computers with write-back caches. US Patent 5,530,835.

[Wenger et al., 2013] Wenger, E., Unterluggauer, T., and Werner, M. (2013). 8/16/32 shades of elliptic curve cryptography on embedded processors. In *International Conference on Cryptology in India*, pages 244–261. Springer.

[Wilson Jr, 1987] Wilson Jr, A. W. (1987). Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 244–252. ACM.

[Yang et al., 2012] Yang, K., Fu, Y., Han, X., and Jiang, J. (2012). Efficient broadcast scheme based on sub-network partition for many-core cmps on gem5 simulator. In *CCF National Conference on Compujter Engineering and Technology*, pages 163–172. Springer.

[Zucker, 1992] Zucker, R. N. (1992). *Relaxed consistency and synchronization in parallel processors*. PhD thesis, University of Washington.