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

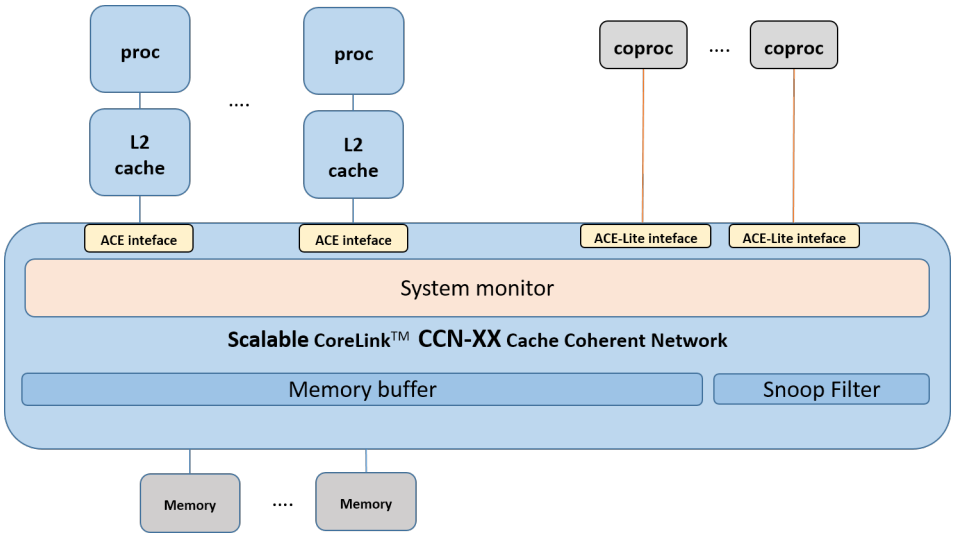

FIGURE 4.1 – L'architecture générique proposée.

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.

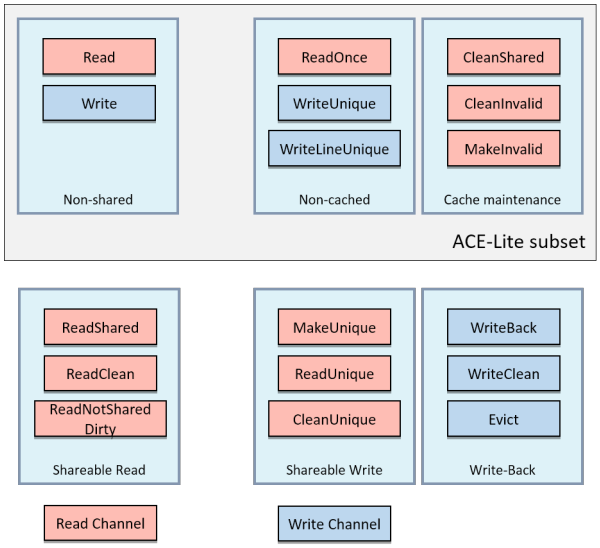

FIGURE 4.2 – Les groupes de transactions selon le protocole AMBA ACE

.

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

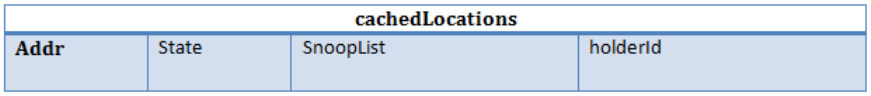

FIGURE 4.4 – Structure d'une entrée du *snoopfilter*.

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

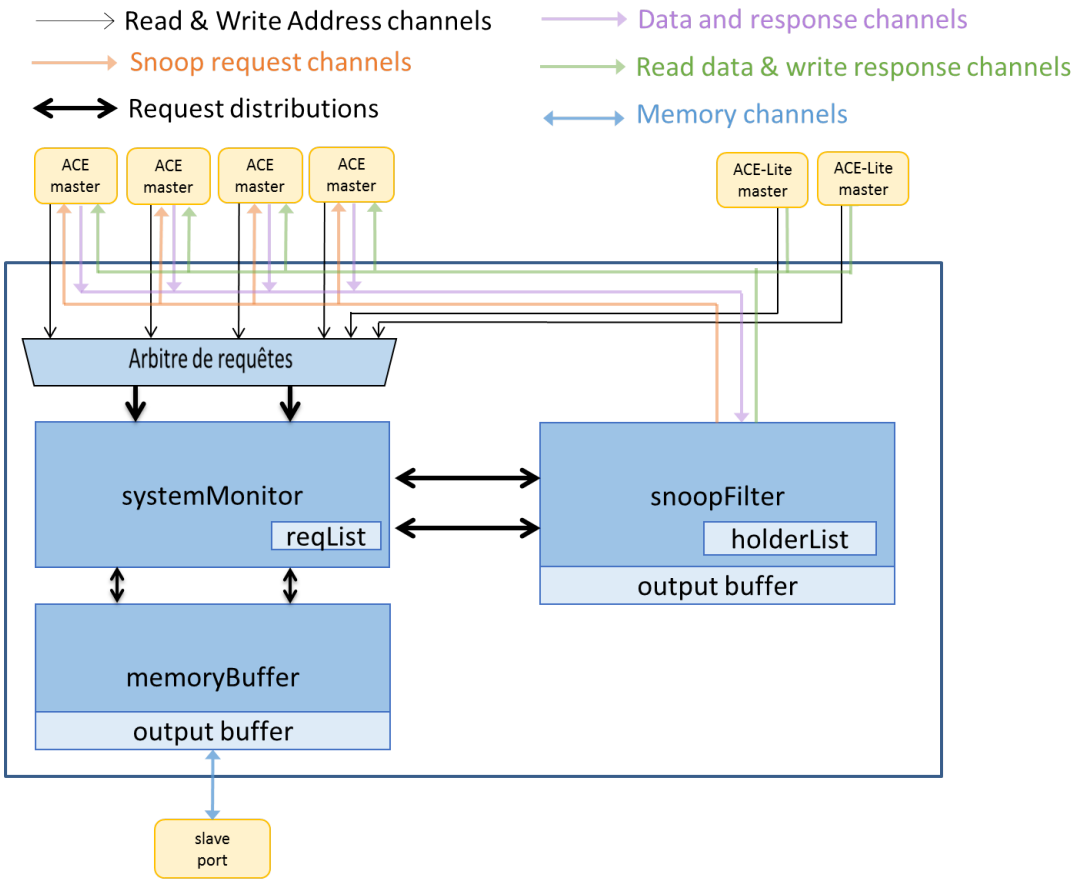

FIGURE 4.5 – Micro-architecture de l'interconnexion *CCN-xx*.

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 ;

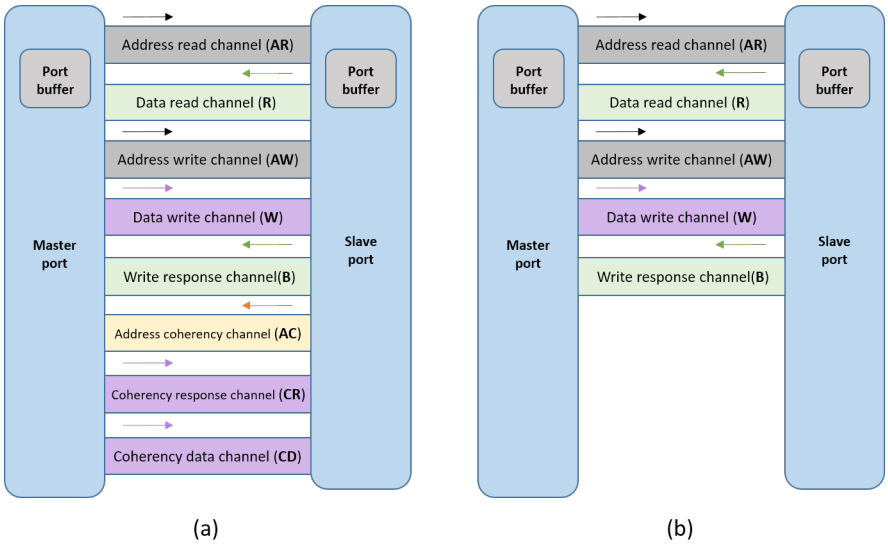

FIGURE 4.6 – Architecture des canaux : (a) pour un ACE *master*, (b) pour un ACE-Lite *master*.

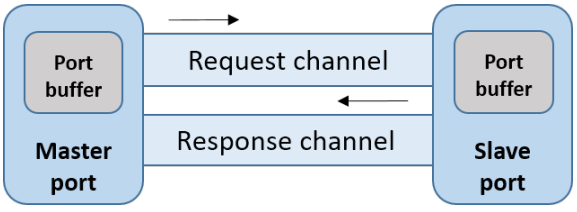

FIGURE 4.7 – Architecture des canaux entre le *CCN-xx* et la mémoire.

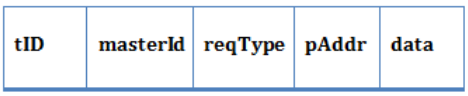

FIGURE 4.8 – Structure d'une transaction.

— 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

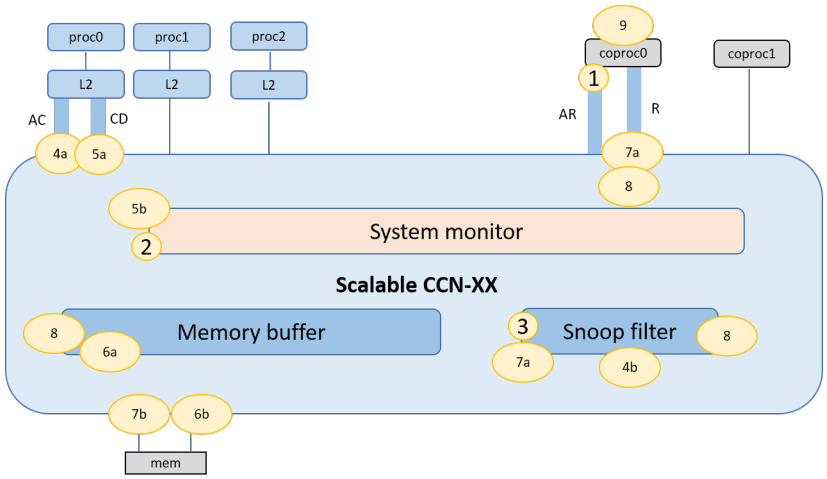

FIGURE 4.9 – Déroulement d'une transaction *ReadOnce* dans le système à base de *CCN-xx*.

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.