Protocoles scalables de cohérence des caches pour processeurs manycore à espace d'adressage partagé visant la basse consommation.

Hao Liu

RAPPORT DE THESE

Spécialité : Informatique

École doctorale : « EDITE de Paris »

réalisée au

Département SOC du laboratoire LIP6

présentée par

Hao LIU

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Protocoles scalables de cohérence des caches pour processeurs manycore à espace d'adressage partagé visant la basse consommation

soutenue le 27 janvier 2016

devant le jury composé de :

Pr	Daniel Etiemble	Rapporteur
Pr	Smail Niar	Rapporteur
Pr	Bertrand Granado	Examinateur
Dr	Huy-Nam Nguyen	Examinateur
Pr	Alain Greiner	Directeur de thèse
Dr	Franck Wajsbürt	Encadrant de thèse

Résumé

Mots-clés : manycore, cohérence des caches, écriture immédiate, écriture différée, TLB.

L'architecture TSAR (Tera-Scale ARchitecture) développée conjointement par BULL, le LIP6 et le CEA-LETI est une architecture Manycore CC-NUMA extensible jusqu'à 1024 cœurs. Le protocole de cohérence des caches DHCCP dans l'architecture TSAR repose sur le principe du répertoire global distribué en utilisant la stratégie d'écriture immédiate afin de passer à l'échelle, mais cette scalabilité a un coût énergétique important que nous cherchons à réduire. Actuellement, les plus grandes entreprises dans le domaine des semi-conducteurs, comme Intel ou AMD, utilisent les protocoles MESI ou MOESI dans leurs processeurs multicoeurs. Ces types de protocoles utilisent la stratégie d'écriture différée pour réduire la consommation énergétique due aux écritures. Mais la complexité d'implémentation et la forte augmentation de ce trafic de cohérence, quand le nombre de processeurs augmente, limitent le passage à l'échelle de ces protocoles au-delà de quelques dizaines de cœurs.

Dans cette thèse, nous proposons un nouveau protocole de cohérence de cache utilisant une méthode hybride pour traiter les écritures dans le cache L1 privé : pour les lignes non partagées, le contrôleur de cache L1 utilise la stratégie d'écriture différée, de façon à modifier les lignes localement. Pour les lignes partagées, le contrôleur de cache L1 utilise la stratégie d'écriture immédiate pour éviter l'état de propriété exclusive sur ces lignes partagées. Cette méthode, appelée RWT pour Released Write Through, passe non seulement à l'échelle, mais réduit aussi significativement la consommation énergétique liée aux écritures.

Nous avons aussi optimisé la gestion de la cohérence des TLBs dans l'architecture TSAR, en termes de performance et de consommation énergétique.

Enfin, nous introduisons dans cette thèse un niveau de cache, appelé Micro-Cache, entre le cœur et le cache L1, afin de réduire le nombre d'accès au cache d'instructions et donc la consommation énergétique sans aucun impact sur les performances.

abstract

Key words : manycore, cache, write-back, write-through, TLB.

The TSAR architecture (Tera-Scale ARchitecture) developed jointly by Lip6 Bull and CEA-LETI is a CC-NUMA manycore architecture which is scalable up to 1024 cores. The DHCCP cache coherence protocol in the TSAR architecture is a global directory protocol using the write-through policy in the L1 cache for scalability purpose, but this write policy causes a high power consumption which we want to reduce. Currently the biggest semiconductors companies, such as Intel or AMD, use the MESI MOESI protocols in their multi-core processors. These protocols use the write-back policy to reduce the high power consumption due to writes. However, the complexity of implementation and the sharp increase in the coherence traffic when the number of processors increases limits the scalability of these protocols beyond a few dozen cores.

In this thesis, we propose a new cache coherence protocol using a hybrid method to process write requests in the L1 private cache : for exclusive lines, the L1 cache controller chooses the write-back policy in order to modify locally the lines as well as eliminate the write traffic for exclusive lines. For shared lines, the L1 cache controller uses the write-through policy to simplify the protocol and in order to guarantee the scalability.

We also optimized the current solution for the TLB coherence problem in the TSAR architecture. The new method which is called CC-TLB not only improves the performance, but also reduces the energy consumption.

Finally, this thesis introduces a new micro cache between the core and the L1 cache, which allows to reduce the number of accesses to the instruction cache, in order to save energy without any impact on performances.

Sommaire

Figures

Mise en forme du texte

- Tous les noms en anglais sont en *italique*.
- Tous les mots en abréviation sont en **gras**.
- Tous les noms de requête sont en police `courrier-italique`.
- Tous les noms d'état d'automate sont en police `courrier`.
- Tous les noms d'automate sont en police **`courrier-gras`**.
- Tous les noms de composants ou de signaux de composant sont en ***italique-gras***

Introduction

Les microprocesseurs généralistes sont apparus commercialement en 1971 avec l'Intel 4004. Il était alors possible d'intégrer tous les composants d'un processeur dans un seul boîtier. Il est désormais possible d'intégrer des dizaines et très bientôt des centaines de coeurs dans un seul boîtier. La complexité de telles architectures n'est pas, ou n'est plus, dans les unités de calcul. La complexité se trouve dans le système mémoire.

En effet, au début, le système mémoire devait être capable de fournir à l'unité de calcul une nouvelle instruction tous les 5 à 10 cycles d'horloge et lire ou écrire des données avec un débit plus faible encore. Dans les architectures modernes, et pour celles à venir, le système mémoire doit fournir des centaines d'instructions à chaque cycle et permettre des dizaines d'accès simultanées en lecture ou en écriture de données. Pour parvenir à atteindre ces débits, le système mémoire est devenu une hiérarchie de caches de mémoire. Les caches permettent de rapprocher des unités de calcul, les instructions et les données fréquemment utilisées en les recopiant dans des petites mémoires locales. Toutefois, ces caches doivent tous être cohérents. La gestion de la cohérence de caches est un problème d'autant plus délicat que les solutions doivent être extensibles (passer à l'échelle), et donc rester performantes, quelque soit le nombre de caches à maintenir cohérent. En outre, les solutions doivent être économes en énergie, c'est-à-dire ne pas imposer de nombreux échanges d'information entre les caches.

Cette thèse adresse justement le problème de la cohérence de caches en proposant des solutions à la fois efficaces en performance et économes en énergie. Elle repose sur l'architecture de processeur massivement multicœurs (*manycore*) généralistes, **TSAR**, pour laquelle elle étudie et propose de nouveaux protocoles de cohérence.

Organisation du manuscrit

Dans le premier chapitre, nous présentons le contexte et la problématique de la thèse. Nous commençons par introduire les architectures *manycores* et nous détaillons quelques problèmes de cohérence de leurs caches de données et d'instructions, mais aussi les problèmes de cohérence de caches de traduction d'adresse (**TLB**) qui permettent au système d'exploitation de faire travailler les applications dans des espaces d'adressage virtuel. Dans la deuxième partie, nous détaillons le protocole de cohérence de caches (instructions, données et adresses), nommé **DHCCP**, à l'œuvre dans l'architecture *manycore* **TSAR** support de notre travail. Puis, le chapitre analyse les faiblesses du protocole **DHCCP**, en particulier pour ce qui concerne la consommation énergétique. Enfin, le chapitre s'achève par la liste des questions auxquelles nous avons tenté d'apporter des réponses.

Le deuxième chapitre propose un état de l'art des solutions existantes pour la cohérence de caches. Nous analysons en détail les protocoles de cohérence de cache et les différentes stratégies de mise à jour (écriture) de la mémoire, dans trois architectures, multicoeurs et *manycore*, commerciales actuelles (INTEL, AMD, TILERA).

Le troisième chapitre présente une évolution du protocole **DHCCP**, appelée **RWT** (*Released Write Through*), pour réduire de manière significative le trafic sur le micro réseau et donc l'énergie consommée, sans perdre en performance ou en extensibilité. Nous décrivons à la fois le principe général et l'implémentation du protocole **RWT**, en insistant sur les différences par rapport au protocole **DHCCP**.

Le quatrième chapitre présente un protocole de cohérence de cache plus proche des protocoles actuellement utilisés dans l'industrie. Ce protocole **HMESI** a été implémenté dans l'architecture **TSAR** pour permettre une comparaison non biaisée (c'est-à-dire sur la même architecture matérielle) entre les protocoles utilisant une stratégie d'écriture différée (**HMESI**) et les protocoles utilisant une stratégie d'écriture immédiate (**DHCCP**) ou mixte (**RWT**).

Le cinquième chapitre propose une évolution du protocole de cohérence des **TLBs** dans l'architecture **TSAR**. Tout d'abord, nous décrivons le protocole existant, puis nous expliquons pourquoi nous avons fait évoluer ce protocole. Ensuite, nous détaillons notre contribution et nous montrons les avantages à la fois sur le plan de la performance et d'énergie, mais également sur le plan du coût matériel.

Le sixième chapitre présente une optimisation portant sur l'accès au cache instructions de premier niveau qui permet de réduire significativement la consommation énergétique pour un coût mineur en matériel.

Les chapitres 7, 8 et 9 décrivent les expérimentations mises en œuvre pour évaluer chacune des trois optimisations présentées dans les chapitres 3, 5 et 6. Dans ces chapitres, nous détaillons les plates-formes de prototypages construites, les logiciels de test (*benchmarks*) choisis, et les instruments de mesure ajoutés aux plates-formes permettant l'analyse quantitative des solutions proposées. Nous montrons tout d'abord les résultats en matière de performance et de consommation énergétique pour les protocoles de cohérence de cache : **DHCCP**, **RWT** et **HMESI**. Ensuite, nous présentons l'impact de la modification de la gestion de la cohérence des **TLBs**. Enfin, nous étudions la réduction de la consommation énergétique sur le cache instructions de premier niveau.

Les annexes détaillent certains automates des protocoles **RWT** et **HMESI** dans les caches de niveau 1 et 2.

Chapitre 1

Problématique

1.1 Architectures *manycores*

Les architectures multicœurs [1] sont apparues commercialement au début des années 2000, mais elles se sont imposées à partir de 2005 [2]. Avant cette date, l'augmentation du nombre d'instructions exécutées par seconde était obtenue en complexifiant l'architecture interne des cœurs et en augmentant leur fréquence de fonctionnement. Ce faisant, les processeurs dissipaient toujours plus de puissance électrique. Le problème est qu'il n'est pas simple d'évacuer la chaleur produite par un circuit avec une simple ventilation au-delà de 100W/cm2. Alors, pour continuer à augmenter le nombre d'instructions exécutées par seconde, les constructeurs ont choisi de remplacer un cœur complexe par deux plus simples sans augmenter le budget énergétique. La réduction de puissance électrique consommée par cœur a été obtenue en simplifiant leur architecture interne, par exemple en réduisant les mécanismes d'exécution spéculative. Ils ont également plafonné la fréquence de fonctionnement ou encore l'ont rendu modulable, tout comme la tension d'alimentation, en fonction de la charge de calcul. Dès lors, le nombre de cœurs n'a cessé d'augmenter, pour atteindre plusieurs dizaines et bientôt plusieurs centaines par circuit. Les architectures contenant quelques cœurs sont nommées multicœurs. Au-delà de cent cœurs, elles sont massivement multicœurs, mais le terme consacré est *manycore*.

1.1.1 Architectures à mémoire partagée

Il existe plusieurs types d'architecture *manycore*. Notre travail porte sur un type en particulier : les architectures *manycores* à mémoire partagée et caches cohérents [3]. Dans ces architectures : (i) tous les cœurs partagent le même espace d'adressage physique, c'est-à-dire qu'ils partagent la même mémoire ; (ii) le système mémoire est constitué d'une hiérarchie des caches dont les mouvements de données sont gérés par des automates matériels ; (iii) la cohérence des caches est garantie par des automates matériels ; (iv) chaque application dispose d'un espace d'adressage propre dit espace virtuel, et chaque cœur contient une unité de traduction d'adresses virtuelles vers adresse physique. Les architectures (i) sans caches ou (ii) avec des espaces d'adressages distincts pour chaque cœur (ou groupe de cœurs) imposant au logiciel la gestion du mouvement des données entre les bancs de mémoire ou encore ; (iii) sans cohérence des caches

ou enfin ; (iv) sans mémoire virtuelle ne rentrent pas dans le cadre de notre étude. En effet, seules les architectures à mémoire partagées et à caches cohérents sont capables d'exécuter simplement des systèmes d'exploitation de type UNIX, comme Linux ou BSD. Il existe deux grands types d'architectures à espace d'adressage partagé [4] : les architectures **SMP** (*Symmetric Multi Processor*) et les architectures **NUMA** (*Non Uniform Memory Access*).

La figure 1.1 montre une architecture **SMP**. Tous les cœurs et la mémoire sont connectés par un *bus*. Les cœurs sont identiques, et le temps d'accès à la mémoire est uniforme. Les architectures **SMP** qui garantissent la cohérence des caches s'appellent **CC-SMP**.

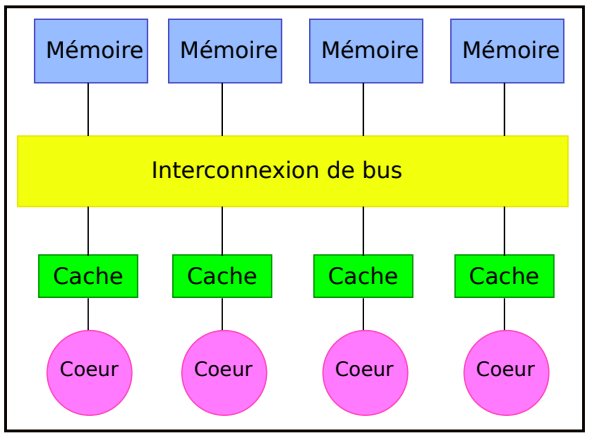
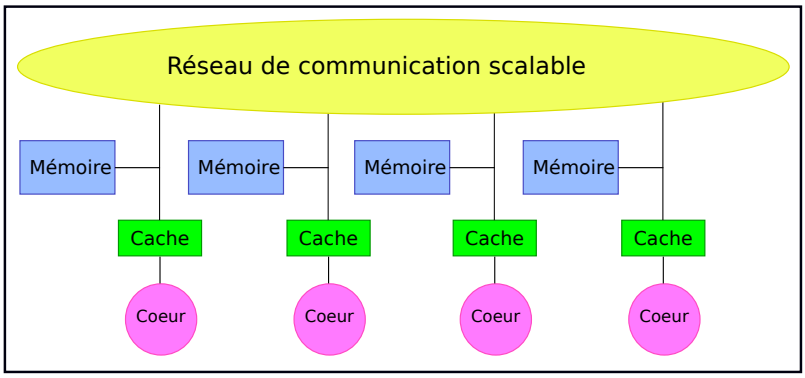


FIGURE 1.1 – Architecture **SMP**

La figure 1.2 montre une architecture **NUMA**. Les cœurs identiques sont connectés en réseau. Chaque nœud dans ce réseau contient un ou plusieurs cœurs. La mémoire partagée est physiquement distribuée dans chaque noeud du réseau.



Pour chaque cœur, les temps d'accès ainsi que la consommation énergétique différent donc suivant la zone de mémoire accédée (figure 1.2). **CC-NUMA** signifie architecture **NUMA** avec cohérence des caches. Les architectures **NUMA** peuvent contenir plus de cœurs que l'architecture **SMP**.

FIGURE 1.2 – Architecture **NUMA**

1.1.2 Mémoires caches

Plus il y a de cœurs, plus le nombre d'instructions exécutées par cycle est important, plus la quantité de données traitées augmente, plus le débit des échanges entre les cœurs et la mémoire augmente. La hiérarchie des caches présente entre les cœurs et la mémoire principale permet d'augmenter ce débit, en recopiant localement, au plus près des cœurs, les instructions et les données fréquemment utilisées. La capacité des caches est bien inférieure à la capacité de stockage de la mémoire principale. Les caches copient temporairement une partie du contenu de la mémoire. Les caches de premier

niveau sont les plus rapides, mais ils sont aussi les plus petits. Plus on s'éloigne des cœurs, plus les caches sont partagés, plus ils sont grands, mais plus ils sont lents.

Les caches stockent des blocs de données contiguës en mémoire. Ces blocs sont de taille fixe et sont appelés **lignes de cache** (*cache line* en anglais). L'adresse d'un octet à l'intérieur d'une ligne de cache est nommée *offset*. L'adresse en mémoire de chaque ligne présentée dans un cache est rangée dans un répertoire. Lors d'un accès mémoire par un cœur, le répertoire permet de savoir si une donnée est présente dans le cache en y recherchant son numéro de ligne. Afin de réduire le temps d'accès, la position de la ligne dans le cache est définie par une partie de l'adresse de la ligne appelée *set*. Le reste de l'adresse de ligne est, nommé *tag*, est stocké dans le répertoire. Ce répertoire est indexé par *set* ou encore associatif par ensemble comme montré dans la figure 1.3.

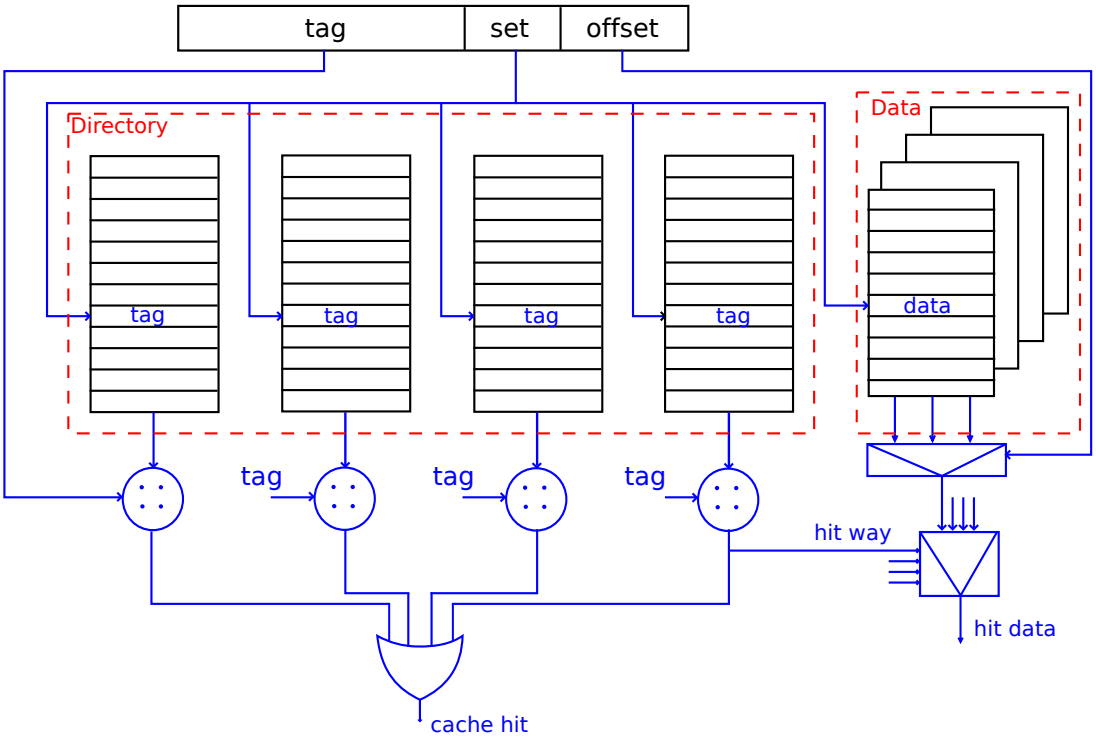


FIGURE 1.3 – Principe de cache

1.1.3 Protocoles de cohérence

Dans une architecture *manycore* disposant des caches, chaque cœur contient son ou ses caches privés (caches de premier niveau), lesquels accèdent à des caches de niveau supérieur et le dernier niveau de cache (**LLC** comme *Last Level Cache*) est partagé par tous les cœurs. Dans un programme (une application) multitâches coopératives, une ligne de cache écrite par un coeur peut être lue par plusieurs autres coeurs. Par conséquent, les caches privés des différents lecteurs stockent la même ligne de cache partagée. Lorsque le cœur écrivain modifie une donnée dans la ligne partagée, il faut que tous les cœurs lecteur voient cette modification. Il faut donc que la ligne partagée soit mise à jour ou invalidée dans chaque cache privé de lecteur. Il n'est pas envisageable de faire traiter cette mise à jour par le programme applicatif lui-même. C'est pourquoi un protocole de cohérence des caches doit être implémenté en matériel, ou dans le système d'exploitation [5] :

- Pour les protocoles de cohérence logiciels : le système d'exploitation (ou les bibliothèques systèmes) contrôle toutes les pages du système mémoire. Le noyau du système d'exploitation connaît le placement des pages partagées, il est donc capable, en principe, de gérer la cohérence entre les différentes copies.
- Pour les protocoles de cohérence matériels : le protocole de cohérence est implémenté dans les contrôleurs des caches qui gèrent la hiérarchie mémoire et la modification d'une ligne de cache provoque des échanges de message entre les caches de différents niveaux.

Les protocoles de cohérence logiciels imposent des contraintes fortes. En particulier, la granularité de partage de données est la page de mémoire virtuelle, alors que pour les protocoles de cohérence des caches matériels, la granularité de partage est la ligne de cache. Dans cette thèse, nous nous intéressons aux protocoles de cohérence matérielle. Il en existe deux grandes classes : *Snoop* et *Directory* [6] :

Protocole *Snoop*

Le protocole *Snoop* est largement utilisé par les architectures **SMP**. L'idée principale est que chaque cache observe (*Snoop* en anglais) les requêtes de cohérence émises par les autres caches afin de mettre à jour l'état de ses propres lignes.

Le protocole *Snoop* a une latence de mise à jour très basse. Il est, par ailleurs, simple à l'implémenter. Toutefois il fait l'hypothèse de l'existence d'un *bus* partagé. C'est là le principal désavantage. En effet, sur un *bus* la bande passante est bornée et doit être partagée entre tous les cœurs. Le nombre de coeurs connectés sur un *bus* ne dépasse généralement pas une dizaine de coeurs.

Protocole *Global Directory*

Le protocole *Global Directory* est implémenté dans l'architecture **NUMA** pour résoudre le problème du passage à l'échelle du protocole *Snoop*.

Le protocole *Global Directory* est conceptuellement implanté du côté du contrôleur mémoire, qui gère un répertoire global de l'état des toutes les lignes de cache. Ce répertoire global contient précisément le nombre de copies de chaque ligne, voire la localisation de chaque copie. Chaque contrôleur de cache informe directement le répertoire global des modifications qu'il fait sur ses propres copies. Le répertoire global envoie alors les requêtes de cohérence vers les autres contrôleurs de cache concernés.

Avec le protocole *Global Directory*, c'est le répertoire global, par lequel passent toutes les demandes des contrôleurs de cache qui va garantir l'ordre du traitement des requêtes de cohérence. Ce protocole est donc bien adapté aux architectures *manycore* **NUMA**. Actuellement, la plupart des architectures multiprocesseurs industrielles possédant plus d'une dizaine de coeurs implémentent le protocole *Global Directory*.

1.1.4 Stratégies d'écriture

Quand une tâche s'exécutant sur un coeur effectue une écriture, deux politiques [7] de mise à jour (co-)existent, pour ce qui concerne le cache de premier niveau (cache L1) :

La stratégie d'écriture immédiate (*Write-Through* en anglais)

Avec cette stratégie, toutes les requêtes d'écriture sont directement transmises au contrôleur de cache de niveau supérieur. La donnée modifiée est enregistrée dans le cache L1 seulement si la ligne de cache concernée est déjà présente dans le cache. Le cache de niveau supérieur contient donc toujours les valeurs les plus récentes.

La stratégie d'écriture différée (*Write-Back* en anglais)

Avec cette stratégie, les requêtes d'écriture sont toujours effectuées dans le cache L1, et ce, même si la ligne de cache concernée n'est pas encore présente dans le cache local. Le cache au niveau supérieur n'est mis à jour que plus tard lors de l'évincement de la ligne du cache L1. Il s'ensuit que le cache L1 doit se souvenir de l'état de modification de chaque ligne par exemple par l'ajout d'un bit *Dirty* dans le répertoire du cache. Le cache de niveau supérieur ne contient donc plus nécessairement les valeurs les plus récentes.

Le protocole de cohérence des caches matériel dépend fortement du choix de la stratégie d'écriture :

- Malgré le fait que l'écriture immédiate génère un plus grand nombre de requêtes d'écriture, elle simplifie significativement le protocole de cohérence des caches. En effet, les données partagées par les cœurs sont toujours à jour dans le cache partagé de niveau supérieur, il peut donc répondre aux requêtes de lecture directement.
- La stratégie d'écriture différée supprime les transactions d'écriture inutiles, car portant sur des données non partagées, mais elle complexifie le protocole de cohérence des caches. Lorsqu'une ligne de cache X est modifiée localement dans un cache L1 privé, le cache partagé est obligé d'envoyer une requête de cohérence pour ramener les données à jour de la ligne X quand un autre cache local privé veut lire la même ligne X.

1.1.5 Mémoire virtuelle

L'objectif de la mémoire virtuelle [8] est de simplifier la gestion de l'espace d'adressage pour les applications (les processus). Elle permet de donner l'illusion au programmeur (ou au compilateur) qu'un processus dispose pour lui seul de tout l'espace d'adressage. Elle permet de masquer l'exécution simultanée d'autres processus, car tous peuvent utiliser les mêmes adresses sans conflit. Elle permet aussi de proposer un espace accessible plus grand que la taille de la mémoire physique en utilisant de manière transparente le disque dur externe si nécessaire. Ainsi, chaque programme peut utiliser un espace d'adressage complet et consécutif sans se soucier de la mémoire physique réellement disponible.

Dans tous les systèmes d'exploitation modernes, cet espace d'adressage est partitionné en pages. Chaque page virtuelle (représentant un contenu) est associée à une page physique (représentant le contenant) rangée dans la mémoire physique. En cas de manque de place en mémoire, une page virtuelle peut être temporairement stockée sur le disque dur. Dans le cas où le disque est utilisé, la mémoire physique est une sorte de cache de l'union des espaces virtuels des différents processus en cours d'exécution.

De plus, à chaque page virtuelle sont associés des droits d'accès tels que *Read_Only* ou *Kernel_Mode*. De même, la mémoire virtuelle permet la protection des périphériques lorsque ceux-ci sont commandés par des registres adressables dans l'espace physique.

La page physique contenant ces registres peut être associée ou pas à une page virtuelle d'un processus avec des droits d'accès spécifiques. En gérant ainsi les droits d'accès de page, la mémoire virtuelle permet de restreindre l'accès des périphériques afin de renforcer la sécurité.

Dans les architectures *manycore*, le mécanisme de pagination peut participer au protocole de cohérence des caches en permettant au système d'exploitation d'enregistrer certaines caractéristiques telles que `privées` ou `partagées` dans la table des pages. Une page ***privée*** ne doit contenir que des données utilisées par un seul cœur, par exemple toutes les pages de la pile d'exécution d'un processus placé sur un cœur, sans possibilité de migration sont `privées`. Une page `partagée` peut contenir des données utilisées par plusieurs cœurs. Lors de l'accès à la mémoire par un cœur, le contrôleur de cache de premier niveau peut savoir si la donnée concernée est `privée`, donc sans besoin de cohérence, ou `partagée`, donc avec besoin de cohérence. Ceci permet de réduire le trafic de cohérence, mais suppose l'aide du système d'exploitation pour distinguer explicitement les données `privées` ou `partagées`.

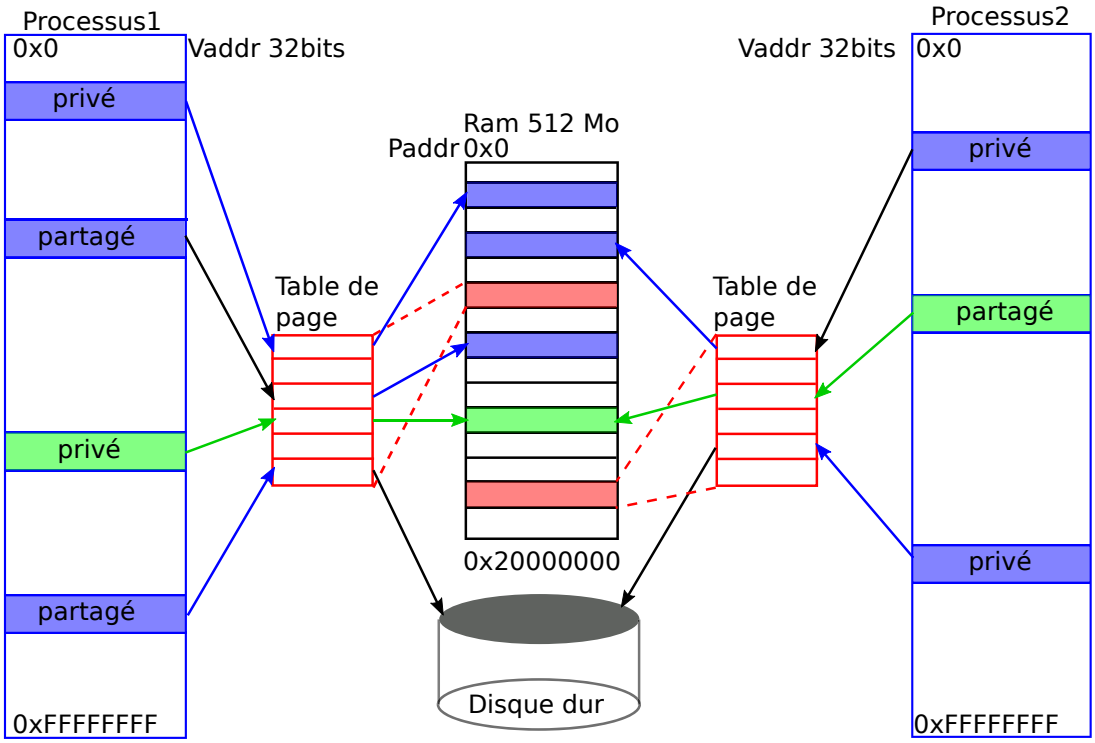


FIGURE 1.4 – Système de mémoire virtuelle

La figure 1.4 représente les espaces d'adressage de deux processus se partageant la mémoire et le disque dur. Les tables de traduction d'adresse contiennent la correspondance numéro de page virtuelle (**VPN**) vers numéro de page physique (**PPN**).

La figure 1.5 montre qu'une table de traduction d'adresse ne contient pas seulement les numéros des pages physiques, mais aussi des droits d'accès et des attributs sur le degré de partage des pages.

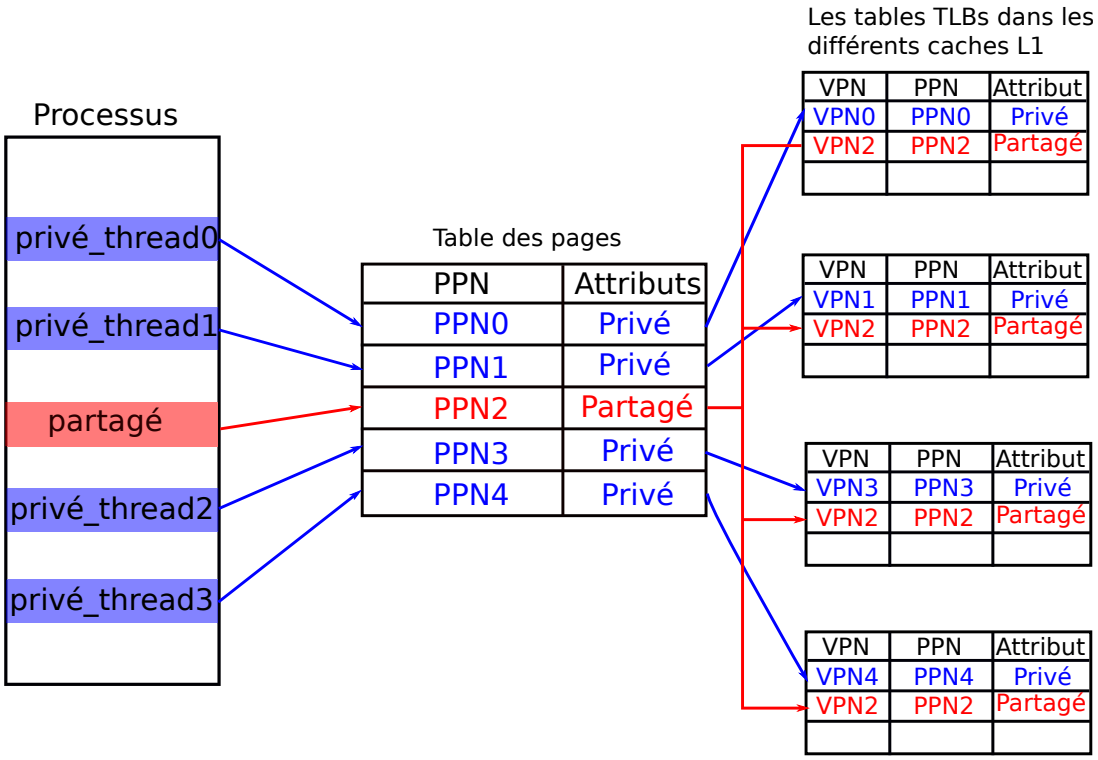


FIGURE 1.5 – Principe de protocole de cohérence des caches à base de mémoire virtuelle

La **MMU** *Memory Management Unit* assure la conversion des adresses virtuelles en adresses physiques en consultant une table des pages (*page table* en anglais) propre à chaque processus. Chaque entrée de la table des pages (**PTE** : *Page Table Entry* en anglais) associe un **VPN** (*Virtual Page Number* c.-à-d. le numéro de page virtuelle) à un **PPN** (*Physical Page Number* c.-à-d. le numéro de page physique).

La **TLB** (*Translation Lookaside Buffer*) est un petit cache de traduction d'adresses présent dans la **MMU** permettant d'accélérer la traduction des adresses virtuelles en adresses physiques. Chaque entrée de la **TLB** sauvegarde temporairement des couples **VPN-PPN** ainsi que les droits et états associés aux pages. Lorsqu'un cœur accède à une adresse virtuelle, la **TLB** est consultée et si le numéro de page virtuelle demandé est présent dans la **TLB** alors le numéro de page physique concerné est connu. Dans ce cas, il n'est plus nécessaire de consulter la table des pages.

Lorsque la **TLB** ne contient pas le numéro de page virtuelle demandé, alors le système d'exploitation, ou le matériel, doit parcourir la table des pages (*Page Table Walk* en anglais) pour y trouver le numéro de page physique, ses droits et états et mettre à jour la **TLB**.

1.1.6 Cohérence des TLBs

Dans une application multitâches en mémoire virtuelle, une page d'instructions ou de données partagées peut être accédée par plusieurs cœurs. Par conséquent, le numéro de la page virtuelle partagée peut être stocké dans plusieurs **TLBs**. Lorsque l'association **VPN-PPN** est modifiée par le système d'exploitation (*unmap* une page) ou que les droits ou l'état de la page sont modifiés par le matériel (mis à jour du bit *Dirty* marquant une

modification de la page), alors les entrées de la **TLB** doivent être invalidées ou mises à jour. Il est donc nécessaire de prévoir une solution pour la gestion de la cohérence des **TLBs**.

Le plus souvent, la cohérence des **TLBs** est assurée par le système d'exploitation grâce au mécanisme ***TLB** Shootdown* [9]. Dans ce mécanisme, lorsque le système d'exploitation veut modifier une entrée de la table des pages, il verrouille cette entrée et envoie une interruption aux cœurs susceptibles de contenir cette entrée dans leur **TLB** (nommé **IPI** *Inter Processor Interruption*). Le service exécuté en réponse à l'interruption doit invalider l'entrée concernée. Il est donc nécessaire que le système d'exploitation connaisse la liste des cœurs contenant l'entrée modifiée ou qu'il utilise une méthode d'invalidation générale (`Broadcast Invalidate`). Le mécanisme interruptif ***TLB** Shootdown* s'adapte bien à des architectures multicœurs construites autour d'un *bus*, mais plus difficilement aux architectures *manycore*, car ce mécanisme ne passe pas à l'échelle.

Pour résoudre le problème de la cohérence des caches dans les architectures *manycore*, il faut que le protocole de cohérence des **TLBs** soit également pris en charge par le matériel.

1.2 L'architecture TSAR

L'architecture **TSAR** [10] (*Tera-Scale ARchitecture*) développée conjointement par BULL, le Lip6 et le CEA-LETI est une architecture *Manycore* ***CC-NUMA*** extensible jusqu'à 1024 cœurs.

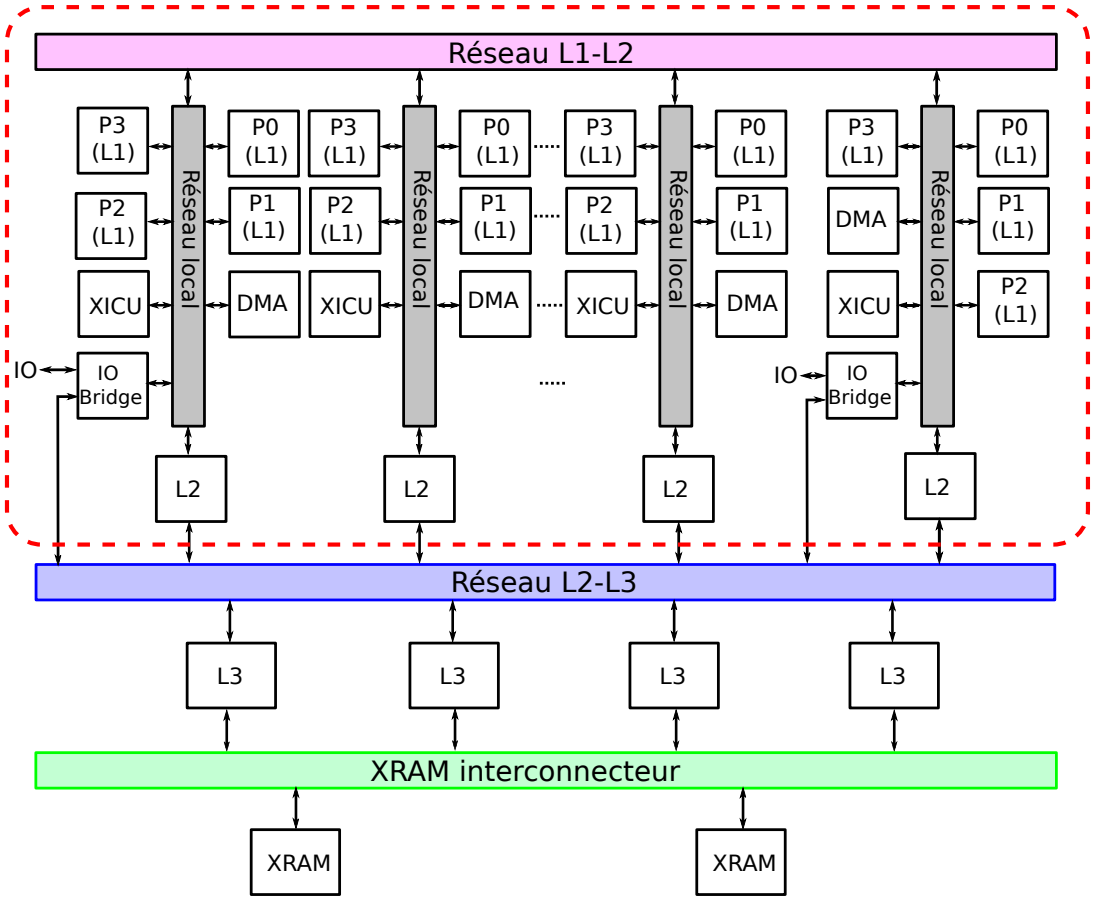


FIGURE 1.6 – L'architecture **TSAR**

Le passage à l'échelle a été permis par le choix d'un protocole de cohérence des caches à base de répertoire utilisant une stratégie d'écriture immédiate. Ce protocole, appelé **DHCCP** (*Distributed Hybrid, Cache Coherence Protocol*), garantit à la fois la cohérence des caches de premier niveau et la cohérence des **TLBs**. Ce protocole totalement matériel passe à l'échelle, puisqu'il a permis de déployer des applications parallèles multitâches sur des plateformes matérielles comportant jusque 1024 coeurs.

TSAR est une architecture clustérisée utilisant un **NOC** (*Network On Chip*) ayant une topologie en *mesh* 2D (figure 1.6). Chaque cluster contient 4 cœurs. Chaque cœur contient un processeur MIPS(32bits) RISC (*Reduced instruction set computer*) sans *Superscalar*, pas d'exécution dans le désordre, et pas de prédiction de branchement, aucune exécution spéculative et un cache privé de niveau 1. Chaque cluster contient aussi un segment de mémoire physique est stocké uniquement dans un cache de niveau 2. L'architecture **TSAR** supporte des systèmes d'exploitation généralistes de type UNIX (comme LINUX ou NetBSD). La thèse de Ghassan Almaless [11] sur un système d'exploitation de type UNIX a démontré sur un ensemble d'applications que le protocole **DHCCP** était scalable jusqu'à 1024 cœurs.

1.2.1 Hiérarchie des caches

L'architecture **TSAR** supporte trois niveaux des caches [12], mais la version utilisée dans cette étude implémente un système de cache à deux niveaux (Figure 1.7).

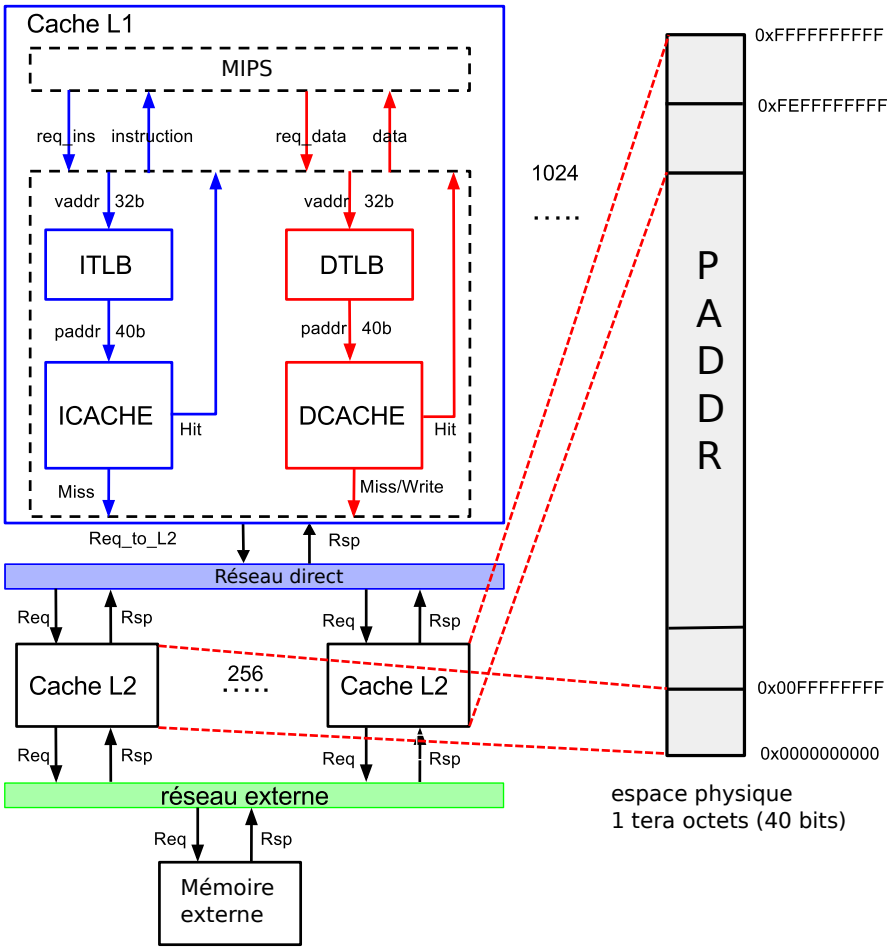


FIGURE 1.7 – Hiérarchie mémoire

Le premier niveau de cache (caches L1) [13] est privé par cœur. Le deuxième niveau de cache (cache L2) [12] est partagé par tous les caches L1. Afin de réduire la latence d'accès et d'augmenter la bande passante, le cache L2 est physiquement distribué dans tous les clusters, chaque cluster étant en charge d'un segment de mémoire physique.

Cache L1

Les instructions et les données sont gérées séparément par un cache des instructions (16 Ko : 64 *sets* * 4 *ways* * 64 octets) et un cache des données (16 Ko, 64 *sets* * 4 *ways* * 64 octets).

- Le contrôleur de cache L1 renvoie directement la réponse au processeur lorsque la ligne de cache accédée est présente dans le cache L1. Sinon, en cas de *Miss* en lecture, le contrôleur de cache L1 demande au cache de niveau supérieur (le cache L2) une copie de la ligne. Le contrôleur de cache L1 doit trouver une place pour la ligne demandée et doit, le plus souvent, évincer une ligne déjà présente lorsque toutes les lignes d'un ensemble (set) sont occupées. Le contrôleur choisi prioritairement une ligne non récemment utilisée (mécanisme **LRU** pour *Least Recently Used*).

- Le cache L1 utilise une stratégie d'écriture immédiate et transmet donc toutes les écritures au cache L2, ce qui simplifie énormément le protocole de cohérence et permet le passage à l'échelle.

Cache L2

TSAR gère un espace d'adressage physique de 1 To. Cet espace est partitionné en autant de segments qu'il y a de clusters. Chaque cache L2 est en charge d'un segment. Les adresses physiques font 40 bits, les 8 bits de poids fort permettent d'identifier un cluster, les 32 bits de poids faible désignent un octet dans ce cluster. Dans la configuration maximale, on a donc 256 clusters et chaque cluster gère un segment de 4 Go de mémoire physique. Les caches L2 sont partagés par tous les caches L1, ce qui signifie que n'importe quel cache L1 peut contenir n'importe quelle ligne de cache de l'espace 40 bits. Les caches L2 sont inclusifs, ce qui signifie qu'une ligne de cache présente dans un ou plusieurs caches L1 est nécessairement présente dans le cache L2 qui gère le segment auquel cette ligne appartient. Toutes les requêtes sur une ligne de cache envoyées par n'importe quel cache L1 sont transmises au cache L2 qui gère la ligne. Le répertoire d'un cache L2 contient exclusivement les informations d'utilisation de ses propres lignes pour gérer leur cohérence. La capacité de chaque cache L2 est de 256 Koctets : 256 *sets* * 16 *ways* * 64 octets.

Grâce à la stratégie écriture immédiate, les caches L2 contiennent toujours les valeurs les plus récentes (par rapport aux caches L1). En revanche, la mémoire principale ne contient pas toujours la ligne de cache la plus à jour, car le cache L2 utilise une stratégie d'écriture différée en mémoire pour réduire la consommation énergétique. Deux types de transactions sont envoyées à la mémoire principale :

1. Requête de lecture – lorsque le cache L2 reçoit une requête de lecture ou d'écriture correspondantes une ligne de cache non présente, alors le cache L2 envoie la transaction de lecture à la mémoire principale afin de ramener la ligne manquante.
2. Requête d'écriture – le cache L2 envoie une transaction d'écriture dans le cas où une ligne doit être évincée et que cette ligne a été modifiée depuis son chargement dans le cache L2.

Chaque cache L2 pouvant être potentiellement sollicité par tous les caches L1, le contrôleur de cache L2 doit avoir le comportement le plus parallèle possible, pour minimiser la contention : Dans le cache L2, toutes les transactions vers la mémoire principale sont enregistrées dans une table nommée ***table TRT*** (*TRansaction Table*) de façon à mettre en pipeline ces transactions. Cette table peut stocker jusqu'à 8 transactions en cours. Les transactions ne restent dans la ***table TRT*** que le temps de leur traitement par la mémoire. Le contrôleur de cache L2 vérifie que deux transactions ne correspondent pas à la même ligne de cache.

Les requêtes de lecture demandées par les caches L1 au cache L2 sont traitées par un automate spécifique **`READ`**. Lorsque la ligne demandée est absente, il sauvegarde l'intégralité de la requête dans la ***table TRT*** et ensuite il commence le traitement de la requête suivante. Un autre automate nommé **`XRAM_RSP`** prend en charge la mise à jour de la ligne dans le cache L2 et la réponse au cache L1 dès la réponse de la mémoire principale. Les requêtes d'écriture demandées par les caches L1 sont traitées par un troisième automate **`WRITE`**. Si la requête d'écriture concerne une ligne partagée,

et qu'une opération de cohérence est nécessaire, elle est sous-traitée à deux autres automates, et l'acquittement de la requête d'écriture n'est effectué que quand l'opération de cohérence est terminée. (La procédure de ces automates est décrite dans la thèse de Éric Guthmuller [12])

1.2.2 Réseaux d'interconnexion

La figure 1.8 montre la structure des réseaux intégrés de l'architecture **TSAR**. Il y a deux réseaux, interne et externe.

Le réseau externe assure la communication entre les caches L2 et les bancs de mémoire externes (ou les caches L3 dans la version complète de **TSAR**). Pour éviter les interblocages, ce réseau externe se décompose en deux sous-réseaux : réseau des commandes et réseaux des réponses.

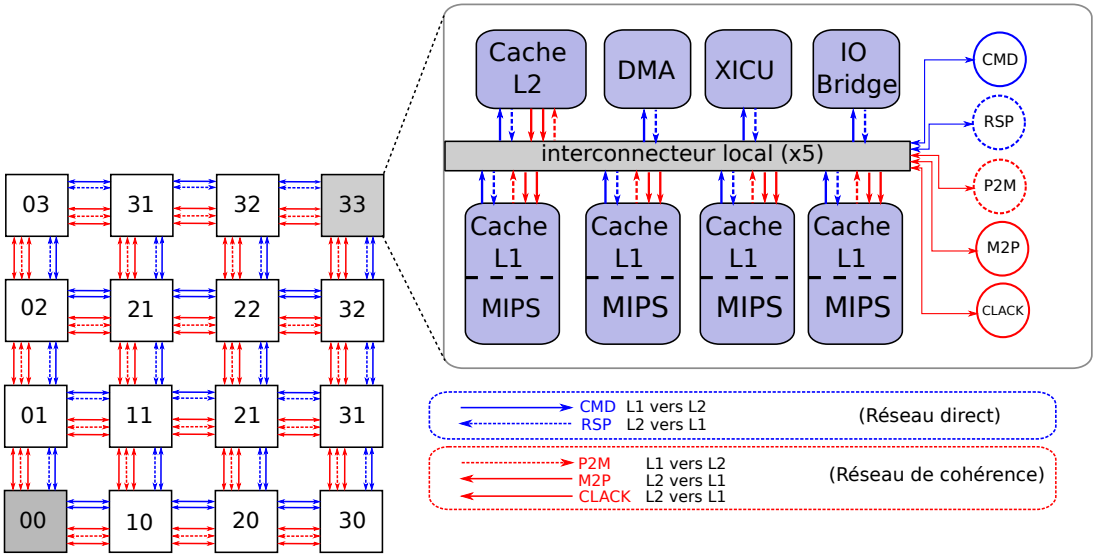


FIGURE 1.8 – Réseaux d'interconnexion

Le réseau interne permet la communication entre les caches L1 et les caches L2. Pour éviter les interblocages, ce réseau interne se décompose en 5 sous-réseaux indépendants permettant d'acheminer en parallèle 5 types de trafic, dont trois sous-réseaux pour le trafic de cohérences.

1. réseau **CMD** (L1 -> L2)
   permet d'acheminer les commandes de lectures ou d'écritures.
2. réseau **RSP** (L2 -> L1)
   permet d'acheminer les réponses à ces commandes.
3. réseau **M2P** (L2 -> L1)
   permet d'acheminer les requêtes de cohérence de type `Invalidate` ou `Update`.
4. réseau **P2M** (L1 -> L2)
   permet d'acheminer les requêtes de cohérence de type `Cleanup`.
5. réseau **CLACK** (L2 -> L1)
   permet d'acheminer les requêtes de cohérence de type `Clack`.

Chaque sous-réseau interne est structuré en deux parties : (1) un *crossbar* local connecte tous les composants internes d'un même cluster ; (2) un *mesh* 2D connecte les clusters entre eux.

1.3 Protocole DHCCP dans l'architecture TSAR

Le protocole **DHCCP** repose sur le principe du répertoire global distribué. Chaque ligne de cache X présente dans un cache L2 peut avoir 0, 1, ou n copies dans les caches L1. Pour chaque ligne X, le répertoire du cache L2 contient soit l'emplacement des copies (une liste chaînée contenant les numéros identifiants des caches L1), soit le nombre de copies (un compteur) si n dépasse un certain seuil (le seuil est égal 3 par défaut).

Le protocole **DHCCP** utilise une stratégie d'écriture immédiate. Chaque écriture effectuée par un cœur est immédiatement envoyée par le contrôleur du cache L1 au cache L2 gestionnaire de la ligne dans laquelle se fait l'écriture sur le réseau **CMD**. Par conséquent, une ligne de cache dans le cache L2 est toujours à jour.

1.3.1 États d'une ligne dans le cache L2

Pour le cache L2, une ligne valide ne peut donc être que dans deux états `REPLICATED` (une ou plusieurs copies), ou `NON REPLICATED` (0 copie). Pour une ligne X `REPLICATED`, le contrôleur de cache L2 enregistre l'information suivante :

- si le nombre de copies est inférieur au seuil, non seulement le nombre de copies de la ligne X est sauvegardé, mais aussi l'identifiant du cache L1 possédant la copie est enregistré dans une liste chaînée gérée entièrement en matériel, dans une mémoire locale (nommée ***HEAP***) non adressable par le logiciel, et capable d'enregistrer 4096 identifiants pour ensemble des lignes de cache présentes dans le L2.
- Si, pour une ligne X, le nombre de copies est égal ou supérieur au seuil, ou si le ***HEAP*** est plein, le contrôleur de cache L2 libère la liste chaînée correspondante à la ligne X et sauvegarde seulement le nombre de copies dans le répertoire du cache L2.

1.3.2 Les 4 types de transactions de cohérence

Dans le protocole **DHCCP**, le contrôleur de cache L2 déclenche une transaction de cohérence s'il reçoit une requête d'écriture sur une ligne `REPLICATED`. Ou s'il évince une ligne `REPLICATED` pour faire de la place. Pour sa part, le contrôleur de cache L1 déclenche une transaction de cohérence s'il évince une ligne pour faire de la place.

Globalement, le protocole définit quatre types de transactions de cohérence, qui utilisent 6 types de requêtes (figure 1.9).

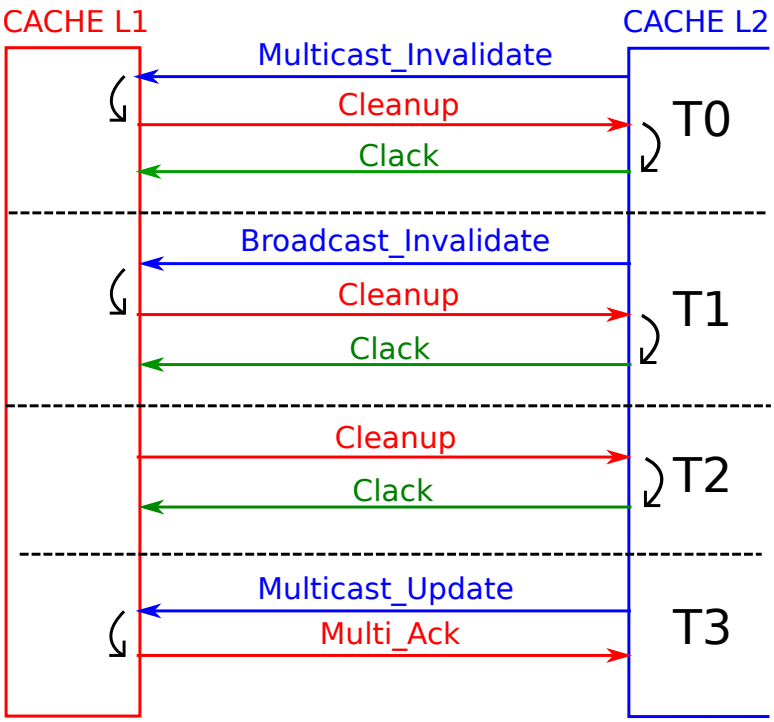


FIGURE 1.9 – Transactions de cohérence des caches

Transaction T0

Lorsque toutes les lignes d'un ensemble (*set*) sont occupées, et a besoin d'une case pour enregistrer une nouvelle ligne, il doit évincer une ligne X. Lorsque la ligne évincée est `REPLICATED` et en mode ***liste chaînée***, les caches L1 concernés doivent invalider cette ligne pour maintenir la propriété d'inclusivité des L1 dans les L2. La transaction **T0** nécessite trois requêtes :

- requête *`Multicast Invalidate`*(X) (L2 → L1) sur le canal **M2P**. Puisque tous les identifiants de cache L1 contenant la copie de la ligne X sont enregistrés dans la ***liste chainée***, le contrôleur de cache L2 envoie une requête *`Multicast Invalidate`*(X) à chaque cache L1 concerné. Dès que le cache L2 a envoyé cette requête, les informations concernant les copies de la ligne X sont éliminées du répertoire du cache L2. Si le cache L2 reçoit une requête de lecture *`Read`*(X), cette requête est mise en attente jusqu'à ce que la transaction **T0** soit terminée. Pour permettre de paralléliser les transactions de cohérence **T0**, et détecter la fin de ces transactions, le cache L2 dispose d'une table d'invalidation à 8 entrées, appelée ***table IVT*** (pour *InValidation Table*) qui contient le nombre de copies des lignes en cours d'invalidation. Lorsque le cache L2 reçoit une réponse à une requête *`Multicast Invalidate`*(X), le compteur de copies pour la ligne X est décrémenté. La ***table IVT*** supprime l'entrée sur la ligne X lorsque le compteur de copie atteint 0, et le contrôleur de cache L2 débloque la requête *`Read`*(X) en attente.
- requête *`Cleanup`*(X) (L1 → L2) sur le canal **P2M**. Lorsque le cache L1 reçoit une requête *`Multicast Invalidate`*(X), le traitement dépend de l'état de la ligne X dans le cache L1. Si la ligne est présente, le cache L1 invalide la copie

locale et envoie la requête `Cleanup`(X) au cache L2 pour signaler cette invalidation. Puisque la ligne X est à jour dans le cache L2, la requête `Cleanup`(X) ne contient que l'adresse de la ligne X. Si le cache L1 ne contient pas la ligne X, c'est qu'elle a été spontanément évincée. Dans ce cas, le cache L1 ne répond pas la requête `Multicast Invalidate`(X), parce que la requête `Cleanup`(X) a déjà été envoyée au cache L2. Si le cache L1 vient d'effectuer une requête de lecture `Read`(X), il est possible qu'il reçoive une requête de cohérence `Multicast Invalidate`(X) avant la réponse à sa commande `Read`(X), puisque la réponse à la commande de lecture et la requête de cohérence sont transmises sur deux réseaux séparés. Dans ce cas, le cache L1 envoie la requête `Cleanup`(X) lorsqu'il reçoit la réponse à sa commande de lecture, mais la ligne X est jetée.

- requête `Clack`(X) (L2 → L1) sur le canal **CLACK**. Cette requête est systématiquement envoyée par le cache L2 pour acquitter une requête `Cleanup`(X) reçue d'un cache L1. Nous expliquons ici pourquoi cet acquittement est indispensable. Supposons que le cache L1 vient d'évincer la ligne X et d'envoyer la requête `Cleanup`(X) au cache L2, mais que le processeur redemande à lire une valeur contenue dans la ligne X. Puisque la ligne X n'est plus présente, le cache L1 doit envoyer une requête de lecture `Read`(X) au cache L2. Puisque la requête `Cleanup`(X) et la requête `Read`(X) sont transmises dans les deux canaux séparés **P2M** et **CMD**, le cache L2 pourrait recevoir la requête `Read`(X) avant la requête `Cleanup`(X), en conséquence le répertoire du cache L2 se trouverait dans une situation incohérente puisque pour lui la ligne X posséderait deux copies dans le même cache L1. Pour éviter cette situation, le cache L1 s'interdit d'envoyer une requête `Read`(X) tant que le cache L2 n'a pas acquitté la requête `Cleanup`(X). Une ligne en cours de `Cleanup` dans le cache L1 passe donc temporairement dans un état `Zombie` qui bloque toute tentative de lecture par le processeur, mais bloque également toute requête de lecture vers le cache L2. C'est la réception de la requête `Clack`(X) qui change l'état de la ligne X de `Zombie` vers `Invalid`. Pour faciliter ce changement d'état, la requête `Clack`(X) contient également le numéro de voie (*way*) dans le cache L1, qui doit donc être transmis dans la requête `Cleanup`(X).

Transaction T1

Lorsque le cache L2 reçoit une requête d'écriture sur une ligne X en mode compteur ou évince cette ligne pour faire de la place, toutes les copies de cette ligne dans les caches L1 doivent être invalidées. La transaction **T1** nécessite également 3 requêtes :

- requête `Broadcast Invalidate`(X) (L2 → L1) sur le canal **M2P**. Puisque la ligne X est en mode ***compteur***, le cache L2 n'enregistre plus l'emplacement des copies, mais seulement le nombre de copies. Le contrôleur de cache L2 envoie donc la requête `Broadcast Invalidate`(X) à tous les caches L1 de la plateforme, en utilisant la fonctionnalité *Broadcast* du réseau **M2P**.
- requête `Cleanup`(X) (L1 → L2) sur le canal **P2M**. Le cache L1 traite la requête `Broadcast Invalidate`(X) de la même façon que la requête `Multicast Invalidate`(X) : seuls les caches L1 possédant une copie de la ligne X renvoient une requête `Cleanup`(X) au cache L2. Ceci permet au cache L2 de compter les réponses dans la ***table IVT***, et de détecter la fin de la transaction **T1**.

- requête `Clack`(X) (L2 → L1) sur le canal **M2P**. Comme pour la transaction **T0**, le cache L2 doit acquitter chaque requête `Cleanup`(X) reçue, en envoyant une requête `Clack`(X) afin de changer l'état de la ligne de `Zombie` à `Invalid` dans le cache L1.

Transaction T2

Cette transaction est déclenchée par le cache L1 pour signaler au cache L2 l'évincement spontané d'une ligne X. Ceci permet au cache L2 d'avoir un état exact du nombre de copies de la ligne X dans la machine. Cette transaction **T2** nécessite deux requêtes :

- requête `Cleanup`(X) (L1 → L2) sur le canal **P2M**.
- requête `Clack`(X) (L2 → L1) sur le canal **M2P**.

Lorsque toutes les lignes d'un ensemble (*set*) sont occupées dans le cache L1, et a besoin d'une case pour enregistrer une nouvelle ligne, il y a une transaction **T2** pour chaque *Miss*, et le nombre de transactions **T2** est donc important. Ceci justifie que toutes les requêtes `Cleanup`(X) soient traitées par un automate **`CLEANUP`** dédié dans le cache L2. Cet automate lit le répertoire du cache L2 pour vérifier si la ligne est présente. Si la ligne X est valide (la requête `Cleanup`(X) correspond à un évincement spontané), l'automate **`CLEANUP`** décrémente le compteur de copies. De plus, il enlève l'identifiant du cache L1 de la liste si la ligne X est en mode ***liste chainée***. Lorsque la ligne X n'est pas présente dans le cache L2, elle est enregistrée dans la ***table IVT***car cette requête `Cleanup`(X) fait partie d'une transaction **T0** ou **T1**, et l'automate **`CLEANUP`** décrémente le compteur pour la ligne X dans la ***table IVT***.

Transaction T3

Si le cache L2 reçoit une requête d'écriture `Write`(X) sur une ligne X `REPLICATED` en mode ***liste chaînée***, il connaît les identifiants des caches L1 contenant des copies, et peut envoyer les valeurs modifiées aux caches L1 concernés, pour une mise à jour des copies. Cette transaction **T3** nécessite 2 requêtes :

- requête `Multicast Update`(X,DATA) (L2 → L1) sur le canal **M2P**.
  Cette requête est envoyée explicitement à chaque cache L1 contenant une copie. Elle contient l'adresse de la ligne X et les valeurs modifiées par la commande d'écriture pour chaque octet de la ligne X (cela peut représenter jusque 64 octets). Puisque ces requêtes sont déclenchées par une commande d'écriture, une des contraintes imposées par le protocole **DHCCP** concernant les rafales d'écriture est donc que toutes les écritures faisant partie d'une même rafale appartiennent à la même ligne de cache.
- requête `Multi Ack`(X) (L1 → L2) sur le canal **P2M**.
  Cette requête permet à chaque cache L1 de signaler au cache L2 que la requête `Multicast Update`(X,DATA) a bien été effectuée. Le cache L1 met à jour sa copie locale de la ligne X, et envoie ensuite la requête `Multi Ack`(X). Si la ligne n'est pas présente dans le cache L1 car la ligne a été évincée, le cache L1 envoie quand même la requête `Multi Ack` au cache L2. Pour paralléliser les transactions **T3**, le cache L2 contient une table à 8 entrées nommée ***table UPT*** (comme

UPdate Table) qui enregistre pour chaque transaction **T3** en cours, le nombre de réponses attendues. Lorsque le cache L2 a reçu toutes les requêtes `Multi Ack`(X) correspondantes à une transaction **T3**, il retire cette transaction de la ***table UPT*** et peut finalement acquitter la commande d'écriture `Write`(X) qui a déclenché cette transaction de cohérence.

1.3.3 États d'une ligne dans le cache L1

La figure 1.10 présente le graphe simplifié des transitions entre états pour une case du cache L1. Une entrée du cache L1 identifiée par le couple (*set*, *way*).

Elle passe de l'état `Invalid` à l'état `Valid` lorsqu'une ligne de cache manquante est copiée dans cette entrée (transition *a*). Elle reste dans cet état tant qu'elle n'est pas évincée (transition *b*). Elle passe de l'état `Valid` à l'état `Zombie` lorsqu'elle doit être invalidée, suite à un évincement spontané (manque de place), ou suite à une requête de cohérence d'invalidation envoyée par le cache L2 (transition *c*). Le cache L1 envoie une requête `Cleanup` pour la ligne X contenue dans l'entrée (*set*,*way*). Dans l'état `Zombie`, la ligne X ne peut plus être utilisée par le processeur, mais l'entrée (*set*,*way*) ne peut pas encore être utilisé pour stocker une autre ligne cache. Elle ne passe de l'état `Zombie` à l'état `Invalid` (c'est à dire utilisable pour stocker une nouvelle ligne) que lorsque le cache L1 reçoit l'acquittement `Clack` (transition *d*).

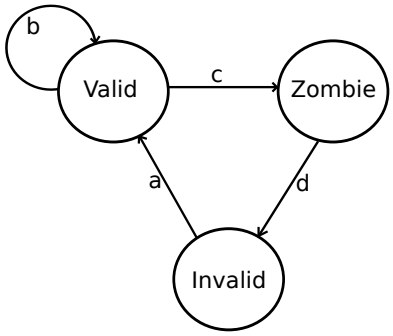


FIGURE 1.10 – Diagramme des transitions des états du cache L1

1.4 Cohérence des TLBs dans l'architecture TSAR

Pour la mémoire virtuelle, l'architecture **TSAR** implémente des pages de 4 Koctets. L'architecture **TSAR** supporte également des grandes pages de 2 Moctets, utilisées par les systèmes d'exploitation, qu'il n'est pas utile de présenter ici. Les adresses physiques sont sur 40 bits, mais les adresses virtuelles font 32 bits car, pour minimiser la consommation, **TSAR** utilise des processeurs **RISC** 32 bits. Le **VPN** est donc codé sur 20 bits et le **PPN** est codé sur 28 bits.

Les tables des pages - construites par le système d'exploitation - ont une structure à deux niveaux, et sont rangées dans la mémoire principale. La table de premier niveau contient 2048 entrées de 32 bits indexés par les 11 bits de poids fort du numéro de page virtuel (**VPN**). Chaque entrée contient un **PTD** (*Page Table Descriptor*), qui est un pointeur sur une table de 2e niveau. Une table de deuxième niveau contient 512 entrées **PTE** (*Page Table Entry*) de 64 bits. Elle est indexée par les 9 bits de poids faible du **VPN**. Chaque **PTE** contient la valeur du **PPN** associé au **VPN** et quelques bits définissant les droits d'accès et l'état courant de la page (figure 1.11).

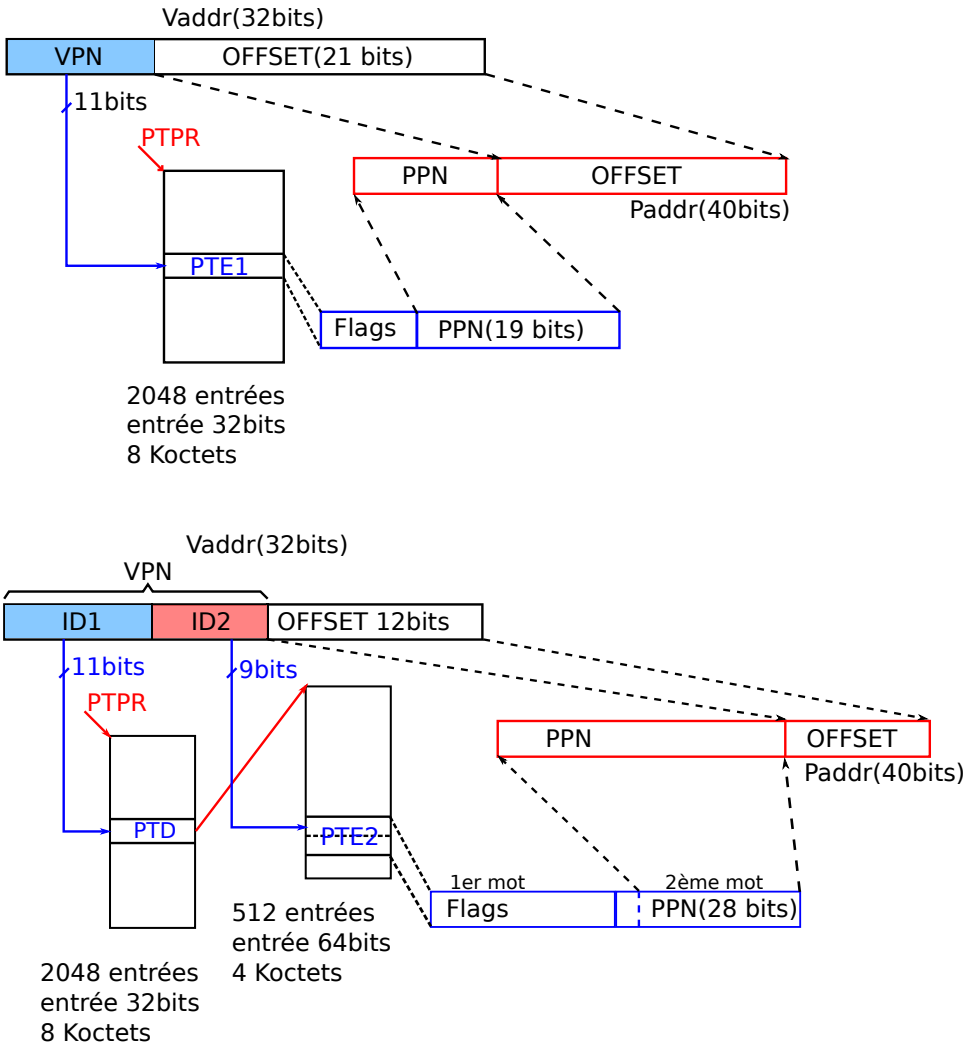


FIGURE 1.11 – Hiérarchie page mémoire

1.4.1 MMU générique de TSAR

L'unité gestionnaire de mémoire virtuelle (**MMU**) chargée de la traduction **VPN** vers **PPN** est placée dans le contrôleur de cache L1, entre le cœur et les deux caches L1, données et instructions. Cette **MMU** est générique, car elle peut supporter différents types de cœur **RISC** 32 bits. Elle contient deux **TLBs** : une **TLB** pour les adresses instructions (64 entrées = 8 *sets* * 8 *ways*) et une autre **TLB** pour les adresses de données (64 entrées = 8 *sets* * 8 *ways*). Les évènements de type *Miss* **TLB** (quand une paire **VPN->PPN** est absente d'une **TLB**) sont gérés par un automate matériel (appelé *Table Walk*) sans intervention du système d'exploitation.

1.4.2 Cohérence des TLBs dans TSAR

La thèse de Yang GAO [13] a étendu le protocole **DHCCP** pour gérer la cohérence des **TLBs**. Les idées principales de la méthode sont les suivantes :

- Les tables des pages sont cachables dans les caches L1 des coeurs. En cas de *Miss*

TLB, l'automate **`TABLE_WALK`** s'adresse directement au contrôleur du cache L1 local pour obtenir l'entrée manquante dans la **TLB**.

- Puisque le protocole **DHCCP** garantit la cohérence entre les caches L1 et les caches L2, toute modification d'une table des pages par le système d'exploitation sera répercutée vers tous les caches L1 contenant une copie de la ligne de cache concernée.
- Ceci permet d'invalider les entrées d'une **TLB** lorsqu'une ligne de cache contenant une ou plusieurs entrées de **TLB** est modifiée, soit par le processeur local, soit par une requête de cohérence.
- Lorsqu'une ligne X présente dans le cache L1 contient des données d'une table des pages partiellement recopiées dans une **TLB** alors la ligne est marquée de type **IN-TLB** dans le répertoire du cache L1. En cas de requête de cohérence de type *`Multicast Update`*(X), *`Multicast Invalidate`*(X), ou *`Broadcast Invalidate`*(X) portant sur une ligne de type **IN-TLB**, ou en cas d'écriture par le processeur local, le contrôleur de cache L1 déclenche localement une opération *`Scan-TLB`* ou *`Reset-TLB`* permettant d'invalider toutes les entrées des deux **TLBs** susceptibles d'avoir été modifiées. L'opération *`Scan-TLB`* est une recherche associative sur les entrées appartenant à la ligne de cache X. L'opération *`Reset-TLB`* est une invalidation globale du contenu des deux **TLBs**.

Évidemment, cette méthode suppose une inclusivité stricte du contenu des deux **TLBs** dans le cache L1 local : si une entrée est présente dans une **TLB**, la ligne de cache à laquelle cette entrée appartient doit être présente dans le cache L1 local pour que la requête de cohérence soit transmise aux **TLBs**.

1.5 Points faibles du protocole DHCCP actuel

1.5.1 Trafic d'écriture

Les expérimentations montrent que le protocole **DHCCP** est scalable, c'est-à-dire que le trafic de cohérence augmente linéairement avec le nombre de clusters. (Voir le chapitre 7). En effet, la stratégie d'écriture immédiate simplifie fortement le protocole de cohérence des caches puisque les lignes du cache L2 sont toujours à jour. Ainsi, un cache L2 peut répondre aussitôt aux requêtes de lecture des caches L1. Au contraire, la stratégie d'écriture différée ajoute une transaction supplémentaire du fait que le cache L2 doit rechercher les valeurs à jour de la ligne lorsque celle-ci est dans l'état `EXCLUSIVE`. En outre, la stratégie d'écriture différée suppose que l'écriture dans une ligne non encore allouée commence par le chargement de la ligne dans le cache L1 avant sa modification (*Write Allocate*) pour réduire le trafic lié aux écritures.

Cette simplification du protocole a un coût : la stratégie d'écriture immédiate impose que toutes les écritures effectuées par les cœurs soient envoyées vers le cache L2. Ces écritures génèrent beaucoup de trafic sur le réseau et constituent une source importante de consommation énergétique, car le nombre de transactions d'écritures sur le réseau **CMD** est finalement beaucoup plus grand que le nombre de transactions de lecture. En effet, seules les lectures qui font MISS sur les caches L1 déclenchent des transactions de lecture sur le réseau. Outre la consommation énergétique, ce grand nombre de

commandes d'écriture dans les caches L2 peut provoquer une contention d'accès aux caches L2 et une augmentation de la latence moyenne des transactions.

Une grosse partie de ce trafic est inutile, puisqu'il concerne des données non partagées, telles que les données contenues dans les segments de piles des différentes tâches. Ces données ne posent pas de problème de cohérence, et pourraient très bien utiliser une politique d'écriture différée.

Le premier objectif est donc de faire évoluer le protocole DHCCP pour réduire le trafic des écritures, et donc la consommation énergétique sans remettre en cause la scalabilité, en combinant les stratégies d'écriture différée et d'écriture immédiate.

1.5.2 Inclusivité des TLBs dans les caches L1

Le rapport de thèse de Yang GAO [13] a bien montré la scalabilité du protocole **DHCCP** pour ce qui concerne la cohérence des **TLBs**. Mais nous observons cependant que la plupart des évènements `Scan-TLB` (entre 90% et 95%) ou des évènements `Reset-TLB` (%50) sont dus à des évincements spontanés de lignes de type **IN-TLB** dans le cache L1. Ceci se traduit par des invalidations parasites - totale ou partielles - des **TLBs** non réellement justifiées, mais imposées par la règle d'inclusivité entre le cache L1 et les **TLBs** (si une donnée est présente dans une **TLB**, la ligne de cache correspondante doit être présente dans le cache L1 des données).

Les inconvénients sont une réduction artificielle de l'efficacité des **TLBs** par une augmentation du taux de *Miss* **TLB**, et une augmentation de la consommation énergétique à cause du trafic entre les caches L1 et L2 pour le rechargement des lignes évincées inutilement ;

De plus, l'évincement des entrées **TLB** n'est pas connu par le contrôleur du cache L1. Il est alors possible d'avoir des lignes de type **IN-TLB** qui n'ont plus d'entrées dans la **TLB**, mais qui déclenchent quand même des opérations de `Scan TLB` lors de leur évincement.

Le second objectif est donc de faire évoluer le protocole de cohérence des TLBs pour relâcher la contrainte d'inclusivité des TLBs dans les caches L1, de façon à éviter les invalidations inutiles, pour réduire ici aussi la consommation énergétique.

1.5.3 Consommation du cache instruction

Une réalisation matérielle VLSI d'un prototype comportant 96 cœurs (24 clusters) est en cours de réalisation au CEA-LETI, dans le cadre du projet européen SHARP. Des analyses détaillées de la consommation énergétique de l'architecture TSAR ont été effectuées par le CEA-LETI sur le schéma en portes obtenu à partir des modèles VHDL synthétisables fournis par le Lip6. Ces analyses montrent qu'une part significative de la consommation énergétique est liée aux accès systématiques aux caches L1 contenant les instructions (une fois par cycle pour tous les cœurs actifs de la plateforme). Dans un cluster avec quatre coeurs, les quatre caches L1 instructions consomment un tiers de la consommation totale du cluster.

Le troisième objectif est donc de réduire la consommation énergétique du cache d'instructions en évitant les accès systématiques à chaque cycle, en essayant de

mieux exploiter la localité spatiale des instructions.

1.6 conclusion

Dans les processeurs *manycores* à mémoire partagée, la hiérarchie des caches est indispensable pour réduire les temps d'accès aux données et aux instructions. La cohérence matérielle des caches et des **TLB** permet de simplifier significativement le travail du système d'exploitation et des programmeurs d'application.

Le système mémoire est l'un des consommateurs énergétiques majeurs des systèmes informatiques [14], et dans les processeurs *manycores*, la consommation dynamique est en grande partie liée aux accès mémoire dans les caches L1 et L2, et au déplacement des données dans les réseaux de communication.

Le protocole de cohérence **DHCCP** actuellement implémenté dans l'architecture *manycores* **TSAR** passe à l'échelle, mais cette scalabilité a un coût énergétique important qui peut sans doute être réduit. L'objectif général de ma thèse est donc de réduire significativement la consommation dynamique sans dégrader les performances et la scalabilité.

Plus précisément, ce travail répond à trois questions :

1. Comment réduire le coût énergétique lié aux écritures immédiates systématiques dans le protocole **DHCCP** ?
2. Comment réduire le coût des évincements spontanés de lignes dans le cache L1 qui provoquent une augmentation artificielle des *Miss* **TLB** ?
3. Comment réduire la consommation élevée des caches L1 d'instructions ?

Chapitre 2

État de l'art

2.1 Protocoles à base de *Snoop*

Les protocoles à base de *Snoop* [6] sont toujours populaires dans les architectures **SMP**. Malheureusement, la bande passante du bus restreint le nombre de cœurs. Afin de supporter plus de cœurs, la plupart des architectures à espace d'adressage partagé utilisent des micro-réseaux, et des bancs mémoire physiquement distribués, pour permettre plusieurs transactions simultanées. De manière générale, ces architectures sont donc de type **NUMA**, et ne supportent pas les protocoles de *Snoop*, car les interconnecteurs de type **NOC** ne sérialisent pas les transactions. Pour qu'une architecture **NUMA** puisse implémenter un protocole de *Snoop*, une solution est de modifier le réseau de communication pour permettre de transmettre les requêtes de cohérence dans l'ordre.

SCORPIO [15] est l'exemple d'une telle architecture : il utilise un **NOC**, mais le protocole de cohérence est de type *Snoop*. Le réseau de communication est conçu afin de sérialiser les requêtes de cohérence sur un réseau particulier, appelé (*Notification Network*).

Cette méthode permet d'implémenter effectivement un protocole de type *Snoop* dans une architecture comprenant un **NOC**, mais la sérialisation des requêtes de cohérence limite fortement la scalabilité. Ainsi, le format des messages de notification interdit le passage à l'échelle, parce qu'un vecteur de bits (un bit par nœud) est inclus dans le message.

Nous nous intéressons dans cette thèse aux protocoles de cohérence scalables, pouvant supporter jusque 1024 coeurs. Par conséquent, toutes les solutions existantes que nous présentons dans la suite sont à base de répertoire global, et utilisent une des deux différentes stratégies d'écriture possibles : écriture immédiate ou écriture différée.

2.2 Protocoles à écriture immédiate

Le protocole à écriture immédiate et invalidation (ou **WTI** pour *Write Through Invalidate* [7]) est un des protocoles de cohérence les plus simples. Toutes les écritures sont transmises vers le cache partagé, ce dernier est donc toujours à jour. Il n'y a que deux états pour une ligne de cache : `Valid` et `Invalid`. Lorsque le cache partagé reçoit une requête d'écriture sur une ligne partagée, une requête d'invalidation est

envoyée aux caches privés contenant la copie. Évidemment, l'écriture systématique en mémoire génère énormément de transactions d'écriture qui consomment de l'énergie inutilement. Chtioui, Hajer et Atitallahčitechtioui2008gestion propose un protocole à base de ce type de stratégie qui gère la cohérence avec la façon hybride : détermination des deux méthodes *Update* ou *Invalidate* est réalisée en utilisant un tableau d'historique des opérations réalisées sur une ligne de cache. Mais ce protocole est comme le protocole **DHCCP** génère beaucoup de trafic d'écriture.

Un autre protocole de ce type qui est utilisé par le processeur *manycore* TILEPro64 [16] est développé par la société TILERA. Le processeur contient 64 cœurs, un cœur par nœud. Elle utilise un **NOC** avec une topologie de type *mesh* 2D pour interconnecter les nœuds. Le TILEPro64 utilise le protocole matériel **DDC** (*Dynamic Dsitributed Cache*) afin de gérer la cohérence des caches. Il y a deux niveaux de cache dans chaque nœud, le premier niveau de cache privé (cache L1) est implémenté à côté de chaque cœur, le deuxième niveau de cache (cache L2) est partagé par tous les caches L1 et distribué physiquement dans tous les nœuds.

Le protocole **DDC** utilise la technique de répertoire global. Le répertoire global est distribué dans chaque cache L2. Chaque noeud prend en charge un segment de la mémoire. Pour toutes les adresses de ce segment, c'est à ce cache L2 maître que vont s'adresser les caches L1 pour obtenir une copie d'une des lignes. Dans le protocole **DDC**, une ligne de cache peut non seulement être sauvegardée dans le cache L2 maître, mais aussi être stockée temporairement dans d'autres caches L2 si le cache L1 local a demandé la ligne. La figure 2.1 montre une lecture de la ligne X par le nœud A vers le cache L2 maître dans un nœud B distant.

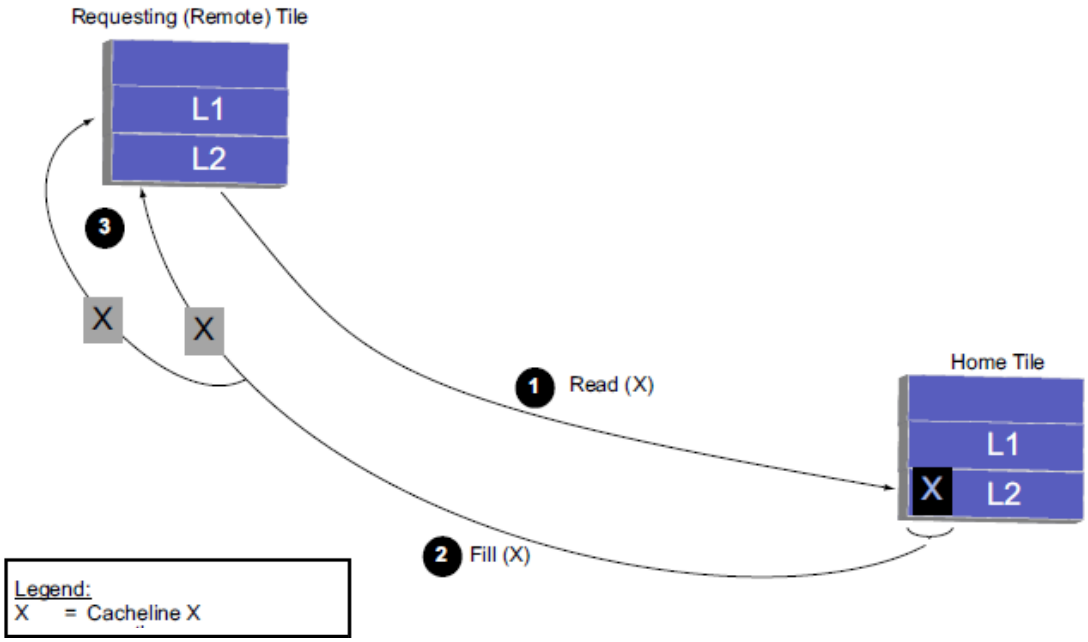


FIGURE 2.1 – Requête de lecture sur la ligne X

1. le nœud A cherche d'abord la ligne X dans son cache L2 local. Si la X n'est pas présente, il envoie une requête de lecture au cache L2 du nœud B, maître de la

ligne X.

2. le nœud B répond à cette requête afin de stocker la ligne X dans le cache L2 local du nœud A.
3. la ligne X donc est copiée dans les caches L1 et L2 du nœud A.

Le protocole **DDC** utilise une stratégie d'écriture immédiate entre le cache L1 et le cache L2. La requête d'écriture est envoyée non seulement au cache L2 maître, mais aussi au cache L2 local. La figure 2.2 montre l'écriture d'un mot dans la ligne X du nœud A au nœud B (le cache L2 maître de la ligne X) :

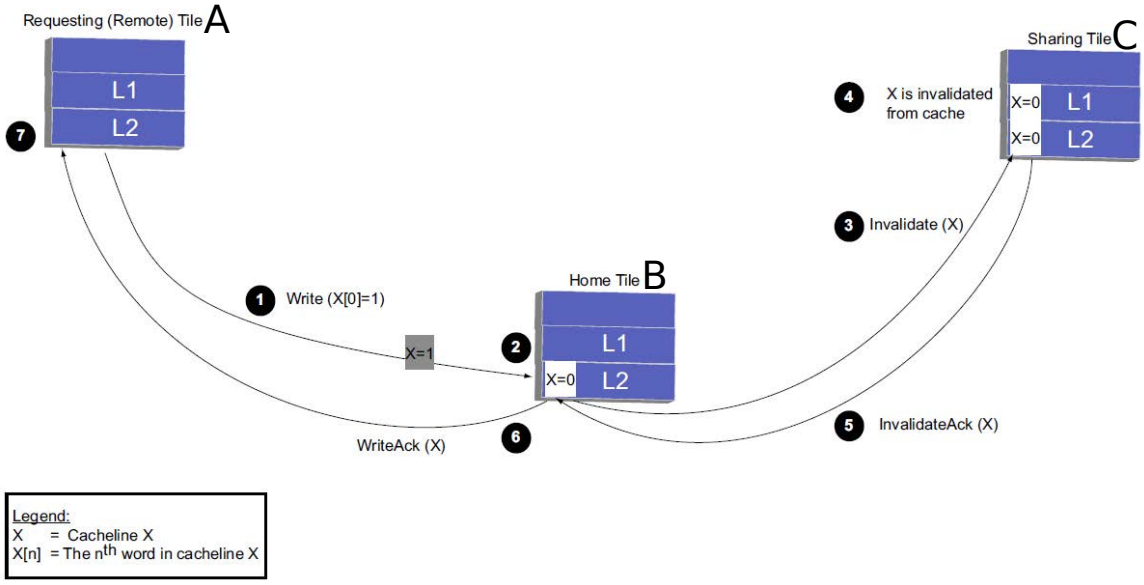


FIGURE 2.2 – Requête d'écriture sur la ligne X

1. Le cache L1 envoie l'écriture au cache L2 local du nœud A et au cache L2 maître du nœud B.
2. Le cache L2 maître met à jour la ligne X et accède le répertoire global afin de chercher les copies de la ligne X (ici le nœud C contient une copie)
3. Le nœud B envoie une requête d'invalidation au nœud C.
4. Le nœud C reçoit la requête d'invalidation et invalide la copie de X dans les caches L1 et L2 locaux.
5. Le nœud C répond à la requête d'invalidation du nœud B.
6. Lorsque le nœud B reçoit la réponse à l'invalidation, il envoie la réponse à l'écriture de X au nœud A.
7. Le nœud A reçoit la réponse, et la transaction d'écriture se termine.

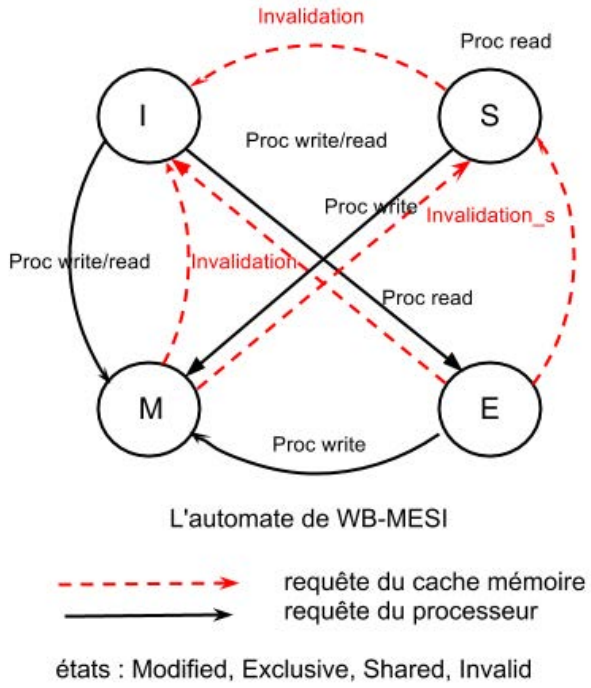
L'architecture TILERA est très voisine de l'architecture TSAR qui nous a servi de référence. Elle contient un seul processeur par cluster (au lieu de 4 dans TSAR). Comme **DHCCP**, le protocole **DDC** implémente une politique d'écriture immédiate. Mais, à la différence de **DHCCP**, il utilise une stratégie d'invalidation systématique, au lieu d'une stratégie hybride d'invalidation et de mise à jour suivant le nombre de copies. Par

ailleurs, le protocole **DDC** autorise la réplication d'une ligne de cache dans plusieurs caches L2. Cette technique réduit la latence des transactions de lecture, et réduit également la consommation énergétique, mais elle crée deux problèmes : (a) la réduction de la capacité globale du cache L2 puisqu'une ligne de cache partagée est répliquée à la fois dans les L1 et dans les L2 ; (b) l'augmentation de la consommation énergétique d'écriture, puisqu'une écriture s'effectue dans les deux caches L2. Surtout, le protocole **DDC** ne résout pas le problème de la consommation énergétique liée à la stratégie d'écriture immédiate.

2.3 Protocoles MESI et MOESI

Les protocoles **MESI** [17] ou **MOESI** [18] sont les protocoles les plus répandus, dans les architectures **SMP** comme dans les architectures **NUMA**, car ils utilisent la stratégie d'écriture différée, qui minimise le trafic sur le bus (ou sur le réseau) entre les coeurs et les bancs de mémoire. En effet, dans les architectures non intégrées sur puce, le bus ***fond de panier*** (ou le réseau de communication) entre les cartes processeurs constitue souvent le goulot d'étranglement de l'architecture, et la réduction du trafic est donc le principal objectif.

Avec la stratégie d'écriture différée, il n'y a plus de requête d'écriture du cache L1 vers le cache L2, mais une requête d'écriture vers une ligne non présente dans un cache L1 entraîne une requête de type *Read For Ownership* (écriture attribuée) vers le cache L2. Une entrée dans un cache L1 peut être dans quatre états :



M `MODIFIED`
La ligne a été modifiée localement dans le cache L1, et ce cache contient la seule copie.

E `EXCLUSIVE`
La ligne possède une seule copie dans un cache L1 avec droit d'écriture, mais n'a pas encore été modifiée par rapport à la copie du L2.

S `SHARED`
La ligne est partagée en lecture par plusieurs caches L1, et non modifiable.

I `INVALID`
La ligne n'est pas valide dans le cache L1.

FIGURE 2.3 – Automate du protocole **MESI**

Le protocole **MESI** supprime donc toutes les transactions d'écriture entre les caches L1 et les caches L2 (autres que les évincements des lignes modifiées dans les caches L1), mais l'existence des états `MODIFIED` et `EXCLUSIVE` augmente fortement la complexité

du protocole de cohérence (figure 2.3) en termes de nombres de messages de cohérence échangés entre les caches L1 et les caches L2.

Dans le protocole **MESI**, lorsqu'un cache L1 doit modifier l'état d'une ligne de cache de `MODIFIED` vers `SHARED` (car un autre cache L1 demande la même ligne), le contenu à jour de la ligne doit être envoyé au cache L2. Afin de supprimer cette requête d'écriture vers le cache L2, une évolution de protocole **MESI** consiste à ajouter un cinquième état (`OWNED`), d'où le nom **MOESI**. Une ligne passe dans l'état (`OWNED` dans un cache quand elle était dans l'état `MODIFIED` et qu'un autre cache demande une copie. Le cache qui possède la ligne dans l'état `OWNED` contient la donnée la plus récente, et doit donc répondre à tous les miss lui-même, car la mémoire n'est plus à jour. Les caches qui font des miss obtiennent eux la ligne dans l'état `SHARED` (voir figure 2.4).

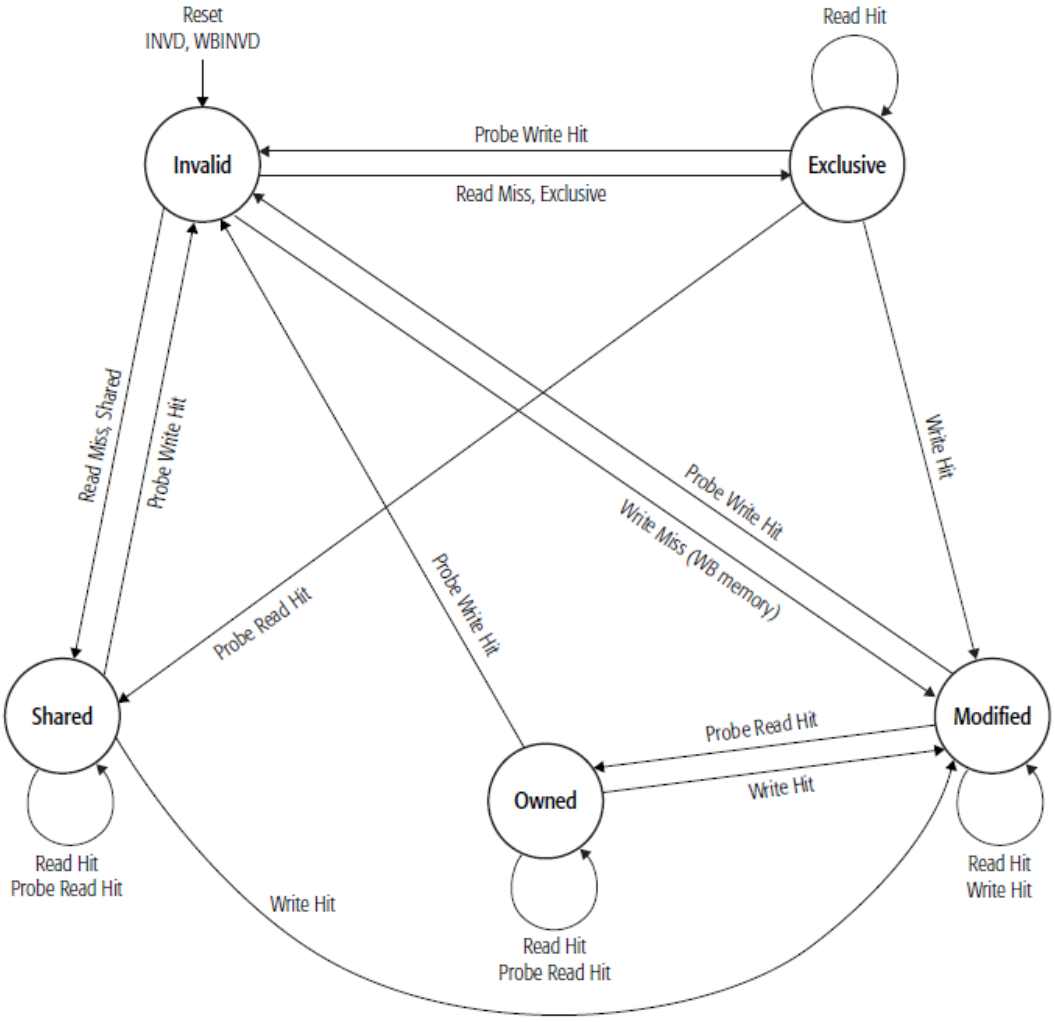


FIGURE 2.4 – Automate du protocole **MOESI**

L'implémentation du protocole **MOESI** est assez simple dans un protocole à base de *Snoop*, puisque chaque coeur surveille toutes les commandes passées sur le bus. Le coeur contenant une ligne en état `MODIFIED` peut transférer cette ligne sur le bus lorsqu'il a détecté qu'un autre coeur demande cette ligne. Mais avec le répertoire global, il faut ajouter des indirections : lorsque le cache L2 reçoit une requête de lecture sur une ligne qui n'est pas partagée (cette ligne est présente exclusivement dans un cache

L1 dans état `EXCLUSIVE` ou `MODIFIED`), le cache L2 doit demander au cache L1 de répondre à cette requête parce qu'il possède la valeur la plus récente. En conséquence, le cache L2 doit commencer par envoyer une requête *Directed Prob* au cache L1 contenant la ligne en état `EXCLUSIVE` ou `OWNED`, puis ce cache L1 envoie la ligne au demandeur, et en même temps répond au cache L2. Enfin ce cache L1 change l'état de la ligne vers `OWNED` si cette ligne était modifiée, ou vers `SHARED` si elle ne l'était pas.

Actuellement, les plus grosses entreprises dans le domaine des semi-conducteurs, comme Intel ou AMD, utilisent les protocoles **MESI** ou **MOESI** dans leurs processeurs multicoeurs.

2.3.1 Protocole MOESI chez AMD

La société AMD utilise le protocole **MOESI** dans le processeur Opteron [19], par exemple dans la version *Magny Cours*, à base de répertoire global. La figure 2.5 montre le diagramme d'un nœud dans cette architecture.

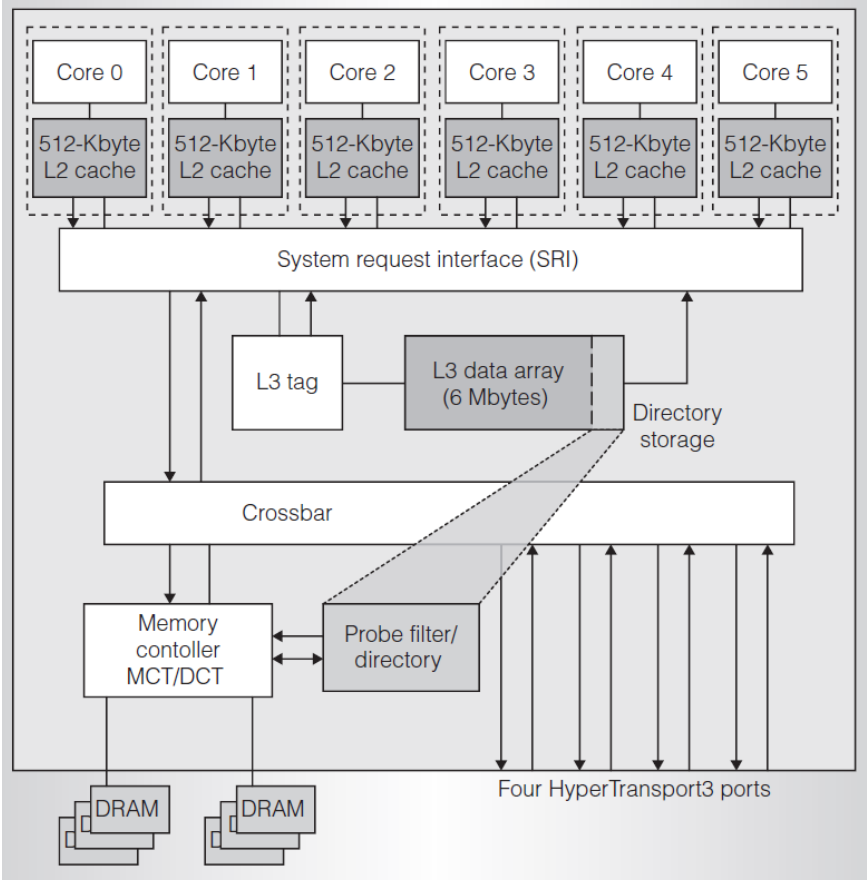


FIGURE 2.5 – Diagramme d'un noeud de l'architecture Opteron *Magny Cours*

Chaque nœud intègre six cœurs x86-64. Les caches L1 et L2 sont privés. Le cache L3 agit comme un cache partagé qui sauvegarde les lignes évincées par tous les caches L2. Dans le processeur *Magny Cours*, le répertoire global (qui s'appelle *Probe Filter*) sauvegarde les informations sur toutes les lignes présentes dans les caches L1, L2 et

L3. Lorsque le contrôleur mémoire reçoit une requête vers la mémoire principale, il accède d'abord au répertoire global afin de récupérer les informations sur cette ligne de cache. En analysant l'état de la ligne de cache, le contrôleur mémoire décide d'envoyer la requête vers la mémoire principale ou de demander la copie au cache L1 qui en est propriétaire. Une entrée du répertoire contient trois champs : (i) le champ *Tag* contient l'adresse physique de la ligne ; (ii) le champ *State* contient l'état de la ligne parmi les cinq états du protocole ; (iii) le champ *Owner* contient l'identifiant du cache L1 propriétaire.

Dans le répertoire global, une ligne peut être dans cinq états :

- `EM` : une copie de la ligne de cache est présente dans un seul nœud, et elle est soit dans l'état `EXCLUSIVE`, soit dans l'état `MODIFIED`.
- `O` : une copie de la ligne de cache est sauvegardée dans un nœud dans l'état `OWNED`. Plusieurs copies existent possiblement dans d'autres nœuds dans l'état `SHARED`. La mémoire principale n'est pas à jour.
- `S` : la ligne de cache (non modifiée) est présente dans plusieurs nœuds en mode `SHARED`.
- `S1` : un seul nœud a une copie de la ligne dans l'état `SHARED`, et l'identifiant de nœud est stocké dans le champ *Owner*.
- `I` : la ligne de cache n'est pas présente.

La figure 2.6 montre les différents scénarios possibles et les transactions associées lorsque le contrôleur de mémoire principale reçoit une requête de lecture instruction (*`Fetch`*), de lecture de donnée (*`Load`*) ou pour modifier localement la ligne (*`Store`*).

Les stratégies associées à ces requêtes sont les suivantes :

- *`Fetch`* : la ligne est mise dans l'état `SHARED`.
- *`Load`* : Par défaut, la ligne est mise dans l'état `EXCLUSIVE`.
- *`Store`* : la ligne est mise dans l'état `MODIFIED`.

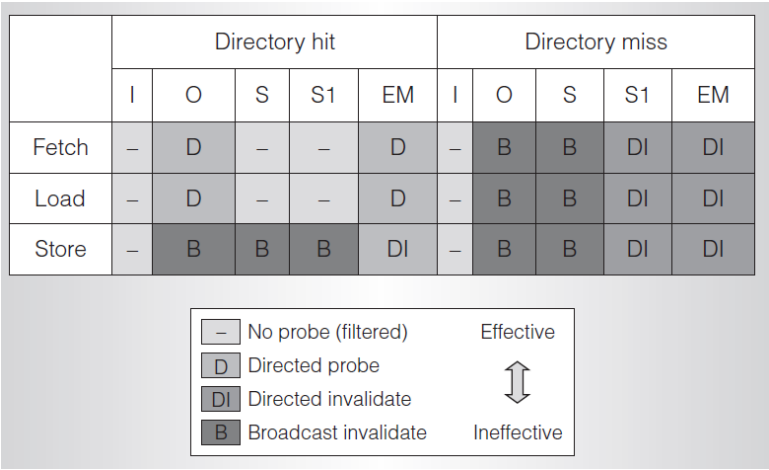


FIGURE 2.6 – Actions du répertoire *Probe filter* suite à la réception d'une requête

Dans chaque scénario, la colonne *Directory Hit* indique que la ligne demandée par la requête est stockée dans le répertoire global. Les colonnes EM, O, S, S1 et I indiquent l'état de cette ligne actuelle, et le contenu de chaque case représente les différentes

transactions de cohérence générées par rapport à l'état de la ligne de cache. La colonne *Directory Miss* indique le cas où la ligne demandée n'est pas encore présentée dans le répertoire global. En conséquence, le contrôleur doit envoyer une transaction de lecture vers la mémoire principale. Les entrées dans la colonne *Directory Miss* montrent les différentes transactions de cohérence sur la ligne de cache qui est remplacée par cette nouvelle ligne.

Lorsque la ligne demandée n'est pas présente dans le répertoire global, la requête est transférée directement à la mémoire principale sans générer de transaction de cohérence supplémentaire. Si le répertoire doit faire une place pour enregistrer cette nouvelle ligne, le contrôleur envoie un `Broadcast Invalidate` si la ligne évincée a plusieurs copies (états `O` et `S`), et un `Direct Invalidate` si elle y a une seule copie (états `S1` et `EM`).

Si la ligne est demandée en lecture, et qu'elle se trouve dans le répertoire dans l'état `EM` ou `O`, le contrôleur envoie une transaction *Directed Probe* au nœud *Owner* afin que ce dernier renvoie la ligne complète au nœud qui la demande.

Si la ligne est demandée en écriture, et qu'elle se trouve dans le répertoire global, le contrôleur de cache doit invalider toutes les copies dans les caches privés.

Si une ligne de cache est partagée en écriture par plusieurs coeurs, à chaque écriture, le contrôleur mémoire doit chercher la valeur à jour dans le cache privé qui en est propriétaire, invalider les éventuelles autres copies, et la transmettre au demandeur afin de lui permettre de la modifier localement.

Les scénarios décrits ci-dessus montrent que les trois états `MODIFIED`, `OWNED` et `EXCLUSIVE`, qui sont la conséquence directe de la politique d'écriture différée, complexifient fortement le protocole. Surtout, lorsque le nombre de processeurs augmente, le trafic lié à la cohérence peut augmenter quadratiquement avec le nombre de processeur, ce qui limite évidemment la scalabilité.

2.3.2 Protocole GOLS pour le coprocesseur Intel Xeon Phi

Le Xeon Phi [20] référence une série de coprocesseurs *Manycore* basée sur l'architecture *Many Integrated Core* d'Intel. Par exemple, le processeur commercial 5110P contient 60 cœurs qui peuvent atteindre 1056 MHz. Chaque cœur contient un cache L1 instruction et un cache L1 donnée (32 Ko), ainsi qu'un cache L2 privé (512 Ko). La cohérence des caches est garantie par un système appelé *Distributed Tag Directory* (**DTD**), le *Tag Directory* est un répertoire global qui contient les informations sur les lignes de cache stockées dans les caches privés. Le nombre de *Tag Directories* est égal au nombre de cœurs. Toutes les lignes de cache sont distribuées dans les *Tag Directories*. Dans le coprocesseur 5110P, tous les cœurs sont connectés sur un bus avec une topologie en double anneau (figure 2.7).

Le coprocesseur Xeon Phi utilise un protocole de cohérence appelé **GOLS** (*Globally Owned Locally Shared*) basé sur le protocole **MESI**, qui utilise le répertoire global **DTD** comme répertoire distribué. L'objectif du protocole **GOLS** est d'éviter d'écrire la ligne à jour dans la mémoire principale lorsqu'un cœur veut lire une ligne de cache dans l'état Modifié, de manière similaire à l'optimisation introduite dans le protocole **MOESI**.

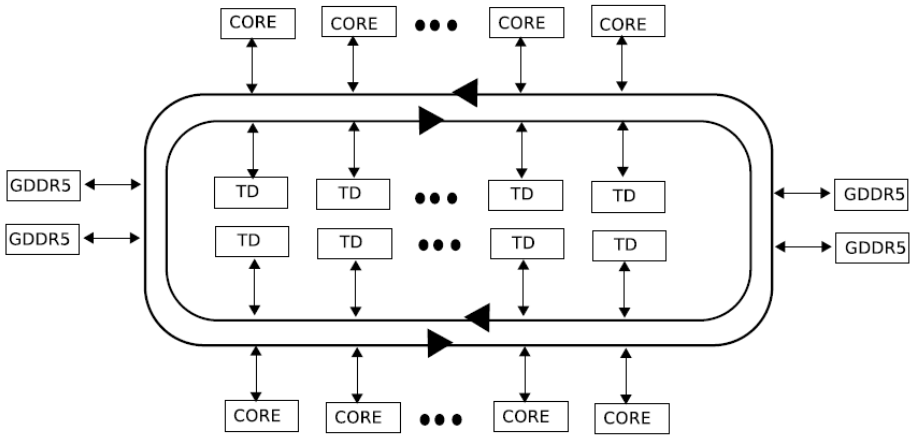


FIGURE 2.7 – Architecture du coprocesseur Xeon Phi d'Intel

Dans le coprocesseur Xeon Phi, chaque ligne de cache L1 maintient un état du protocole **MESI**, mais la signification de l'état `SHARED` est modifiée : dans cet état, la ligne peut être stockée dans plusieurs cœurs, mais elle peut être modifiée. Afin de distinguer les différentes situations sur les lignes de cache dans cet état `SHARED`, des informations plus précises sont enregistrées dans le *Tag Directory*, et les états possibles de la ligne dans le répertoire sont les suivants :

1. `GOLS` (*Globally Owned Locally Shared*) : la ligne de cache peut être stockée dans un ou plusieurs coeurs, et la ligne a été déjà modifiée par rapport à la mémoire externe.
2. `GEGM` (*Globally Exclusive / Modified*) : Un seul cœur contient cette copie, la ligne peut-être dans l'état `EXCLUSIVE` ou `MODIFIED`.
3. `GS` (*Globally Shared*) : la copie de la ligne de cache peut être stockée dans un ou plusieurs coeurs, mais la ligne est forcément non modifiée par rapport à la mémoire externe.
4. `GI` (*Globalement Invalide*) : aucun cœur ne contient la copie.

Lorsqu'un cœur fait un miss de cache sur une ligne X, il envoie la requête au **DTD** correspondant. Le **DTD** répond à la requête selon l'état de la ligne X. Si un autre cœur *Owner* contient une copie de X, le **DTD** demande au cœur *Owner* d'envoyer la ligne complète en réponse au miss. Le **DTD** met à jour l'état de la ligne X quand il a reçu la réponse du cœur *Owner*. Sinon, le **DTD** envoie une requête au contrôleur de mémoire pour lire la ligne.

Comme il apparaît clairement dans la description ci-dessus, les deux états `GOLS` et `GEGM`, qui sont directement liés à la stratégie d'écriture différée, complexifient sensiblement le protocole, puisque dans l'état `GEGM`, le contrôleur mémoire ne peut répondre directement aux requêtes de lecture des processeurs.

2.3.3 Protocole ACKwise du MIT

L'architecture **ATAC** [21] développée par le MIT est un processeur *manycore* clusterisé qui contient 1024 cœurs distribués dans 64 clusters (16 cœurs par cluster). La

cohérence des caches est garantie par le protocole matériel à base de répertoire global **ACKwise**. Le répertoire global est distribué de manière uniforme par cœur. En outre, chaque cœur est défini statiquement comme responsable d'un segment de la mémoire.

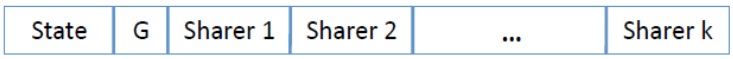


FIGURE 2.8 – Structure d'une entrée dans le répertoire global

Le protocole **ACKwise** est de type **MOESI**. Comme montré dans la figure 2.8, chaque entrée du répertoire global contient trois types d'information pour maintenir la cohérence des caches :

1. Un état du protocole **MOESI** pour la ligne de cache associée à l'entrée de répertoire.
2. La liste des copies pour la ligne de cache. Le protocole **ACKwise** garde au maximum k identifiants pour une ligne de cache partagé.
3. Lorsque le nombre de copies pour une ligne de cache partagée est dépassé, c'est-à-dire qu'il y a plus de k copies, la liste des copies est abandonnée, le bit **G** (*global*) est activé, et seul le nombre de copies est sauvegardé.

Au cas où le contrôleur de répertoire reçoit une requête pour une ligne partagée, il traite la requête suivant l'état de la ligne :

- si l'état est `INVALID`, le contrôleur du répertoire transfère la requête au contrôleur de la mémoire principale. Ce dernier envoie directement la copie au demandeur et il envoie aussi une réponse au contrôleur de répertoire afin de mettre à jour l'état à `EXCLUSIVE`, puis ajoute l'identifiant du demandeur à la liste des copies.
- si l'état est un état `VALID` (parmi `M, O, E, S`), alors le contrôleur de répertoire transfère la requête à un des caches ayant une copie. Ce cache expédie alors la copie directement au demandeur et envoie une réponse au contrôleur de répertoire pour le notifier, et il modifie le champ état par rapport au protocole **MOESI**. Si la liste des copies n'est pas pleine, le répertoire ajoute l'identifiant du demandeur dans la liste des copies. Sinon le bit **G** est activé, et le nombre de copies total (k + *1*) est sauvegardé dans Sharer_k.

Lorsque le contrôleur de répertoire reçoit une requête pour une copie exclusive, il vérifie d'abord le champ d'état de la ligne de cache :

- si l'état est `INVALID`, le contrôleur de répertoire poursuit la même action que lors d'une demande d'une ligne partagée. À la fin de la requête, le champ état est changé en `MODIFIED`.
- si l'état est en un état `VALID`, alors deux types de transactions sont envoyés par le contrôleur de répertoire afin d'invalider les copies :
  - si le bit **G** n'est pas activé, le contrôleur de répertoire envoie une requête de *`Multicast Invalidate`* au cache contenant une copie ;

- si le bit Global est activé, alors le contrôleur de répertoire envoie une requête de `Broadcast Invalidate` à tous les cœurs. Le nombre de réponses est compté par rapport au nombre de copies sauvegardé dans le champ Sharer_k.

L'architecture **ATAC** a beaucoup de points communs avec l'architecture **TSAR**, mais le protocole de cohérence possède les mêmes limitations en termes de passage à l'échelle que les autres implémentations des protocoles **MESI** ou **MOESI** citées ci-dessus. À notre connaissance, aucune publication n'a démontré la scalabilité de cette architecture pour exécuter des applications multi-threads sur une architecture à 1024 coeurs.

2.4 Conclusion

Les protocoles **MESI** ou **MOESI** utilisent l'écriture différée afin de réduire le nombre de transactions d'écriture entre les caches L1 et les caches L2. Cependant, la gestion de 4 ou 5 états pour une ligne de cache complexifie fortement le protocole, et génère beaucoup plus de trafic de cohérence que les protocoles à écriture immédiate. C'est la forte augmentation de ce trafic de cohérence quand le nombre de processeurs augmente qui limite le passage à l'échelle de ces protocoles au-delà de quelques dizaines de coeurs. Nous pensons donc que les protocoles **MESI** ou **MOESI** présentent des limitations intrinsèques en termes de scalabilité et doivent être remis en cause pour les processeurs *manycore*.

Comme **DHCCP**, le protocole **DDC** de TILERA utilise une stratégie d'écriture immédiate entre les caches L1 et les caches L2, pour simplifier le protocole de cohérence. Mais pas plus que **DHCCP**, il ne résout pas le problème de la consommation parasite liée aux écritures systématiques en mémoire.

Nous chercherons donc à définir une stratégie hybride où l'on utilise l'écriture immédiate pour les données partagées (qui ont besoin de cohérence, mais sont peu nombreuses), et où l'on utilise l'écriture différée pour les données privées (qui sont beaucoup plus nombreuses, mais n'ont pas besoin de cohérence).

Chapitre 3

Protocole *Released Write Through*

Pour réduire le trafic lié aux requêtes d'écriture, nous souhaitons faire évoluer le protocole **DHCCP** en combinant les deux stratégies, écriture immédiate et écriture différée, et en exploitant le fait que la grande majorité des lignes de cache manipulées par les applications logicielles ne sont pas partagées par plusieurs tâches. Par exemple, les piles d'exécution des tâches sont des données privées qui n'ont pas besoin de cohérence. L'idée générale, initialement proposée par Franck Wajsbürt et Ghassan Almaless, est de n'utiliser la stratégie écriture immédiate que pour les données partagées, et d'utiliser la stratégie écriture différée pour les données non partagées. Nous appelons ce protocole **RWT** (*Released Write Through*).

Ce protocole ayant été conçu comme une évolution du protocole **DHCCP**, nous décrivons le protocole en insistant sur les modifications à apporter dans le protocole **DHCCP** existant.

3.1 Principe du protocole RWT

L'utilisation de la stratégie écriture différée par le cache L1 a pour conséquence que la donnée la plus à jour pour certaines lignes de cache est celle qui est présente dans le cache L1. Par conséquent, une ligne de cache valide dans un cache L1 peut être : soit propriété exclusive d'un seul cache L1, éventuellement modifiée ; soit partagée par plusieurs caches L1, mais non modifiée. Dans le protocole **RWT**, il y a donc trois états pour une ligne de cache contenue dans un cache L2 :

- `NC` (***Non Cohérent***) : La ligne de cache est présente dans un seul cache L1. Il n'y a pas de problème de cohérence. Le cache L1 contenant cette ligne peut donc utiliser la stratégie écriture différée afin de réduire le trafic sur le bus.
- `C` (***Cohérent***) : la ligne de cache peut être répliquée dans plusieurs caches L1. Tout cache L1 contenant une ligne de type `C` doit utiliser la stratégie d'écriture immédiate vers le cache L2.
- `INVALID` la ligne n'est répliquée dans aucun cache.

Dans le protocole **RWT**, l'état d'une ligne est défini par le cache L2, et transmis au cache L1 dans la réponse à une requête de lecture. Lorsque le cache L2 reçoit une requête de lecture venant d'un cache L1, il envoie non seulement la ligne complète au

cache L1, mais aussi le code d'état de la ligne (`C` ou `NC`). Dans **TSAR** cette information circule dans le champ **RPKTID** du paquet réponse VCI).

Cependant, en cas de *Miss* sur une requête d'écriture, le contrôleur du cache L1 ne connaît pas l'état (`C / NC`) de la ligne manquante. Il peut donc en principe utiliser une des deux stratégies suivantes :

- le cache L1 envoie directement l'écriture au cache L2 en utilisant la stratégie d'écriture immédiate.
- le cache L1 demande au cache L2 une copie de la ligne manquante. Si la ligne est en état `NC`, l'écriture n'est effectuée que dans le cache L1 (écriture différée). Si la ligne est `C`, l'écriture est immédiatement envoyée au cache L2 et la copie locale est mise à jour.

Pour minimiser le nombre de cycles de gel du processeur causé par les écritures, le protocole **RWT** implémente la première méthode, et utilise un tampon d'écritures postées supportant jusque 8 rafales d'écriture pipelinées,

3.2 Conséquences pour le cache L1

Nous analysons dans cette section les conséquences sur le cache L1 de l'introduction de l'état `NC` pour une ligne de cache.

3.2.1 Évincement d'une ligne de cache

Si le cache L1 reçoit du processeur une requête d'écriture sur une ligne `NC`, cette écriture modifie seulement la copie locale, et le cache L2 n'est pas informé de cette modification. Le cache L1 doit donc envoyer les valeurs modifiées au cache L2 au cas où cette ligne est évincée du cache L1, ce qui nécessite l'introduction et la gestion d'un bit *Dirty* dans le répertoire du cache L1.

Par ailleurs, quand une ligne de cache X de type `NC` est évincée du cache L1 qui en a la propriété exclusive, deux informations doivent maintenant être envoyées au cache L2 : (i) il faut informer le cache L2 que la ligne X est évincée ; (ii) il faut envoyer les valeurs à jour si le bit *Dirty* est activé. Il y a deux possibilités pour transmettre ces informations l'architecture TSAR :

- le cache L1 peut envoyer au cache L2 une requête d'écriture (dans le canal **CMD**), et envoyer en parallèle une requête `Cleanup` (dans le canal **P2M**).
- le cache L1 peut transmettre les valeurs à jour dans la requête `Cleanup` en utilisant uniquement le canal **P2M**.

C'est la seconde possibilité qui a été retenue, car l'utilisation de deux requêtes induit des complications de protocole dues au fait que l'on ne contrôle pas l'ordre d'arrivée des différentes requêtes, et que l'utilisation du réseau direct dans ce cas entraîne un risque d'interblocage. Nous avons donc fait le choix de modifier la requête `Cleanup` sur le réseau de cohérence afin qu'elle puisse contenir des données, en plus de signaler l'évincement de la ligne. Cette nouvelle requête s'appelle `Cleanup-data`.

Cette méthode est utilisée pour tous les types d'évincements sur les lignes NC dans le cache L1, que ce soit un évincement initié par le cache L1 pour faire de la place en cas de *Miss* ou un évincement initié par le cache L2 pour invalider la copie exclusive dans le cache L1.

Dans le protocole **RWT** comme dans le protocole **DHCCP**, lorsqu'une ligne de cache (C ou NC) est évincée du cache L1, cette ligne passe temporairement dans l'état ZOMBIE. Quand le cache L1 reçoit un acquittement `Clack`, la case retourne dans l'état INVALID.

3.2.2 États d'une case dans le cache L1

La figure 3.1 représente le graphe des transitions pour une case du cache L1 :

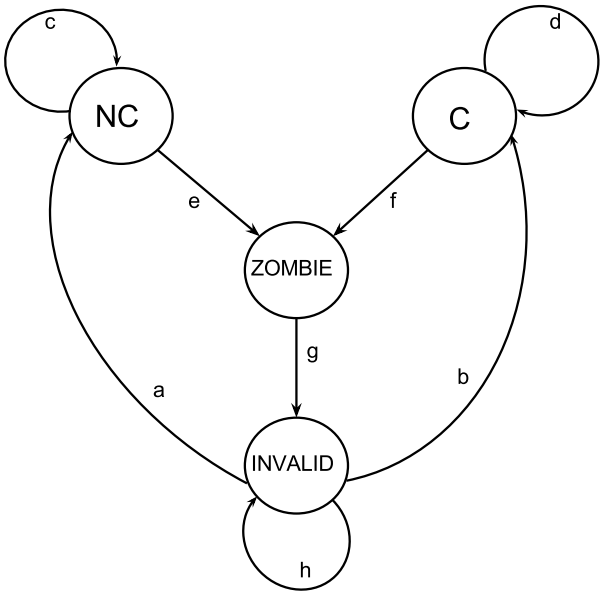


FIGURE 3.1 – Transitions entre états pour une case du cache L1

- Les deux transitions *a* et *b* dépendent du contenu de la réponse du cache L2 à une requête de lecture émise par le cache L1 (état C ou NC).
- les transitions *c*, *d* et *h* correspondent à des requêtes d'écriture : une requête d'écriture du processeur ne change pas l'état de ligne de cache, même si la ligne n'est pas présente dans le cache L1.
- les transitions *e* et *f* correspondent à une invalidation, qui peut être soit initiée par la requête d'invalidation provenant du cache L2, soit initiée par le cache L1 lui-même.
- La transition *g* correspond à la réception de l'acquittement `Cleanup` CLACK, envoyé par le cache L2.

3.3 Conséquences pour le cache L2

Nous analysons dans cette section les conséquences sur le cache L2 de l'introduction de l'état NC (non partagé) pour une ligne de cache.

Dans l'architecture **TSAR**, l'espace d'adressage physique est strictement partitionné entre les clusters. Par conséquent, une ligne de cache est gérée par un unique cache L2, qui a connaissance du nombre de copies dans les caches L1, et peut donc déterminer si la ligne est partagée ou non. L'état d'une ligne de cache peut donc être géré dynamiquement, suivant le nombre de copies, par le cache L2 gestionnaire de cette ligne, et il doit informer le (ou les) cache(s) L1 concerné(s) en cas de changement de l'état de la ligne. Lorsqu'une ligne `NC` du cache L2 est accédée par un deuxième cache L1, le cache L2 envoie une requête d'invalidation au cache L1 qui contient la ligne plutôt que d'envoyer un nouveau type de transaction permettant de changer directement l'état (de `NC` vers `C`) dans le cache L1.

Dans le protocole **RWT**, l'état `NC` n'est utilisé que pour les lignes de cache copiées dans les caches de données. Les lignes copiées dans les caches d'instructions sont toujours copiées dans les caches L1 avec l'état `C`, quel que soit le nombre de copies. En effet, les écritures dans les lignes de cache instruction sont des événements extrêmement rares, et on souhaite éviter le coût associé aux changements d'état de `NC` à `C`, lorsque le même code est copié dans plusieurs caches L1.

3.3.1 Évolution de l'état d'une ligne de cache

Si une ligne de données est lue ou écrite par un seul cache L1, la ligne reste dans l'état `NC` jusqu'à la fin de la vie de cette ligne dans le cache L2. En revanche, si la ligne est partagée par plusieurs caches L1, son état évolue pendant le cycle de vie de cette ligne, comme décrit par le chronogramme (3.2) :

(1) Le processeur du cache L1(C0) veut lire une donnée contenue dans la ligne X, mais celle-ci n'est pas présente dans L1(C0). L1(C0) envoie donc une requête de lecture *Read*(X) au cache L2 gestionnaire de X. Puisque c'est le premier accès sur la ligne X, le cache L2 accède à la mémoire principale (ou au cache L3) pour obtenir une copie de la ligne X. Puis le cache L2 envoie à L1(C0) une copie de la ligne X dans l'état `NC` en utilisant une réponse *Rsp*(X,NC).

(2) Lorsque le processeur du cache L1(C0) veut écrire dans la ligne X, l'écriture est effectuée seulement dans L1(C0) puisque la ligne X est dans l'état `NC`, et le bit *Dirty* de la ligne X est activé dans le répertoire de L1(C0).

(3) Si le processeur d'un autre cache L1(C1) veut lire une donnée contenue dans la ligne X, celle-ci n'est pas présente dans L1(C1). L1(C1) doit donc demander la ligne X au cache L2 avec une requête de lecture *Read*(X). Puisque le cache L2 sait que la ligne X est en état `NC`, et qu'elle est copiée dans L1(C0), le cache L2 envoie une requête *Invalidate*(X) à L1(C0), afin de récupérer les valeurs à jour sur la ligne X.

(4) Lorsque le L1(C0) reçoit la requête d'invalidation pour la ligne X, il envoie la requête *Cleanup-data* au cache L2, puisque le bit *Dirty* est activé. La ligne X passe dans l'état `ZOMBIE` dans le cache L1(C0).

(5) Lorsque le cache L2 reçoit la requête *Cleanup-data* sur la ligne X, il met à jour les données contenues dans la ligne X et change l'état de `NC` à `C`. Puis le cache L2 envoie à L1(C1) une copie de la ligne X dans l'état `C` avec une réponse *Rsp*(X,C), et en parallèle envoie à L1(C0) un acquittement *Clack* sur la ligne X.

⑥ Puisque la ligne X est désormais en état C. Toutes les écritures dans la ligne X par le processeur de L1(C1) sont transmises immédiatement au cache L2 par une commande *Write*(X).

⑦ La ligne X n'étant plus présente dans L1(C0), si le processeur veut relire une donnée contenue dans X, L1(C0) doit redemander la ligne X au cache L2. Puisque la ligne X est en état C, le cache L2 envoie la ligne X dans l'état C à L1(C0).

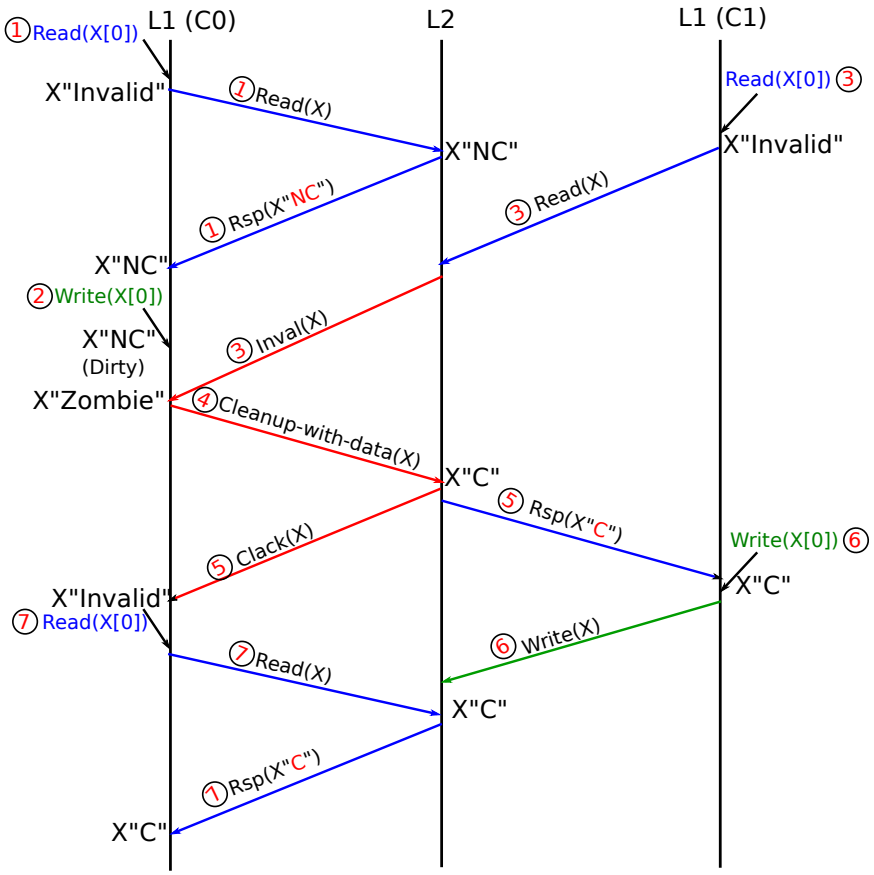


FIGURE 3.2 – Principe du protocole **RWT**

Il est important de noter qu'une ligne peut passer de l'état NC vers l'état C, (comme dans le chronogramme 3.2), mais une ligne de cache qui est passée dans l'état C restera dans l'état C jusqu'à ce qu'elle soit évincée du cache L2, même si la ligne n'a plus aucune copie dans aucun cache L1. Cette méthode d'abord simplifie le protocole, ensuite on considère en effet qu'une ligne de cache qui a été partagée dans le passé a de fortes chances de rester partagée dans le futur.

3.3.2 États d'une case dans le cache L2

La figure 3.3 présente les transitions d'états pour une case du cache L2 :

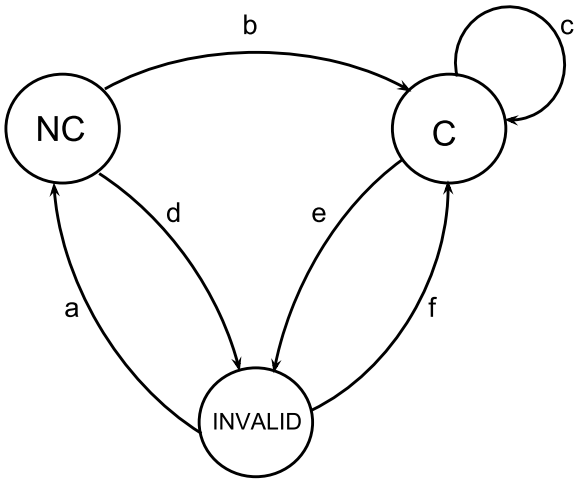


FIGURE 3.3 – Transitions entre états pour une case du cache L2

- les transitions *a* et *f* sont déclenchées par une requête (lecture ou en écriture) provenant d'un cache L1 pour une ligne non contenue dans le cache L2 (*a* : la requête provient d'un cache de données / *f* : la requête provient d'un cache d'instructions).
- la transition *b* correspond à une requête de lecture ou d'écriture sur une ligne `NC` par un autre cache L1 que le propriétaire actuellement enregistré dans le L2.
- la transition *c* correspond à n'importe quelle requête de lecture ou d'écriture provenant d'un cache L1 : une ligne dans l'état `C` ne revient jamais dans l'état `NC` tant qu'elle n'est pas évincée du cache L2.
- les transitions *d* et *e* représentent l'évincement de la ligne du cache L2.

Lorsque le cache L2 reçoit une nouvelle requête de lecture sur une ligne X dans l'état `NC`, il doit enregistrer cette ligne dans la ***table IVT*** afin d'invalider cette ligne dans le cache L1 pour réaliser le changement l'état de `NC` vers `C`. En effet l'automate **`READ`** du cache L2 ne peut pas répondre à cette requête de lecture tant que le cache L2 n'a pas reçu confirmation de l'invalidation. Pour éviter que l'automate **`READ`** reste bloqué en attente de cette confirmation, il stocke toutes les informations sur cette requête de lecture dans la ***table IVT***. L'automate **`READ`** peut ainsi traiter la requête suivante, et c'est l'automate **`CLEANUP`** qui va répondre à la requête de lecture sur la ligne X lorsqu'il reçoit la requête `Cleanup` ou `Cleanup-data` sur la ligne X.

3.3.3 Mécanisme d'évincement dans le cache L2

Lorsqu'une ligne de cache est évincée du cache L2 (pour faire de la place), et si cette ligne a été modifiée (le bit *Dirty* est activé dans le répertoire du cache L2), la ligne de cache doit être re-écrite dans la mémoire principale (ou dans le cache L3).

Dans le protocole **DHCCP**, le cache L2 doit envoyer des commandes d'invalidation à tous les caches L1 possédant une copie (pour garantir la propriété d'inclusivité des caches L1 dans le cache L2), et il peut en parallèle envoyer la requête d'écriture vers la mémoire principale, puisque le cache L2 contient toujours les données les plus à jour. Le

contrôleur de cache L2 enregistre donc sans attendre cette commande d'écriture dans la ***table TRT***, qui est utilisée par les automates gérant les accès à la mémoire principale pour paralléliser différentes transactions vers celle-ci.

Dans le protocole **RWT**, la situation est différente : Si la ligne évincée est dans un l'état `NC`, le contrôleur de cache L2 ne sait pas si la ligne `NC` évincée est à jour ou pas. Il doit donc commencer par envoyer une requête d'invalidation au cache L1 possédant l'unique copie, puis attendre la réponse `Cleanup-data` provenant du cache L1 avant de déclencher la transaction d'écriture vers la mémoire principale.

3.4 Surcoût matériel du protocole RWT

Le protocole **RWT** a été implémenté dans les modèles **cycle-accurate** des contrôleurs de cache L1 et L2 du prototype virtuel SystemC de l'architecture **TSAR**. Cette implémentation a permis d'évaluer de façon précise le surcoût matériel du protocole **RWT** par rapport au protocole **DHCCP**.

Ce surcoût est en principe faible, puisque le protocole **RWT** réutilise toutes les stratégies du protocole **DHCCP** afin de traiter la cohérence des lignes de cache dans l'état `C` (partagées par plusieurs coeurs). En particulier, le protocole **RWT** réutilise sans modifications l'ensemble des structures de données permettant de représenter dans le cache L2 les informations décrivant l'état d'une ligne de cache présente dans le cache L2 : nombre de copies dans les caches L1, et localisation des copies au moyen d'un ***tas*** complètement géré par le matériel.

- Le surcoût dans le contrôleur de cache L1 est le suivant :
  - il faut ajouter un bit *Dirty* dans chaque entrée du répertoire du cache L1.
  - il faut une **FIFO** de 16 mots pour stocker la ligne de cache envoyée sur le canal **P2M** dans les requêtes `Cleanup-data`.
  - il faut 3 états supplémentaires dans les automates du contrôleur de cache L1.
- Le surcoût dans le contrôleur du cache L2 est le suivant :
  - il faut ajouter un bit `NC` dans chaque entrée du répertoire du cache L2.
  - il faut ajouter 15 états dans les différents automates du cache L2.

3.5 Conclusion

Afin de réduire le trafic lié aux écritures, nous proposons que le matériel adapte dynamiquement la politique d'écriture (écriture immédiate / écriture différée) pour chaque ligne de cache X, en fonction du nombre de copies dans les caches L1. Comme cette information n'est connue que par le cache L2 gestionnaire de la ligne X, c'est le contrôleur du cache L2 qui est décisionnaire, mais ce sont les contrôleurs des caches L1 qui appliquent les décisions prises par le L2 :

Si la ligne est non cohérente (elle est propriété exclusive d'un seul cœur), le cache L1 utilise la stratégie écriture différée pour supprimer la transaction d'écriture vers le cache L2. C'est le cas pour toutes les données privées à une tâche particulière (telles que les données stockées dans la pile d'exécution de la tâche).

Si la ligne est cohérente (elle est partagée, en lecture ou en écriture, par plusieurs cœurs), le cache L1 effectue immédiatement toutes les écritures vers le cache L2, qui possède donc toujours la version la plus à jour des données partagées.

Nous avons analysé en détail les conséquences de cette proposition sur les différents automates implantés dans les deux contrôleurs de cache (cache L1 et cache L2) en précisant les graphes de transition d'état pour une ligne de cache dans chacun des deux caches. Le coût matériel est très limité puisqu'il n'implique qu'un seul bit supplémentaire (bit *Dirty*) pour chaque entrée du répertoire du cache L1, et un seul bit supplémentaire (bit `NC`) pour chaque entrée du répertoire du cache L2.

Pour les lignes ayant besoin de cohérence (car partagées par plusieurs cœurs), le protocole **RWT** réutilise tous les mécanismes de maintien de la cohérence définis par le protocole **DHCCP** qui ont démontré leur capacité de passage à l'échelle jusque 1024 cœurs. Soulignons enfin que le changement dynamique d'état d'une ligne de cache (de `NC` vers `C`) n'est effectué qu'une seule fois pendant le cycle de vie de la ligne dans le cache L2.

Les résultats expérimentaux, à la fois en termes de trafic sur les différents réseaux d'interconnexion, et en termes de performances et de scalabilité sont présentés dans le chapitre 7.

Chapitre 4

Protocole HMESI

Dans ce chapitre, nous décrivons le protocole de cohérence **HMESI** que nous avons implémenté dans les contrôleurs de cache L1 et L2 de l'architecture **TSAR**, afin de permettre une comparaison quantitative des performances, de la scalabilité et de la consommation énergétique entre les différents protocoles de cohérence.

Actuellement, la plupart des protocoles de cohérence des caches dans les architectures multi-cores industrielles utilisent des protocoles de type **MESI** ou **MOESI**, reposant sur une stratégie d'écriture différée, bien que cette stratégie nous semble non appropriée pour les architectures *manycores*. Les études existantes comparent les performances des protocoles de cohérence des caches pour des architectures comportant quelques dizaines cœurs. Mais nous n'avons trouvé aucune étude analysant les performances et la scalabilité pour des architectures comportant 1024 cœurs.

Pour permettre une comparaison non biaisée, il fallait donc implémenter tous les protocoles dans la même architecture. Ce travail a été réalisé en coopération étroite avec Quentin Meunier.

4.1 Principe du protocole HMESI

Le protocole **HMESI** que nous avons implémenté dans **TSAR** respecte le protocole **MESI** standard pour les 4 états d'une ligne de cache dans le cache L1. Chaque ligne peut donc avoir les 4 états du protocole **MESI** : `MODIFIED`, `EXCLUSIVE`, `SHARED` et `INVALID` (plus un état transitoire `ZOMBIE`, ajouté afin de garantir la cohérence dans certaines situations). Pour invalider une ligne de cache dans le cache L1, le contrôleur de cache L2 utilise comme pour **DHCCP** une stratégie Hybride *`Broadcast Invalidate`*& *`Multicast Invalidate`* en fonction du nombre de copies dans les caches L1, d'où le nom **HMESI**.

4.1.1 Requête *`GetM`*

Pour répondre à une requête de lecture d'une ligne d'instructions ou de données, le cache L2 envoie l'état de la ligne (`EXCLUSIVE` ou `SHARED`) dans la réponse, comme pour le protocole **RWT**. Pour les écritures, le protocole **HMESI** utilise les mêmes mécanismes permettant l'écriture différée que le protocole **RWT**. L'objectif est de supprimer tout

le trafic d'écritures entre les caches L1 et L2 dans le réseau primaire. En particulier, la transaction `Cleanup-data` permet au cache L1 de renvoyer une ligne *dirty* au cache L2 en cas d'évincement spontané.

Lorsque le cache L1 reçoit une requête d'écriture venant du processeur, il peut modifier directement une ligne de cache qui se trouve dans l'un des états `EXCLUSIVE` ou `MODIFIED`. Si la ligne est dans l'état `SHARED` ou `INVALID` il faut remplacer la requête d'écriture par un autre type de requête. Si la ligne est dans l'état `INVALID`, le cache L1 doit envoyer une requête au cache L2 pour demander une copie et prévenir d'une future modification locale. Si la ligne est dans l'état `SHARED`, le cache L1 doit prévenir le cache L2 de cette écriture et changer l'état de cette ligne localement de `SHARED` vers `MODIFIED`. Afin de simplifier l'implémentation du protocole, au lieu de définir deux types de requêtes, nous définissons une seule nouvelle requête `GetM` (*Get for Modified*), transmise dans le réseau primaire, et envoyée par le cache L1 au cache L2 en cas d'écriture sur une ligne de cache dans l'un des états `SHARED` ou `INVALID`. La requête `GetM` est traitée par l'automate **`READ`**, puisqu'elle est similaire à une requête de lecture. Le cache L2 répond au `GetM` avec deux informations : les 16 mots de la ligne et l'état `EXCLUSIVE`. Le cache L1 met à jour l'état en `MODIFIED` quand il écrit la ligne en local.

4.1.2 Transactions de cohérence

Chaque ligne de cache valide dans le cache L2 possède les 2 mêmes états que pour le protocole **RWT** : `SHARED` et `EXCLUSIVE`. La transition entre les deux états peut se faire dans les deux sens : Une ligne dans l'état `EXCLUSIVE` peut passer dans l'état `SHARED`, et une ligne dans l'état `SHARED` peut retourner dans l'état `EXCLUSIVE` lorsque le cache L2 reçoit une requête `GetM`. Nous présentons ci-dessous toutes les transactions de cohérence liées au changement d'état dans le cache L2.

De `SHARED` vers `EXCLUSIVE`

Pour les lignes de cache partagées, le répertoire global utilise comme pour **DHCCP** et **RWT**, deux différentes méthodes pour sauvegarder les informations :

- le mode ***liste chainée*** : si le nombre de copies est inférieur ou égal à un certain seuil, le répertoire sauvegarde de manière explicite les identifiants des caches L1 ayant une copie ;
- le mode **Micro-Cache** : si le nombre de copies est supérieur à ce seuil, le répertoire sauvegarde seulement le nombre de caches L1 ayant une copie.

Dans le cas où le cache L2 reçoit une requête de type `GetM` sur une ligne au moins une copie, avant de répondre à cette requête, le cache L2 doit invalider toutes les copies dans les caches L1, y compris l'éventuelle copie du cache L1 qui envoie la requête si la ligne est présente. C'est la réponse à la requête `GetM` qui met à jour l'état de la ligne (figure 4.1). Si la ligne est en mode ***liste chaînée***, le cache L2 envoie la requête `Multicast Invalidate` aux caches L1 contenant la copie. Sinon, si la ligne est en mode ***compteur***, le cache L2 envoie la requête `Broadcast Invalidate` à tous les caches L1. Quand le cache L2 a reçu toutes les requêtes `Cleanup` attendues, aucun

cache L1 ne contient alors de copie. Le cache L2 peut donc envoyer la ligne dans l'état EXCLUSIVE au cache L1 demandeur, puis enregistrer la ligne dans l'état EXCLUSIVE.

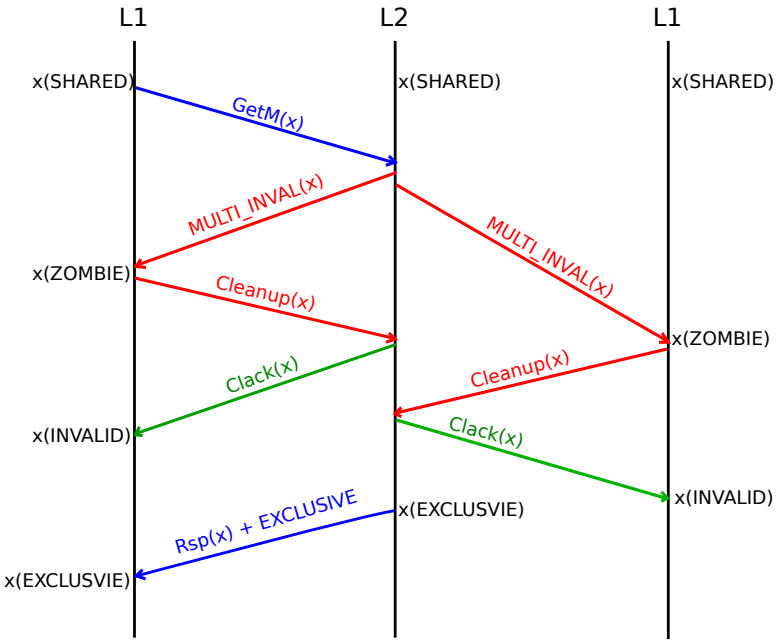


FIGURE 4.1 – Exemple de changement d'état d'une ligne de SHARED à EXCLUSIVE

De EXCLUSIVE à SHARED

Le passage de l'état EXCLUSIVE à l'état SHARED se produit quand un deuxième cache L1 demande une copie (requête de lecture) sur une ligne dans l'état EXCLUSIVE. Dans le protocole **RWT**, le contrôleur de cache L2 invalide la copie dans le cache L1 propriétaire, puis il envoie la réponse de lecture au deuxième cache L1 avec l'état SHARED. Pour respecter le protocole **MESI** standard, la ligne EXCLUSIVE dans le cache L1 propriétaire doit rester valide, mais son état doit être modifié vers SHARED. Le contrôleur de cache L2 donc doit envoyer un nouveau type de requête au cache L1 pour ce changement.

Dans le protocole **DHCCP**, la requête *Multicast Update* est utilisée pour mettre à jour la valeur sur les lignes de cache L1. Le protocole **HMESI** ne supporte pas la mise à jour, et nous pouvons utiliser cette requête de cohérence pour réaliser le changement d'état (de EXCLUSIVE à SHARED) dans le cache L1. Dans cette utilisation, la requête *Multicast Update* ne contient que l'adresse physique de la ligne de cache et elle n'est envoyée qu'au seul cache L1 qui contient la ligne dans l'état EXCLUSIVE ou MODIFIED. Cette requête est rebaptisée *CC-UPDT*. Lorsque le cache L1 qui possède la copie reçoit cette requête, il doit changer l'état de la ligne locale vers SHARED puis comme pour le protocole **DHCCP**, la réponse *Multi Ack* est envoyée au cache L2. La figure 4.2 illustre ce mécanisme.

- si la ligne du cache L1 est dans l'état EXCLUSIVE, la ligne n'est pas modifiée, et le cache L1 envoie une réponse *Multi Ack* simple.
- si la ligne du cache L1 est dans état MODIFIED, la valeur à jour de la ligne doit être transmise au cache L2. Le cache L1 envoie donc une réponse *Multi Ack* avec la ligne complète afin de mettre à jour le cache L2.

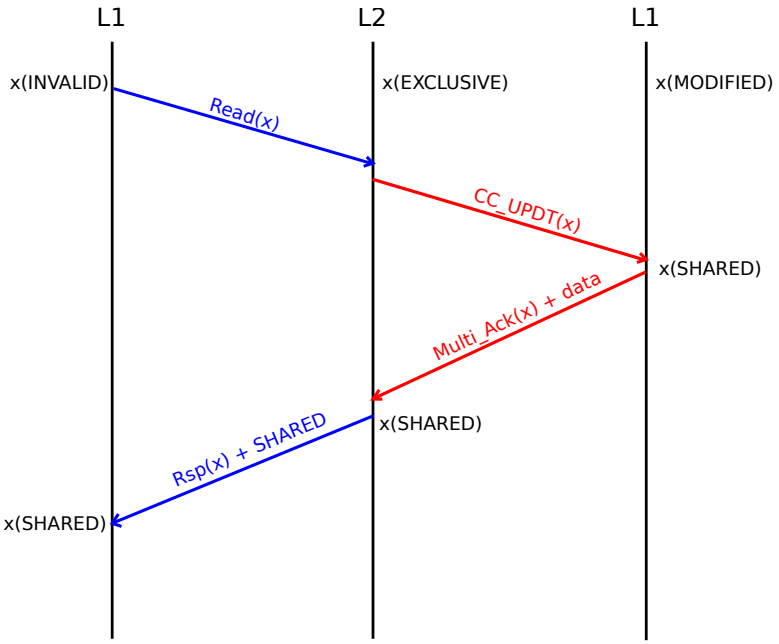


FIGURE 4.2 – Exemple de changement d'état d'une ligne de `EXCLUSIVE` à `SHARED`

De `EXCLUSIVE` à `EXCLUSIVE`

Une ligne de cache dans l'état `EXCLUSIVE` peut changer de propriétaire lorsque le cache L2 reçoit une requête de type *`GetM`* sur cette ligne. Le cache L2 doit s'assurer que la requête *`GetM`* vient bien d'un autre cache L1 par rapport au propriétaire actuel. Le cache L2 envoie au propriétaire actuel une requête de type *`Multicast Invalidate`* afin d'invalider la copie exclusive et reçoit une requête *`Cleanup`* ou *`Cleanup-data`* si la ligne est en état `MODIFIED`. Ensuite le cache L2 change le propriétaire de cette ligne et répond à la requête *`GetM`* du nouveau propriétaire, comme montré figure 4.3.

4.2 Implémentation du protocole HMESI

Le protocole **HMESI** décrit ci-dessus comporte autant de types de transactions de cohérence que le protocole **DHCCP**, et en plus chaque type de transaction comporte autant de requêtes que le protocole **DHCCP**. Nous avons simplement redéfini les quatre types de transactions de cohérence

- **T0** : lorsque le cache L2 reçoit une requête *`GetM`* ou évince une ligne en mode ***liste chaînée***, il déclenche cette transaction afin d'invalider toutes les copies dans les caches L1.
- **T1** : lorsque le cache L2 reçoit une requête *`GetM`* ou évince une ligne en mode ***compteur***, il déclenche cette transaction afin d'invalider toutes les copies dans les caches L1.
- **T2** : le cache L1 évince une ligne spontanément, il envoie la requête *`Cleanup-data`* si la ligne est *Dirty*.
- **T3** : cette transaction est uniquement utilisée pour le changement de l'état de `EXCLUSIVE` à `SHARED`. Le cache L1 envoie la requête *`Multi ack-data`* lorsque

la ligne est *Dirty*.

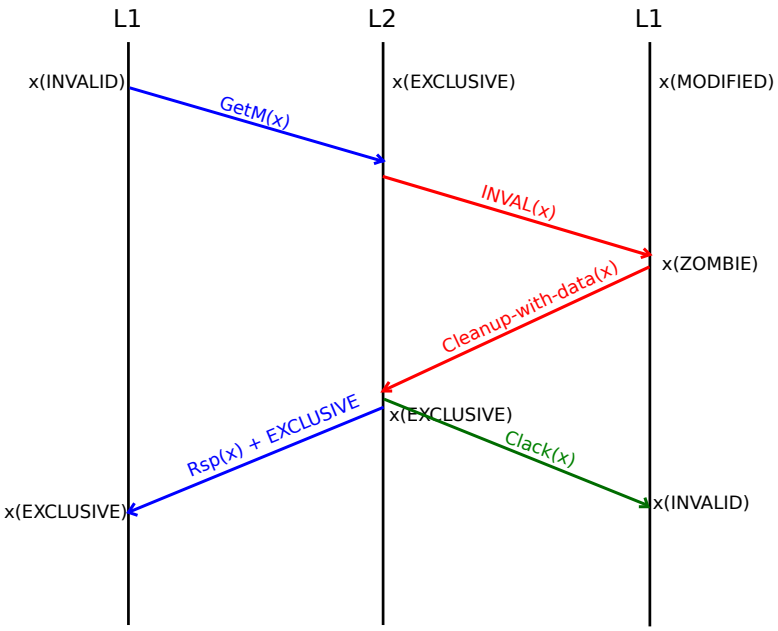


FIGURE 4.3 – Exemple de changement d'état d'une ligne de `EXCLUSIVE` à `EXCLUSIVE`

4.2.1 Table d'Invalidations

Le protocole **HMESI** utilise une seule table d'invalidations (dite ***table IVT***) dans le cache L2 pour sauvegarder toutes les informations sur les transactions de cohérence en cours et les requêtes qui provoquent ces dernières. Il y a deux objectifs à cela :

- détecter la fin des transactions de cohérence afin de mettre à jour l'état de ligne de cache.
- sans mécanisme particulier, une requête de lecture qui provoque une transaction de cohérence bloque l'automate **`READ`** jusqu'à la fin de cette transaction, car on ne peut répondre à la lecture qu'une fois la transaction de cohérence finie. La ***table IVT*** lève cette contrainte en sauvegardant les requêtes de lecture qui nécessitent une transaction de cohérence ; ainsi, ces requêtes ne bloquent pas l'automate **`READ`** puisque leur réponse peut être envoyée plus tard par d'autres automates.

Cinq automates du contrôleur de cache L2 peuvent accéder à la ***table IVT*** :

- L'automate **`READ`** traite les requêtes de lectures et `GetM`, il sauvegarde les requêtes dans la ***table IVT*** lorsque ces requêtes provoquent un changement d'état.
- L'automate **`WRITE`** traite les requêtes d'écritures, il sauvegarde les écritures dans la ***table IVT*** lorsque la ligne écrite possède des copies dans les caches L1.
- L'automate **`CLEANUP`** traite les requêtes `Cleanup` et `Cleanup-data`, il retire les requêtes de `GetM` ou les écritures de la ***table IVT*** à la fin du changement d'état (c.-à-d. lorsque l'automate **`CLEANUP`** a reçu tous les `Cleanup` ou `Cleanup-data` associés à la requête) et envoie la réponse au cache L1 correspondant.

- l'automate **MULTI_ACK** traite la requête *Multi ack*, il retire les requêtes de lecture de la ***table IVT*** lorsqu'il reçoit les requêtes *Multi ack* associées et envoie la réponse au cache L1 correspondant.

- l'automate **XRAM_RSP** prend en charge la mise à jour des lignes dans le cache L2 lorsqu'il reçoit la réponse à une lecture venant de la mémoire principale. Il sauvegarde la ligne victime dans la ***table IVT*** afin de faire une place pour la nouvelle ligne.

La ***table IVT*** est accédée de manière concurrente par 5 automates. Lorsqu'un automate est en cours de lecture ou d'écriture de la table, les autres automates doivent attendre jusqu'à ce que la table soit relâchée. En conséquence, la fréquence d'accès à la ***table IVT*** est un facteur important de la performance du cache L2.

4.2.2 État LOCKED

Dans le protocole **RWT**, lorsque le changement d'état d'une ligne de NC à C est démarré, la ligne de cache reste dans l'état NC pendant toute la durée de ce processus. Le contrôleur de cache L2 bloque les requêtes ciblant la même ligne en vérifiant s'il existe une entrée pour cette ligne dans la ***table IVT***. Mais dans le protocole **HMESI**, il existe pour une ligne trois façons de changer d'état, donc une ligne de cache a plus de chance de changer d'état que dans le protocole **RWT**. Autrement dit, les accès concurrents à la ***table IVT*** par les différents automates réduisent la bande passante du cache L2, car le temps d'attente pour obtenir l'autorisation d'accès à la ***table IVT*** est augmenté. Afin de baisser la contention sur cette table, nous introduisons un nouvel état pour chaque ligne de cache L2, appelé LOCKED. Lorsqu'une ligne de cache est dans l'état LOCKED, elle est valide dans le cache L2, mais en cours de changement d'état ; elle est donc temporairement inaccessible.

Dans les scénarios montrés sur la figure 4.4, lorsque la ligne x est dans l'état LOCKED, une autre lecture sur la ligne x est bloquée dans l'automate **READ** (figure a). Cependant, l'automate **READ** peut répondre à une lecture sur la ligne y. Dans les deux situations, l'automate **READ** n'a pas besoin d'accéder à la ***table IVT*** : grâce à l'état LOCKED, il sait que la ligne x est présente dans la table et que la ligne y n'y est pas présente.

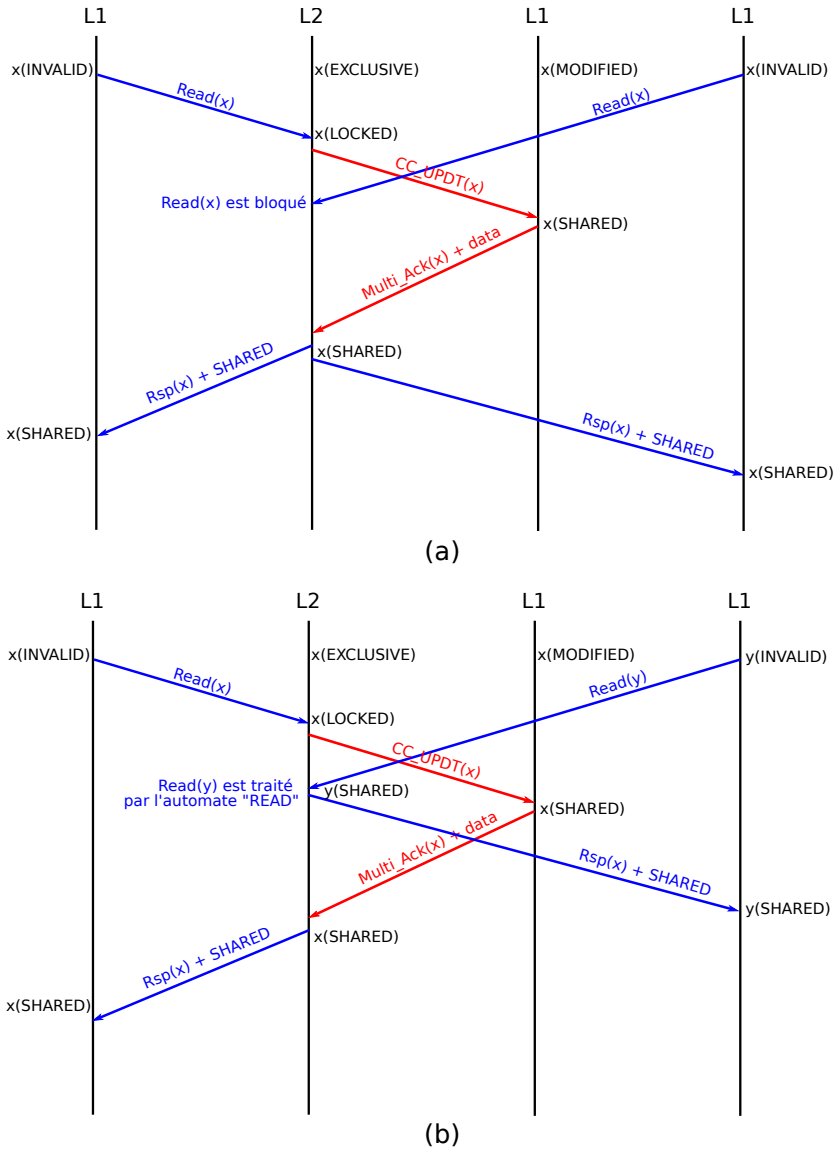


FIGURE 4.4 – Utilité de l'état `LOCKED` lors d'un changement d'état

Il y a trois avantages à avoir un état `LOCKED` :

- le contrôleur de cache L2 peut savoir qu'une ligne est en cours de changement d'état en lisant seulement le répertoire global. Lorsque la ligne est dans l'état `LOCKED`, il bloque la requête. L'automate **READ** n'a plus besoin de vérifier la ***table IVT*** pour savoir si une ligne valide y est contenue (une ligne dans l'état `LOCKED` est forcément sauvegardée dans la ***table IVT***).
- en cas d'évincement spontané d'une ligne dans le cache L2, les lignes présentes dans la ***table IVT*** ne peuvent pas être sélectionnées comme victimes. Sans état `LOCKED`, cela nécessite d'accéder à la ***table IVT*** après avoir sélectionné une ligne victime afin de vérifier qu'elle ne s'y trouve pas. Grâce à l'état `LOCKED`, cet accès n'est plus nécessaire, car toutes les lignes dans la ***table IVT*** sont dans l'état `LOCKED`, il suffit donc de ne pas sélectionner de ligne dans l'état `LOCKED`.
- Dans le protocole **HMESI**, il n'existe pas de requête d'écriture venant des caches L1 (les mises à jour du L2 se font par l'intermédiaire des requêtes `Cleanup-data`),

mais les autres périphériques comme le **DMA** peuvent encore envoyer des écritures au cache L2. Si une ligne cible d'une écriture est présente dans le cache L2, au lieu de l'invalider (dans le protocole **DHCCP**, les lignes en mode ***compteur*** du cache L2 sont évincées), la ligne peut rester valide dans l'état `LOCKED`. Après complétion de l'invalidation de toutes les copies dans les caches L1, la ligne peut passer dans l'état `EXCLUSIVE`.

En conclusion, l'introduction de l'état `LOCKED` simplifie significativement l'implémentation du contrôleur de cache L2, car cet état permet de facilement distinguer les lignes de cache en cours de changement d'état.

La figure 4.5 montre les transitions de changement d'état entre les quatre états :

- Les transitions *a* et *h* représentent le premier accès à une ligne de cache donnée ou instruction.
- Les séquences de transitions *j*, $\{ b \rightarrow f \}$ et $\{ e \rightarrow c \}$ montrent les trois changements d'état sur une ligne valide.
- Les transitions *d* et *i* sont prises à la fin de vie d'une ligne valide dans le cache L2.

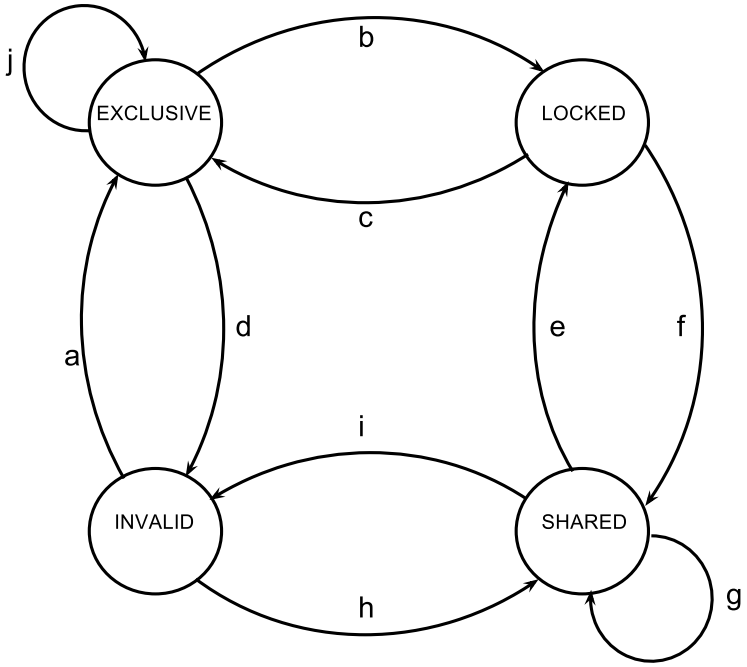


FIGURE 4.5 – États et transitions entre états pour une ligne dans le cache L2

4.2.3 Diagramme de transition du cache L1

Parmi les diagrammes d'états du cache L1 des trois protocoles **DHCCP**, **RWT** et **HMESI**, celui du protocole **HMESI** est le plus complexe (figure 4.6) :

- lorsque le cache L1 reçoit la réponse à une requête de lecture ou `GetM` sur une ligne absente, la ligne est stockée dans le cache L1 en passant la transition *a* ou *b*.
- les transitions *h* et *f* sont prises lorsque le cache L1 reçoit la requête `CC-UPDT`.

- la lecture du cache L1 ne change pas l'état de la ligne de cache (transitions *c*, *d* et *e*).
- les écritures modifient directement la ligne exclusive (transitions *g* et *d*).
- en cas d'invalidation d'une ligne de cache, celle-ci passe forcément dans l'état `ZOMBIE`, séquences $\{ j \rightarrow m \}$, $\{ i \rightarrow m \}$ et $\{ k \rightarrow m \}$.

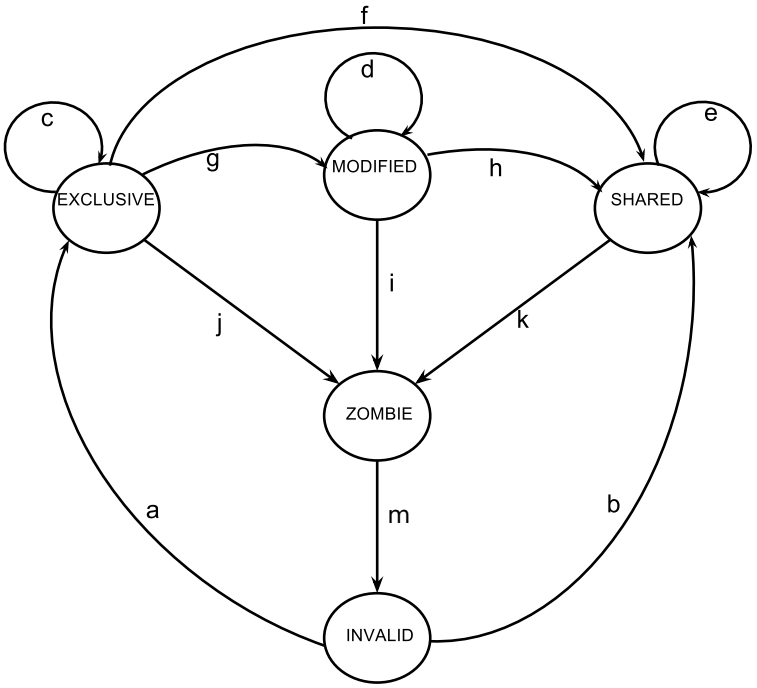


FIGURE 4.6 – Diagramme d'états d'une ligne dans le cache L1

4.2.4 Requête *Multi ack miss*

Le protocole **HMESI** est plus complexe que le protocole **RWT**, non seulement parce que ses changements d'état sont plus nombreux, mais aussi, car il existe certaines situations qui sont plus difficiles à résoudre que pour les autres protocoles. Dans cette section, nous donnons quelques exemples de situations complexes.

Le contrôleur de cache L2 envoie la requête *CC-UPDT* au cache L1 pour changer l'état d'une ligne L_0 de cache de `EXCLUSIVE` à `SHARED`. Lorsque le changement s'est fait localement, le cache L1 envoie le *Multi ack* au cache L2 pour signifier le traitement. En effet, la ligne L_0 peut avoir été évincée lorsque la requête arrive au cache L1. En conséquence, il existe deux différentes situations dans lesquelles le cache L1 reçoit la requête *CC-UPDT* :

- si la ligne L_0 est toujours présente dans le cache L1, le cache L1 met à jour l'état de cette ligne et envoie une requête *Multi ack* ou *Multi ack-data* afin d'informer le cache L2 du succès de ce changement.
- si la ligne L_0 a été évincée, autrement dit si une requête *Cleanup* ou *Cleanup-data* sur la ligne a déjà été envoyée, le cache L1 envoie donc une requête *Multi ack miss* au cache L2 pour indiquer l'absence de la ligne.

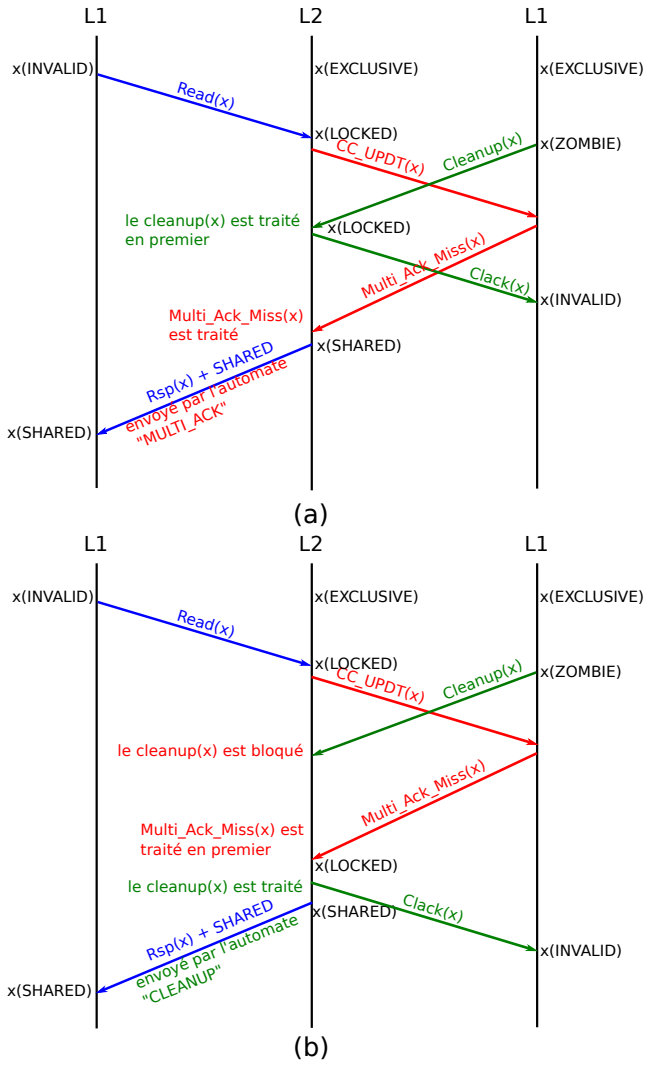


FIGURE 4.7 – Deux scénarios en cas de *Multi ack miss*

Dans le cache L2, l'automate **MULTI_ACK** traite tous les types de *Multi ack* venants du cache L1 afin de répondre aux requêtes de lecture sur une ligne EXCLUSIVE :

- s'il reçoit un *Multi ack* ou *Multi ack-data*, le cache L2 confirme que la ligne a été modifiée en état SHARED dans le cache L1. L'automate **MULTI_ACK** d'abord met à jour les données de la ligne en cas de *Multi ack-data*, après quoi il modifie l'état à SHARED, puis envoie la réponse de lecture au cache L1 correspondant (figure 4.2).
- En cas de *Multi ack miss*, puisque les requêtes *Cleanup* (ou *Cleanup-data*) et *Multi ack miss* sont transmis sur le même canal de communication, le *Cleanup* (ou *Cleanup-data*) arrive donc au cache L2 forcément plus tôt que le *Multi ack miss*. Cependant les deux transactions sont traitées par deux différents automates. Si l'automate **CLEANUP** est occupé et qu'il ne peut pas traiter le *Cleanup* ou *Cleanup-data* immédiatement, l'automate **MULTI_ACK** traitera probablement le *Multi ack miss* avant que le *Cleanup* ou *Cleanup-data* soit traité. En conséquence, deux différentes situations possibles peuvent arriver :
  - Au cas où l'automate **CLEANUP** a déjà traité le *Cleanup* ou *Cleanup-data* sur la ligne L_0, l'automate **MULTI_ACK** peut répondre à la requête de lecture,

car la ligne a été mise à jour dans le cache L2 (figure 4.7.a).

- Si l'automate **MULTI_ACK** observe que l'automate **CLEANUP** n'a pas encore reçu le *Cleanup* ou *Cleanup-data*, il ne doit pas répondre à la requête de lecture et signaler à l'automate **CLEANUP** que celui-ci doit répondre à cette requête lorsqu'il recevra le *Cleanup* ou *Cleanup-data* (figure 4.7.b). Nous utilisons cette solution pour éviter le blocage de l'automate **MULTI_ACK**, car cette attente (si l'automate **MULTI_ACK** attend que la ligne soit traitée par l'automate **CLEANUP**) peut produire un interblocage entre les automates **CLEANUP** et **MULTI_ACK**.

Un problème se pose donc : entre l'automate **CLEANUP** et **MULTI_ACK**, qui doit répondre à la requête de lecture lorsque le *Cleanup* ou *Multi ack miss* arrive au cache L2 ?

Dans le cache L2, une seule ***table IVT*** est implémentée afin de sauvegarder toutes les requêtes provoquées une transaction de cohérence. Cette table peut répondre à la question ci-dessus : lorsque la ***table IVT*** sauvegarde une ligne EXCLUSIVE correspondant à une requête de lecture, une information est ajoutée sur cette ligne afin de connaître la première arrivée dans le cache L2 entre le *Cleanup* (ou *Cleanup-data*) et le *Multi ack miss* :

- Si l'automate **CLEANUP** reçoit le *Cleanup* ou *Cleanup-data* avant que l'automate **MULTI_ACK** ne reçoive le *Multi ack miss*, il accède en premier à la ***table IVT*** afin de récupérer les informations sur cette ligne. Il observe alors que la ligne est en cours d'attente du *Multi ack miss*, mais que l'automate **MULTI_ACK** n'a pas encore signifié l'arrivée du *Multi ack miss*. L'automate **CLEANUP** met alors à jour l'entrée de la ***table IVT*** correspondante à la ligne pour signifier que le *Cleanup* ou *Cleanup-data* a été reçu. Les valeurs de la ligne sont aussi mises à jour dans la partie donnée du cache en cas de *Cleanup-data*, mais l'état de la ligne reste à LOCKED. Lorsque le *Multi ack miss* arrive à l'automate **MULTI_ACK** par la suite, ce dernier observe que la ligne a été mise à jour en lisant la ***table IVT***. Par conséquent l'automate **MULTI_ACK** répond à la requête de lecture et modifie l'état de LOCKED à SHARED (afin de simplifier l'implémentation pour cette situation particulière, cette ligne est dans l'état SHARED même si elle a une seule copie). À la fin, l'automate **MULTI_ACK** supprime l'entrée correspondante de la ***table IVT***.
- Réciproquement, si l'automate **MULTI_ACK** signifie le premier l'arrivée du *Multi ack miss* en mettant à jour l'information sur la ligne dans la ***table IVT***, l'automate **CLEANUP** prend en charge de mettre à jour la ligne dans le cache L2 et de répondre à la lecture. L'automate **CLEANUP** supprime donc l'entrée correspondante de ***table IVT***.

4.3 Protocole MOESI dans TSAR ?

Dans le protocole **HMESI**, il n'existe pas d'état OWNED. Le protocole **MOESI** utilise un état OWNED pour éviter de mettre à jour la mémoire principale lorsque le répertoire global reçoit une lecture sur la ligne "EXCLUSIVE". Cette opération coûte 3 indirections et utilise 4 transactions (figure 4.8.a) :

- Première indirection : un cache L1 envoie une requête de lecture au répertoire global.
- Seconde indirection : le répertoire global transfère cette requête au cache L1 qui possède la ligne dans l'état `OWNED`.
- Troisième indirection : ce cache L1 répond à la requête de lecture du demandeur et envoie simultanément un message au répertoire global afin de l'informer de la transaction.

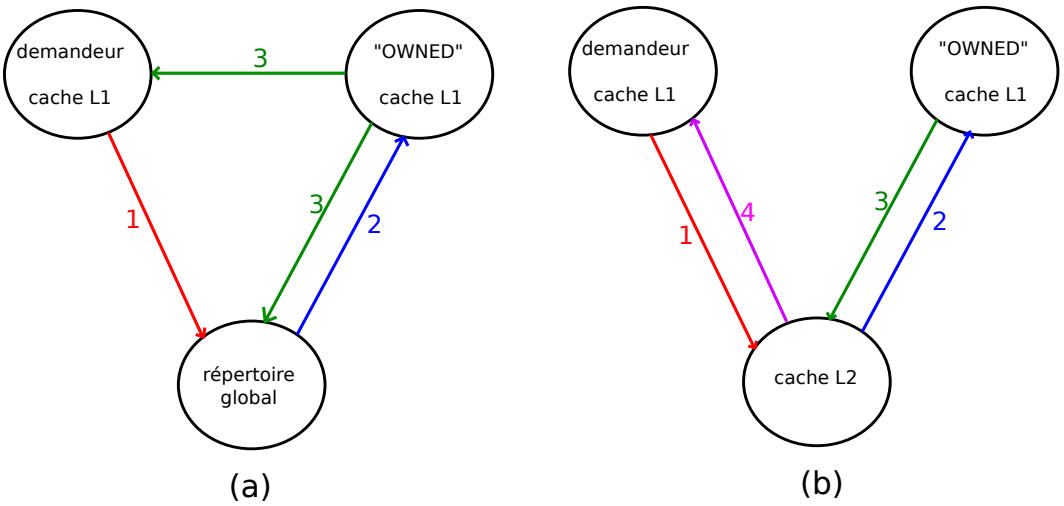


FIGURE 4.8 – Différents traitements sur les protocoles **MOESI** et **HMESI** en cas de lecture d'une ligne `EXCLUSIVE`

Afin de supporter le protocole **MOESI**, la communication directe entre caches L1 doit donc être implémentée dans le réseau de communication. Dans l'architecture **TSAR**, le cache L2 partagé possède à la fois la fonction de répertoire global, et la fonction de cache de deuxième niveau qui stocke les valeurs des lignes dans la partie ***données***. Au lieu de demander au cache L1 `OWNED` de répondre à une lecture, le cache L2 peut sauvegarder la valeur à jour de la ligne dans sa zone des données sans mettre à jour la mémoire principale. Cette opération coûte 4 indirections et utilise 4 transactions pour changer l'état de `EXCLUSIVE` à `SHARED` (figure 4.8.b), mais ne requiert pas d'implémenter la communication entre caches L1. De plus, la ligne est à jour dans le cache L2, donc ce dernier peut répondre directement aux requêtes de lecture suivantes sur cette ligne. Nous ne pensons donc pas indispensable d'implémenter explicitement un protocole **MOESI** dans **TSAR**.

4.4 Conclusion

Pour évaluer le protocole **RWT** (qui est notre principale contribution), et le comparer aux protocoles actuellement utilisés dans l'industrie, nous avons implémenté le protocole **HMESI** comme protocole de référence dans l'architecture **TSAR**. Comme les protocoles **DHCCP** et **RWT**, le protocole **HMESI** utilise deux moyens pour sauvegarder les identifiants des caches L1 : ***liste chaînée*** et ***compteur***, et utilise les mêmes ressources matérielles telles que la table des transactions de cohérence en cours.

Ainsi, nous avons trois différents protocoles de cohérence des caches : **DHCCP**, **RWT** et **HMESI** qui sont implémentés sur la même architecture **TSAR**, et nous pouvons analyser les performances et le coût énergétique associés à ces différentes stratégies.

Chapitre 5

Cohérence des TLBs

Dans ce chapitre, nous proposons un nouveau mécanisme permettant d'améliorer le traitement matériel de la cohérence des **TLBs** dans l'architecture TSAR.

5.1 Mémoire virtuelle paginée dans TSAR

Rappelons que dans l'architecture **TSAR**, les contrôleurs de cache L1 contiennent une **MMU** générique implémentant une mémoire virtuelle paginée à deux niveaux. Cette **MMU** est située entre le processeur et les caches L1 (indexés en adresses physiques). La **MMU** est responsable des traductions (adresse virtuelle -> adresse physique), et doit, pour chaque accès mémoire réalisé par le processeur, calculer le **PPN** (*Physical Page Number*, sur 28 bits) à partir du **VPN** (*Virtual Page Number* sur 20 bits), et du registre **PTPR** contenant l'adresse de base de la table des pages. La valeur contenue dans le registre **PTPR** est évidemment modifiée à chaque changement de contexte, puisque chaque application possède sa propre table de pages.

Cette traduction nécessite au moins deux accès à la mémoire, puisqu'il faut commencer par lire dans la table de premier niveau la valeur du **PTN** (pointeur sur la table de page de deuxième niveau). Les valeurs des index *IX1* et *IX2* sont obtenues à partir du **VPN**.

1. PTN = M[PTPR + IX1]
2. PPN = M[PTN + IX2]

Pour accélérer cette traduction, la **MMU** générique de **TSAR** contient deux **TLBs** séparées, pour les adresses d'instructions, et pour les adresses de données. Ces **TLBs** sont implémentées comme des petits caches associatifs par ensemble, permettant d'enregistrer jusqu'à 64 paires (**VPN** -> **PPN**) récemment utilisées. Le **VPN** est stocké dans la partie *répertoire*, et le **PPN** ainsi que quelques bits définissant les droits d'accès, sont stockés dans la partie de données.

En cas de MISS sur une **TLB**, un automate câblé (appelé *Table-Walk*) se charge d'accéder aux tables de pages construites en mémoire par le système d'exploitation pour obtenir la paire (**VPN** -> **PPN**) manquante et met à jour la **TLB**.

Puisque les **TLBs** sont des petits caches, et que le contenu des **TLBs** est une copie partielle du contenu des tables de pages stockées en mémoire, toute modification

dynamique des tables de pages par le système d'exploitation crée un problème d'obsolescence des **TLBs** par rapport à la mémoire, qui est traditionnellement traité par logiciel.

5.2 Méthode actuellement utilisée pour la cohérence des TLB

Dans l'architecture **TSAR**, c'est le protocole **DHCCP** qui assure - en matériel - la cohérence des **TLBs**. Le mécanisme général, proposé par Yang Gao et Alain Greiner [13] est le suivant (figure 5.1) :

- Les tables de pages sont cachables.
- En cas de miss sur une **TLB**, l'automate **`Table-Walk`** doit calculer (à partir du **VPN** de l'adresse virtuelle demandée et du **PTPR**) l'adresse du **PTN** puis l'adresse du **PPN** recherché.
- L'automate **`Table-Walk`** ne s'adresse jamais directement à la mémoire, mais sous-traite les accès mémoire au contrôleur du cache de données L1 local.
- Si le cache de données L1 local contient la ligne de cache contenant l'adresse demandée, il retourne le **PTN** ou le **PPN** manquant et la **TLB** est mise à jour. Sinon, le contrôleur du cache de données se charge de rapatrier la ligne de cache manquante.
- Puisque le protocole **DHCCP** assure la cohérence entre les caches de données L1 et la mémoire, toute modification de la table des pages par l'OS sera répercutée dans tous les caches L1 possédant une copie.
- Le protocole **DHCCP** stipule que le contrôleur du cache L1 maintient pour chaque entrée du répertoire de donnée un bit ***is_ppn*** indiquant que la ligne appartient à une table de page de deuxième niveau, et contient donc des **PPNs** susceptibles d'avoir été copiés dans une des **TLBs**.
- De même, le protocole **DHCCP** stipule que le contrôleur du cache L1 maintient pour chaque entrée du répertoire un bit ***is_ptn*** indiquant que la page appartient à une table de page de premier niveau, et contient donc des **PTNs** susceptibles d'avoir été utilisés pour calculer les **PPNs** enregistrés dans les **TLBs**.
- Le protocole **DHCCP** stipule que toute paire (**VPN** -> **PPN**) contenue dans une **TLB** contient également un champ ***nline*** qui est le numéro de la ligne de cache contenant le **PPN**.
- Si le contrôleur de cache L1 reçoit une requête de cohérence *`Invalidate`*(***nline***) ou *`Update`*(***nline***) pour une ligne ***nline*** possédant le flag ***is_ppn***, cet événement déclenche un parcours associatif de toutes les entrées des deux **TLBs** pour invalider les entrées contenues dans la ligne ***nline*** (opération *`Scan-TLB`*).
- Si le contrôleur de cache L1 reçoit une requête de cohérence *`Invalidate`*(***nline***) ou *`Update`*(***nline***) pour une ligne ***nline*** possédant le flag `is_ptn`, cet événement déclenche l'invalidation de toutes les entrées des deux **TLBs** (opération *`Reset-TLB`*).

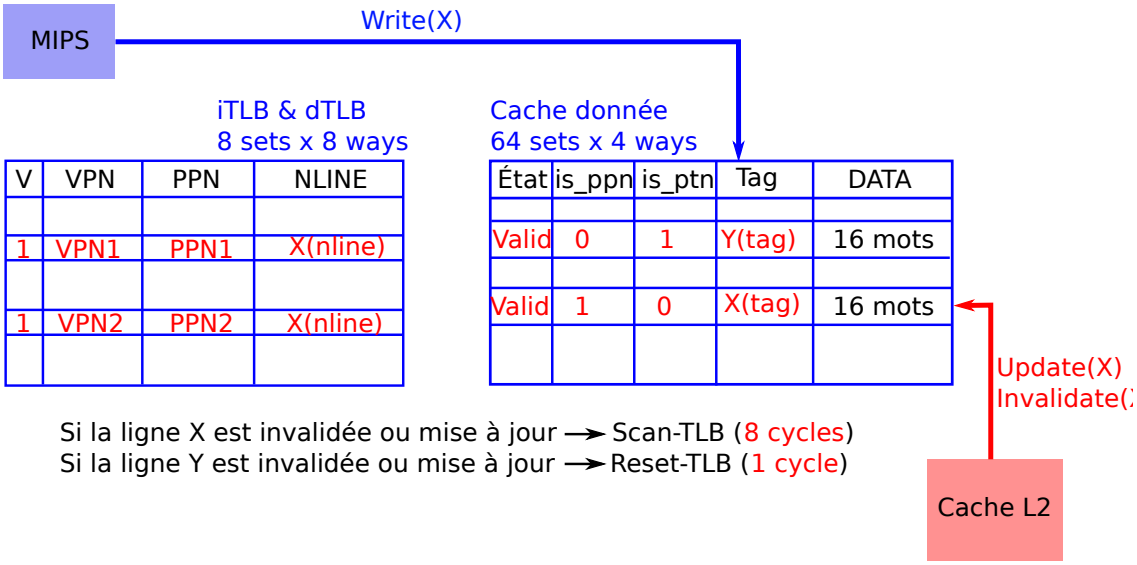


FIGURE 5.1 – Solution de cohérence des **TLBs** dans l'architecture TSAR

Dans la suite de ce chapitre, on dira qu'une ligne de cache est de type ***in_tlb*** lorsqu'elle contient soit un **PPN** (bit ***is_ppn*** à 1), soit un **PTN** (bit ***is_ptn*** à 1).

Cette méthode impose une contrainte d'inclusivité forte : si un **PPN** appartenant à une ligne de cache X est présent dans une des deux **TLBs**, alors la ligne de cache X doit être présente dans le cache de donnée L1 correspondant, pour que les requêtes de cohérence `Invalidate`(X) ou `Update`(X) soient correctement répercutées vers les **TLBs**. Le corollaire est que tout évincement d'une ligne du cache L1 impose d'invalider toutes les entées des **TLBs** contenues dans cette ligne de cache.

Comme en pratique, chaque *Miss* dans le cache L1 impose un évincement spontané (pour libérer une case), cette méthode entraîne un nombre important d'invalidations inutiles dans les **TLBs**, à chaque fois que la ligne évincée est de type ***in_tlb***. Ceci augmente le nombre de *Miss* **TLB** et donc la consommation énergétique.

5.3 Méthode permettant de relâcher la contrainte d'inclusivité

L'objectif principal de la méthode proposée est de relâcher la contrainte d'inclusivité stricte des **TLBs** dans le cache de données L1, afin d'éviter les invalidations parasites des **TLBs** en cas d'évincement spontané d'une ligne de cache de type ***in_tlb***.

Pour cela, nous proposons d'introduire dans le cache L1, une nouvelle table nommée **CC-TLB** qui concerne exclusivement les lignes de type ***in_tlb***.

5.3.1 Principe de la table CC-TLB

Lorsqu'une entrée de table des pages est modifiée par le système d'exploitation, cette modification est réalisée en deux étapes :

1. La requête d'écriture dans la table des pages modifie directement la ligne X contenant cette entrée de la table des pages, si cette ligne est présente dans le cache L1 du processeur qui effectue l'écriture.
2. Cette requête d'écriture est envoyée au cache L2, qui déclenche des requêtes de cohérence `Invalidate`(X) ou `Update`(X) vers tous les autres caches L1 qui contiennent cette ligne X.

Quand un cache L1 reçoit une requête d'écriture par le processeur `Write`(X) ou une requête de cohérence (`Invalidate`(X) ou `Update`(X)) par le cache L2, il doit invalider les entrées de **TLB** qui correspondent à cette ligne. En revanche, s'il s'agit d'un évincement spontané pour faire de la place, les entrées de **TLB** contenues dans la ligne évincée ne doivent pas être invalidées.

Pour que la cohérence des **TLBs** soit maintenue sans que les entrées des **TLBs** ne soient invalidées, deux conditions doivent être respectées :

1. Il faut que le cache L1 continue à recevoir les requêtes de cohérence si la ligne X évincée est modifiée.
2. Il faut que le contrôleur du cache L1 conserve l'information que les **TLBs** contiennent des copies de la ligne X, même si celle-ci a été évincée du cache L1, pour pouvoir exploiter les requêtes de cohérence concernant la ligne X.

Pour atteindre cet objectif sans modifier le comportement du cache L2, il suffit que le cache L1 n'envoie pas de requête `Cleanup` (X) au cache L2 quand il évince une ligne X de type `in_tlb`. Au lieu de cela, il doit enregistrer certaines informations concernant la ligne X évincée dans la table **CC-TLB**, et en particulier le nombre d'entrées de **TLB** qui doivent être invalidées si une requête de cohérence concernant X est reçue par le cache L1.

5.3.2 Structure de table CC-TLB

La table **CC-TLB** est une table associative par ensembles permettant d'associer à chaque ligne de cache de type ***in_tlb*** le nombre d'entrées dans les **TLBs** qu'elle contient, plus quelques bits d'état. Elle est organisée en 8 ensembles associatifs (*sets*) contenant chacun 8 voies (*ways*).

Le format d'une entrée dans la table **CC-TLB** est décrit par la figure 5.2).

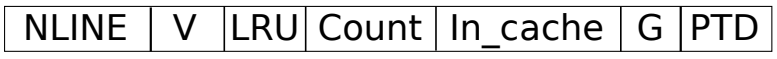


FIGURE 5.2 – Une entrée de la table **CC-TLB**

1. ***NLINE*** (31 bits) adresse physique de la ligne ***in_tlb*** (34 bits moins les 3 bits du *set*).
2. ***V*** (1 bit) entrée valide.
3. ***LRU*** (1 bit) implémente le mécanisme *Least Recently Used* pour le choix d'une victime.
4. ***Count*** (5 bits) nombre d'entrée **TLBs** (au maximum 16 entrées dans une ligne).

5. ***In_cache*** (1 bit) indique la présence de cette ligne ***in_tlb*** dans le cache donnée.
6. ***G*** (1 bit) indique que la ligne contient des entrées appartenant au noyau.
7. ***PTD*** (1 bit) signale que cette ligne ***in_tlb*** contient des entrées de type ***is_ptn***.

Ces sept champs sont utilisés de la façon suivante :

Lorsqu'une ligne X de type ***in_tlb*** est évincée spontanément du cache L1, le champ ***In_cache*** est désactivé. Les entrées de **TLB** correspondantes restent valides. Le contrôleur de cache L1 n'envoie pas de commande `Cleanup` (X) au cache L2.

En cas de *Miss* **TLB**, l'automate chargé de traiter le *Miss* **TLB** s'adresse au cache L1 pour obtenir le **PPN**, et accède à la table **CC-TLB** pour savoir si la ligne X contenant ce **PPN** est enregistrée.

- Si la ligne X est présente à la fois dans le cache et dans la table **CC-TLB**, l'automate incrémente le champ ***Count*** de cette ligne dans la table **CC-TLB**.
- Si la ligne X est présente dans le cache, mais pas dans la table **CC-TLB**, l'automate sélectionne une nouvelle entrée dans la table en utilisant l'algorithme pseudo-LRU, puis l'initialise (***Count*** = 1 / ***In_cache*** = 1).
- Si la ligne X n'est présente ni dans le cache L1, ni dans la table **CC-TLB**, l'automate envoie au cache L2 une requête de lecture normale pour la ligne X, et sélectionne une nouvelle entrée dans la table en utilisant l'algorithme pseudo-**LRU**, puis l'initialise (***Count*** = 1 / ***In_cache*** = 1).
- Si la ligne X n'est pas présente dans le cache, mais est présente dans la table **CC-TLB**, cela signifie que la ligne a été invalidée dans le cache L1, mais que le cache L2 n'en est pas informé. L'automate envoie au cache L2 une requête de lecture *Uncached* pour la ligne X, qui ne modifie pas l'état de la ligne dans le répertoire du cache L2. Le cache L1 et mis à jour, et la table **CC-TLB** également (incrémentation de ***Count*** et ***In_cache*** = 1).

Lorsque le cache L1 reçoit une requête d'écriture `Write`(X), ou une requête de cohérence `Cleanup`(X) ou `Invalidate`(X) pour une ligne de type ***in_tlb***, l'opération `Scan-TLB` ou `Reset-TLB` n'est effectuée que si le champ ***Count*** contient une valeur supérieure à 0.

En cas de changement de contexte dans un processeur, il faut invalider toutes les entrées des **TLBs** puisque chaque contexte possède sa propre table de pages, mais les entrées correspondant aux segments du système d'exploitation (identifiés bar le bit ***G*** : global) sont répliquées dans toutes les tables de pages, et ne sont pas invalidées. Pour ce qui concerne la table **CC-TLB**, toutes les entrées non globales (bit ***G*** = 0) sont re-initialisées en cas de changement de contexte (champ ***Count*** = 0).

5.3.3 Coût matériel de la solution proposée

Puisque la table **CC-TLB** contient toutes les informations nécessaires au maintien de la cohérence des **TLBs**, les différentes extensions introduites dans le répertoire du cache L1 et dans les **TLBs** dans la solution précédente peuvent être supprimées.

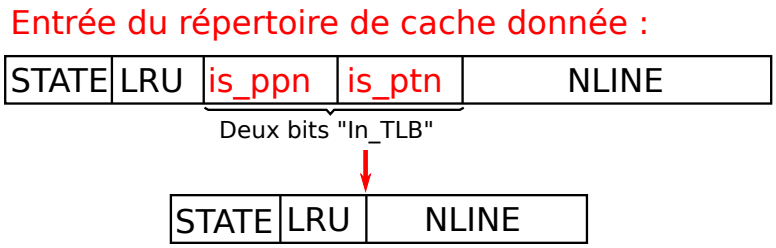


FIGURE 5.3 – Modification dans le répertoire du cache de données

Pour le cache L1, cela concerne les deux bits ***is_ppn*** et ***is_ptn***, présents dans chaque entrée du répertoire. Dans le cas de l'architecture **TSAR**, qui possède des caches de 16 Koctets et des lignes de caches de 64 octets, on économise 256 * 2 = 512 bits (figure 5.3). Pour les **TLBs**, cela concerne le numéro de ligne ***nline***, codé sur 34 bits, qui est remplacé par un index dans la table **CC-TLB**, codé sur 6 bits. Dans le cas de TSAR, on a 2 **TLBs** possédant 64 entrées, on économise donc 2 * 64 * (34 - 6) = 3 584 bits par cache L1 (figure 5.4).

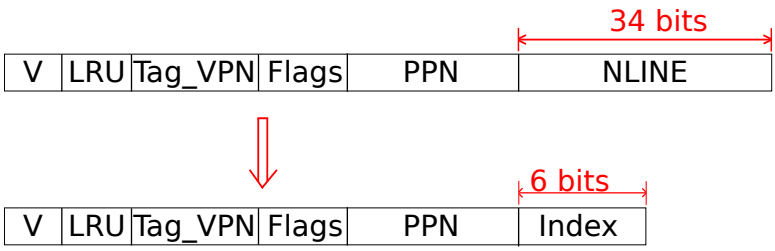


FIGURE 5.4 – Modification dans la TLB

Le nombre de bits mémorisés dans la table **CC-TLB** étant de 64 * 41 = 2624, on constate que cette solution n'augmente pas, mais au contraire diminue le coût matériel, puisqu'on ajoute 2624 bascules, mais qu'on en supprime 3 584.

5.3.4 Fonctionnement détaillé de CC-TLB

Le fonctionnement détaillé de la table **CC-TLB**, est illustré dans le tableau 5.5 qui montre l'évolution des informations sur une ligne X de type ***in_tlb*** stockées dans les trois composants (**TLB**, **CC-TLB**, et cache L1) au cours du temps.

TLB	**CC-TLB**			cache L1	
NOMBRE ENTRÉES	VALID	COUNT	In_cache	VALID	événement
0	0	0	0	0	a
0	0	0	0	1	b
1	1	1	1	1	c
2	1	2	1	1	d
2	1	2	0	0	e
3	1	3	1	1	f
0	0	0	0	0	g
1	1	1	1	1	h
0	1	0	1	1	i
0	0	0	0	0	j

FIGURE 5.5 – Fonctionnement de la table **CC-TLB**

- (a) Après initialisation, aucun composant ne contient d'information sur la ligne X.
- (b) La ligne X est accédée par le système d'exploitation pour construire la table des pages. La ligne X est copiée dans le cache L1.
- (c) Un *Miss* **TLB** se produit pour lequel l'automate *Table-Walk* trouve le **PPN** dans la ligne X. La table **CC-TLB** enregistre les informations sur la ligne X dans une nouvelle entrée (***Count*** = 1 et ***In_cache*** = 1). Le **PPN** manquant est stocké dans la **TLB**, ainsi que l'index dans la table **CC-TLB**.
- d) Un autre *Miss* **TLB** concernant la ligne X se produit. L'entrée X dans la table **CC-TLB** est mise à jour : ***Count*** = 2. Le PPN manquant est stocké dans la **TLB**, ainsi que l'index dans la table **CC-TLB**. Le nombre d'entrées pour X dans la **TLB** passe donc à 2.
- (e) La ligne X est évincée du cache L1 pour faire de la place. Le champ ***In_cache*** pour la ligne X passe à 0 dans la table **CC-TLB**. Le cache L1 n'envoie pas de `Cleanup`(X) au cache L2.
- (f) Un troisième *Miss* **TLB** concernant la ligne X se produit. La ligne X est réinstallée dans le cache L1. Le **PPN** manquant est stocké dans la **TLB**, ainsi que l'index dans la table **CC-TLB**. La TLB contient maintenant trois entrées valides pour la ligne X. Dans la table **CC-TLB**, ***Count*** = 3 / ***In_cache*** = 1.
- (g) Le cache L1 reçoit une requête de cohérence `Invalidate`(X). Il lance l'opération `Scan-TLB` afin d'invalider les trois entrées correspondant à la ligne X dans les **TLBs**. Il invalide l'entrée associée à la ligne X dans la table **CC-TLB**. Il invalide la ligne X dans le cache L1. Il envoie une requête `Cleanup`(X) au cache L2.
- (h) Après un nouveau *Miss* **TLB** concernant la ligne X, les trois composants se retrouvent dans le même état qu'à l'étape (c).
- (i) La **TLB** évince l'entrée correspondant à la ligne X pour faire de la place. Dans la table **CC-TLB**, Le champ ***Count*** de la ligne X passe 0, mais la ligne X est toujours dans le cache L1.
- (j) Le cache L1 reçoit encore une requête `Invalidate`(X). Cette fois, le cache L1 invalide la ligne X et envoie la requête `Cleanup`(X) au cache L2, mais il ne lance pas l'opération `Scan-TLB` pour les **TLBs** puisque le champ ***Count*** dans la table **CC-TLB** pour la ligne X est égal 0.

5.3.5 Sélection d'une victime dans la table CC-TLB

La taille restreinte de la table **CC-TLB** entraîne qu'une entrée valide peut être évincée pour faire de la place. Le mécanisme pour sélectionner la victime dépend de plusieurs facteurs, et vise à minimiser le coût de l'évincement :

1. s'il existe une entrée invalide, c'est toujours le premier choix.
2. sinon, on sélectionne une entrée inutilisée telle que ***Count*** = 0 et ***In_cache*** = 1. Le coût de cet évincement est nul.
3. sinon, on sélectionne une entrée inutilisée telle que ***Count*** = 0 et ***In_cache*** = 0. Le coût pour cet évincement est l'envoi d'une requête `Cleanup`(X) au cache L2.
4. sinon, on sélectionne une entrée parmi les lignes ne contenant pas de **PTD**. Le coût additionnel est celui d'une opération `Scan-TLB` pour invalider toutes les entrées des **TLBs** correspondant à la ligne sélectionnée.
5. sinon, on sélectionne une entrée parmi les lignes contenant un **PTD**. Le coût additionnel est celui d'une opération `Reset-TLB` qui invalide toutes les entrées des deux **TLBs**.

5.3.6 Analyse des chaînes longues liées à la table CC-TLB

La table **CC-TLB** n'augmente pas le chemin critique de cache L1.

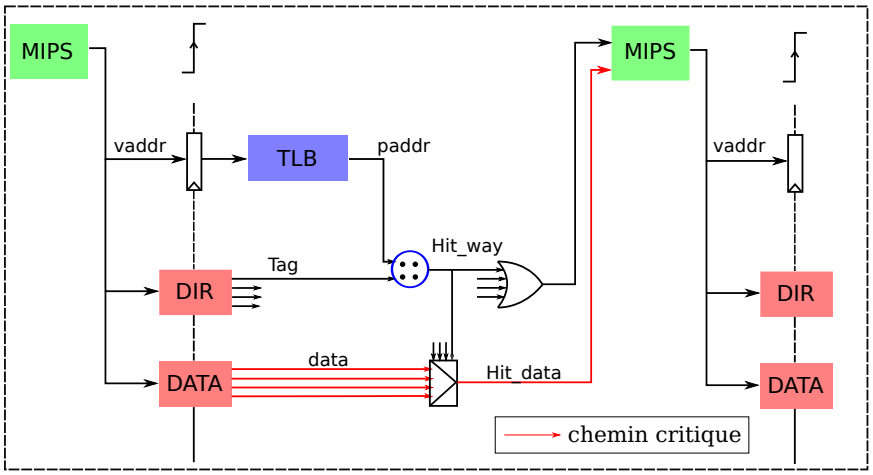


FIGURE 5.6 – Chemin critique original dans le cache L1

Dans l'implémentation VLSI de l'architecture **TSAR**, l'analyse systématique des chaînes longues montre que le chemin critique du cache L1 correspond au scénario suivant, présenté dans la figure 5.6 : le contrôleur de cache L1 lit le cache de données pour répondre à une requête du processeur, et doit renvoyer la donnée lue et la condition *Hit/Miss* avant la fin du cycle. Le temps d'accès au banc mémoire contenant les données étant très grand, le contrôleur de cache L1 a largement assez de temps pour accéder à la **TLB** en parallèle.

Dans l'architecture proposée, la table **CC-TLB**, comme la **TLB** est accédée par le contrôleur de cache L1 en parallèle avec le cache donnée, parce que le *set* pour la table

CC-TLB appartient au champ *Page Offset*. Cette table associative a une structure très semblable à celle de la **TLB** (figure 5.7). Son introduction ne devrait donc pas introduire une nouvelle chaîne longue dans le cache L1 (figure 5.8).

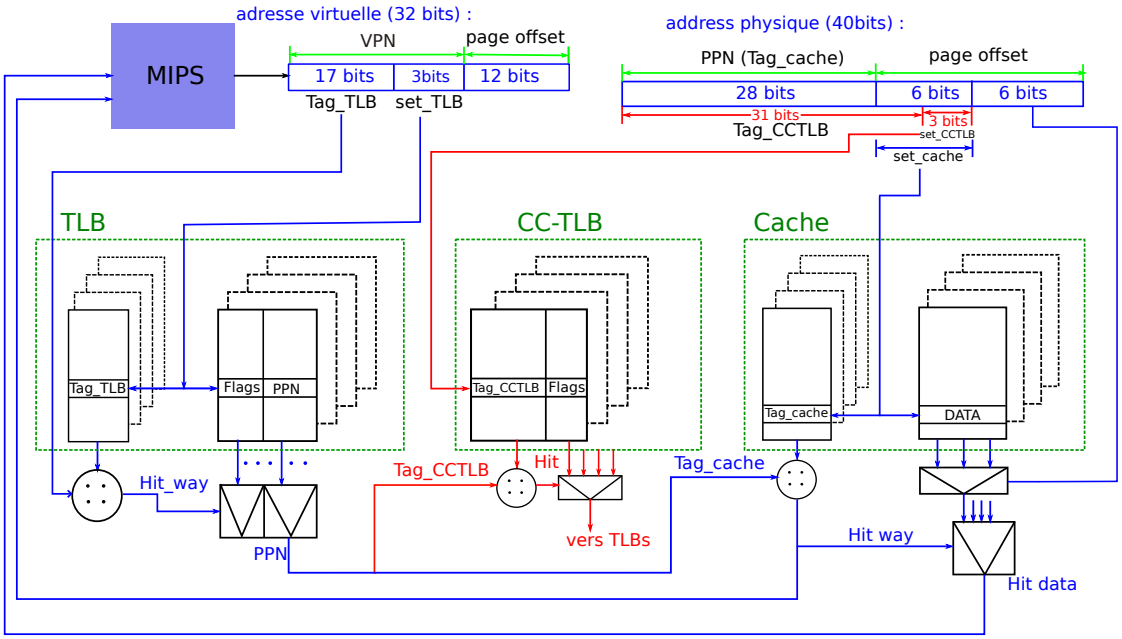


FIGURE 5.7 – Impact de la table **CC-TLB** sur le chemin critique

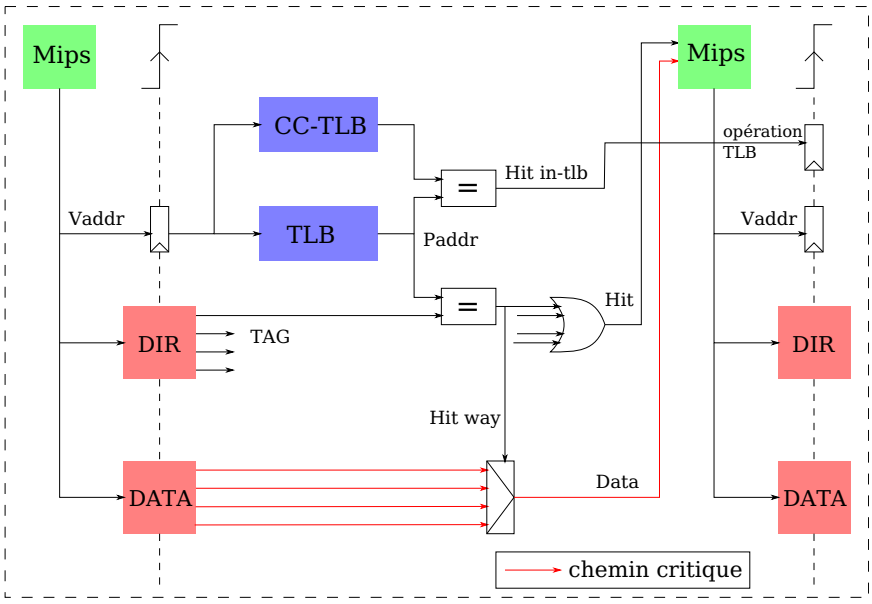


FIGURE 5.8 – Chemin critique avec le **CC-TLB** dans le cache L1

5.4 Conclusion

Dans ce chapitre, nous proposons une solution permettant d'améliorer le mécanisme matériel garantissant la cohérence des **TLBs**. Cette proposition réduit fortement le

nombre d'opérations `Scan-TLB` et `Reset-TLB` inutiles causées par les invalidations spontanées dans le cache L1. Ceci améliore le taux de *Hit* sur les **TLBs**, et réduit d'autant la consommation énergétique liée au traitement des *Miss* **TLBs**.

Le cœur de la méthode proposée consiste à relâcher la contrainte d'inclusivité des deux **TLBs** dans le cache L1, en ne stockant plus les informations nécessaires à la cohérence (champs ***nline*** et bits ***is_ppn*** / ***is_ptn***) dans le répertoire du cache, mais dans une table associative séparée appelée **CC-TLB**.

De plus, la table **CC-TLB** augmente indirectement la capacité du cache L1 : puisqu'elle permet de ne plus répliquer dans le cache L1 les informations stockées dans les **TLBs**, la table **CC-TLB** peut libérer jusqu'à 64 cases (au maximum) dans le cache L1.

On peut souligner que cette architecture passe à l'échelle, puisque la table **CC-TLB** possède une taille fixe, indépendante de la capacité du cache L1. Dans le cas du cache L1 de l'architecture **TSAR**, qui possède une capacité de 16 Koctets, la solution proposée réduit le coût matériel (mesuré en nombre de bits mémorisés) par rapport à la solution actuelle : 2 624 bits au lieu de 3 584 bits.

Enfin, l'architecture proposée n'introduit pas un nouveau chemin critique dans le contrôleur de cache L1.

Cette proposition a été implémentée dans le prototype virtuel SystemC *Cycle-Accurate* du cache L1 de l'architecture **TSAR**, pour évaluer quantitativement les gains en performances, et les résultats sont présentés dans le chapitre 8.

Chapitre 6

Micro-cache

Dans ce chapitre nous présentons deux techniques visant à réduire la consommation des caches d'instructions de premier niveau, dont nous montrons qu'elle représente plus de 34% de la consommation totale.

6.1 Analyse détaillée de la consommation

La consommation énergétique d'un cluster de **TSAR** a été évaluée par le laboratoire CEA-LETI dans le cadre du projet SHARP, en utilisant l'outil Spyglass d'Atrenta, qui a permis d'analyser la réalisation VLSI du cluster TSAR. Cet outil réalise une synthèse avec prise en compte de la topologie pour estimer la longueur des fils. Les caractéristiques énergétiques des mémoires sont également fournies à l'outil, pour qu'il évalue la consommation des blocs de **RAM**.

La technologie de fabrication utilisée est le procédé **CMOS** 28nm *Low Power bulk* de ST Microelectronics. La caractérisation est réalisée avec une alimentation de 1V, une température de 25 degrés, et un procès typique. La consommation donnée par Spyglass est proche de la consommation réelle (10% de marge d'erreur environ), d'autant que la consommation des mémoires est très précise (directement extraite des bibliothèques du fondeur) et celles-ci représentent une partie très importante de la consommation totale.

La figure 6.1 montre le pourcentage de consommation des différents types de composants d'un cluster **TSAR**. Nous voyons que les quatre cœurs (incluant le cache L1) consomment presque 80% de l'énergie totale du cluster. La consommation du cache instruction, présentée dans la figure 6.2, représente 43% de la consommation d'un coeur (23% pour le répertoire et 20% pour les données). Par conséquent la consommation des caches L1 instructions représente plus de 34% de la consommation totale du cluster. Ceci est lié au fait que ces caches sont accèdes une fois par cycle pour chaque coeur actif du cluster.

6.2 Principe du Micro-Cache

Pour réduire la consommation liée aux caches L1, nous proposons d'introduire un nouveau niveau dans la hiérarchie de cache existante, en exploitant la localité spatiale des accès instructions. Nous avons analysé deux mécanismes : Le premier mécanisme

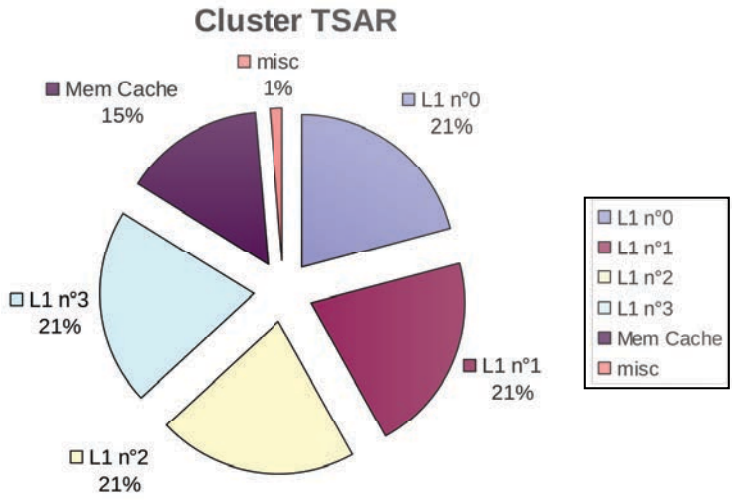


FIGURE 6.1 – Répartition de la consommation du cluster par bloc sur Dhrystone

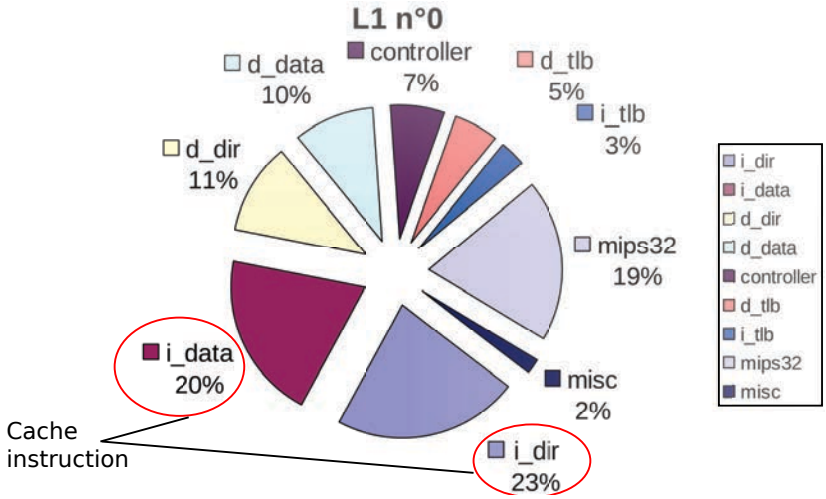


FIGURE 6.2 – Répartition de la consommation du cache L1 par bloc sur Dhrystone

vise à réduire à la fois la consommation de la partie ***répertoire*** et de la partie ***données*** du cache, mais impose des contraintes sévères sur les blocs de **RAM** utilisés. Le second mécanisme est moins ambitieux, puisqu'il ne vise qu'à réduire la consommation du répertoire, mais il ne fait pas d'hypothèse sur les blocs de **RAM**.

6.2.1 Micro-Cache complet

Le **Micro-Cache** possède une capacité d'une seule ligne de cache, et permet en principe d'éviter l'accès au cache L1 dans le cas - fréquent - où deux accès consécutifs appartiennent à la même ligne de cache. Le **Micro-Cache** contient la dernière ligne de cache instructions accédée par le processeur. En cas de *Hit* sur le**Micro-Cache**, le contrôleur du cache L1 accède au **Micro-Cache** afin de répondre au processeur, sans accéder au cache instructions.

Évidemment, le **Micro-Cache** est concerné par le protocole de cohérence. Lorsque le cache L1 reçoit une requête de cohérence pour invalider ou mettre à jour une ligne

de cache, le **Micro-Cache** doit être invalidé quand il contient une copie de la ligne concernée. Le **Micro-Cache** respecte la règle d'inclusivité : si une ligne de cache est présente dans le **Micro-Cache**, elle est forcément présente dans le cache instruction. En conséquence, lorsque le processeur demande une nouvelle instruction au cache L1, il n'existe que trois scénarios :

- *Hit* **Micro-Cache** et *Hit* cache instruction : le contrôleur du cache L1 accède au **Micro-Cache** et n'accède pas au cache L1, pour réduire la consommation.
- *Miss* **Micro-Cache** et *Hit* cache instruction : le contrôleur du cache L1 doit accéder au cache instruction, pour (i) répondre au processeur, et (ii) mettre à jour le **Micro-Cache**.
- *Miss* **Micro-Cache** et *Miss* cache instruction : le contrôleur du cache L1 envoie une requête au cache L2, le processeur est gelé pendant tout le temps de traitement du *Miss*, le cache L1 et le **Micro-Cache** sont mis à jour quand la ligne manquante est renvoyée par le cache L2.

Dans les deux premiers scénarios, le processeur demande une nouvelle instruction à la fin du cycle X, et le contrôleur du cache L1 doit répondre au processeur dans le cycle X+1. Pour éviter que le processeur soit gelé pendant un cycle en cas de *Miss* sur le **Micro-Cache**, le contrôleur du cache L1 a besoin d'un mécanisme de prédiction lui permettant de décider assez tôt s'il doit accéder au **Micro-Cache** ou au cache L1.

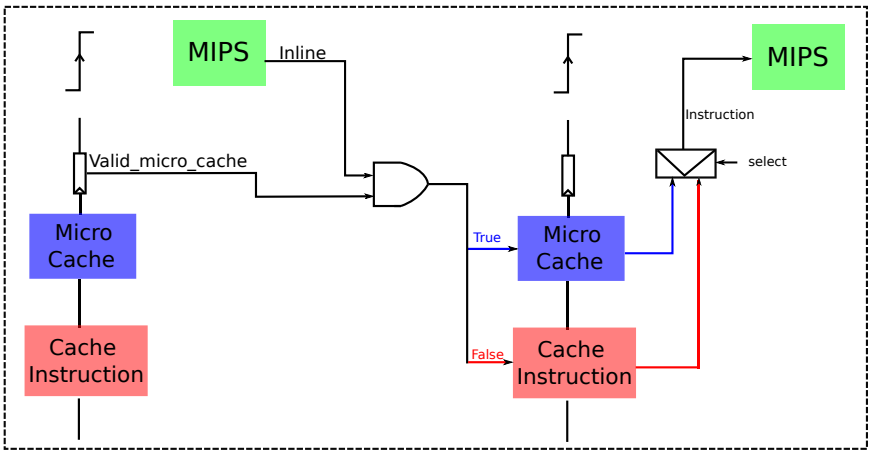


FIGURE 6.3 – Principe de micro-cache

Pour réaliser la prédiction du *Miss* **Micro-Cache**, on introduit un nouveau signal nommé ***inline*** (1 bit) sur l'interface entre le processeur et le cache L1. Ce signal est produit par le processeur, et envoyé au cache L1. Si ce nouveau signal est égal à 1, deux conditions sont assurées : (i) l'adresse demandée est consécutive (l'adresse actuelle = l'adresse précédente + 4) ; (ii) l'adresse virtuelle demandée par le processeur est dans la même ligne de cache que la requête précédente, ce qui impose au processeur d'effectuer une comparaison systématique entre les adresses virtuelles de deux requêtes consécutives. On utilise le fait que si deux adresses virtuelles sont dans la même ligne de cache, les deux adresses physiques sont également dans la même ligne. En cas de branchement le signal ***inline*** est toujours égal 0 même si l'adresse de destination se trouve dans la même ligne de cache que l'instruction précédente. Le comparateur utilisé pour comparer les adresses virtuelles complètes coûte plusieurs couches d'opérateurs

Booléen en matériel. Pour que le **Micro-Cache** n'augmente pas le chemin critique, on ne compare que les quatre bits de poids faibles et lorsque le processeur exécute deux instructions consécutives.

Lorsque le cache L1 reçoit une requête d'instruction, il teste le bit ***inline*** et le bit ***valid*** du **Micro-Cache**. Si les deux bits sont égaux à 1, le contrôleur du cache L1 accède au **Micro-Cache**. Sinon le contrôleur accède au cache L1 (voir figure 6.3).

Avec le mécanisme de prédiction décrit ci-dessus, ni le taux de *Miss*, ni le coût du *Miss* ne sont modifiés, puisque les chronogrammes associés aux deux scénarios *Hit* et *Miss* ne sont pas modifiés. Le nombre de cycles nécessaires pour exécuter une application n'est donc pas être impacté par l'introduction du **Micro-Cache**.

La réduction de la consommation est obtenue pour les requêtes qui font à la fois *Hit* sur le cache L1 et *Hit* sur le **Micro-Cache**. Le gain sera donc proportionnel au produit du taux de *Hit* sur le cache L1 (typiquement de l'ordre de 95% à 98%) par le taux de *Hit* sur le **Micro-Cache**. On peut donc espérer diviser par un facteur 4 la consommation du cache instruction, si le taux de *Hit* sur le **Micro-Cache** est de 80%.

Il faut cependant noter que le mécanisme proposé ci-dessus impose une forte contrainte sur le matériel : En cas de *Miss* sur le **Micro-Cache**, on suppose que la RAM utilisée pour le cache L1 permet de lire une ligne complète (64 octets) en un seul cycle, pour permettre le rechargement du **Micro-Cache** en un cycle dans le deuxième scénario ci-dessus.

6.2.2 Micro-Cache simplifié

Si la contrainte de lecture de 64 octets en un cycle n'est pas respectée par les blocs de RAM disponibles, on peut simplifier le mécanisme. Si la RAM ne peut pas fournir plus d'une instruction par cycle (4 octets), il devient impossible de réduire la consommation de la partie ***données***, mais on peut encore réduire la consommation de la partie ***répertoire***, qui représente plus de la moitié de la consommation totale du cache.

Avec ce second mécanisme, on ne stocke plus de données dans le **Micro-Cache**, et on se contente d'enregistrer dans le **Micro-Cache** le numéro de *set* (6 bits), et le numéro de *way* identifiant la ligne de cache accédée pour répondre à la dernière requête (N) du processeur. Si le bit ***inline*** est valide lors de la requête (N+1), il n'est pas utile d'accéder au répertoire du cache, puisqu'on peut directement utiliser les valeurs *set* et *way* enregistrées dans le **Micro-Cache**. Il reste donc possible de diviser par 4 le nombre d'accès au répertoire du cache instruction.

6.3 Conclusion

En conclusion, ce chapitre propose une méthode permettant de réduire significativement la consommation des caches instructions de premier niveau, qui représente 34% de la consommation totale.

Si les mémoires utilisées permettent de lire une ligne de cache complète (64 octets) en un seul cycle, le mécanisme proposé doit permettre de réduire très fortement le nombre des accès au cache d'instructions, et donc de réduire la consommation, sans dégradation des performances.

Si les mémoires utilisées ne le permettent pas, la version simplifiée du mécanisme proposé permet en principe de réduire très fortement le nombre d'accès au répertoire du cache d'instructions, ce qui permet encore d'espérer une réduction significative de la consommation, puisque les accès au répertoire représentent plus de la moitié de la consommation du cache.

Chapitre 7

Résultats Expérimentaux concernant le protocole RWT

Les expérimentations présentées dans ce chapitre visent à évaluer le protocole **RWT** proposé dans le chapitre 3, en le comparant aux deux protocoles **DHCCP** et **HMESI**, en termes de performance et de consommation énergétique.

7.1 Définition des Métriques

La performance est mesurée par le temps d'exécution (mesuré en nombre de cycles), pour différentes applications parallèles *multi-threads* décrites ci-dessous. Elle peut être mesurée facilement sur le prototype virtuel de l'architecture **TSAR**, qui est précis au cycle. On s'intéresse particulièrement au *speedup* obtenu lorsque le nombre de coeurs augmente, pour chacun des trois protocoles, ce qui permet d'évaluer la scalabilité des protocoles.

La consommation énergétique est plus difficile à évaluer sur un prototype virtuel SystemC. En **CMOS**, la consommation dynamique résulte de la charge et de la décharge des capacités, liées au mouvement des données. Nous utilisons donc comme métrique, appelée dans ce chapitre ***coût énergétique*** le nombre total de flits transférés sur l'un quelconque des cinq réseaux permettant la communication entre les caches L1 et L2 , pondéré par la distance entre le cluster source et le cluster destination. Les cinq réseaux considérés sont les 2 réseaux supportant les transactions directes (Commandes/Réponses) et les 3 réseaux supportant les transactions de cohérence. Un flit est un mot de 32 bits, puisque le réseau **DSPIN** (le réseau global dans l'architecture TSAR) [22] transporte des mots de 32 bits.

7.2 Système d'exploitation *Giet-VM*

Nous utilisons un OS entièrement statique nommé ***Giet-VM*** [23] permettant de contrôler explicitement le placement des différents structures de données logiciels sur les bancs de mémoire physique, de façon à minimiser les impacts négatifs de scalabilité introduits par le système d'exploitation. Le ***Giet-VM*** fournit les mécanismes suivants, pour renforcer la localité des accès et minimiser la contention :

- la totalité du code ***noyau*** et du code ***utilisateur*** est répliquée dans tous les clusters. Par conséquent, en cas de *Miss* instruction, le cache L1 cherche toujours la ligne manquante dans son cache L2 local.
- pour la même raison, la table des pages de tous les segments virtuels est répliquée dans tous les clusters, ce qui réduit fortement le coût des *Miss* **TLB**.
- Les piles d'exécution pour les différentes tâches d'une application parallèle sont distribuées dans chaque cluster. Le segment de pile pour un *thread* se trouve toujours dans le même cluster que le coeur qui exécute ce *thread*.
- Les ***tas*** utilisés pour l'allocation dynamique de mémoire, par le noyau comme par les applications, sont distribués sur tous les clusters, ce qui permet de renforcer la localité des accès mémoire.
- Enfin le ***Giet-VM*** fournit des barrières de synchronisation hiérarchiques et distribuées permettant de réduire la contention lorsque le nombre de participants est grand.
- Finalement, seul le segment des données globales de l'application n'est stocké que dans un seul cluster. Ces données globales sont souvent partagées, et il est difficile de les répliquer sans implémenter un mécanisme logiciel de cohérence entre les différentes copies.

Avec ces mécanismes, le système d'exploitation ***Giet-VM*** permet de minimiser l'impact négatif de choix non optimaux du système d'exploitation concernant le placement des données, ce qui permet de mieux évaluer l'impact spécifique du protocole de cohérence matériel sur la scalabilité.

7.3 Choix des *benchmarks*

Les *benchmarks* sélectionnés pour évaluer les protocoles de cohérence des caches sont des applications parallèles *multi-threads*. En règle générale, on place un seul *thread* par coeur, et il n'y a pas de migration. Toutes ces applications ont été programmées pour qu'on puisse faire varier facilement le nombre de *threads*.

Ces applications peuvent être classifiées en trois types, suivant leur utilisation des données partagées :

- données non partagées : les *threads* travaillent indépendamment les unes des autres. Il y a donc peu de données partagées entre les différents caches L1. Avec ce type de programmes, le protocole de cohérence des caches n'impacte pas le temps d'exécution, car presque toutes les lignes de cache sont lues ou écrites en mode privé.
- données partagées de type *Read Only* : les données partagées sont principalement accédée en lecture par les *threads* ; Le cache L2 reçoit principalement des requêtes de lecture sur les lignes dans l'état `SHARED`. Avec le protocole **DHCCP**, il n'existe pas de changement d'état pour une requête de lecture. Dans le protocole **RWT** il existe un passage d'état de `NC` à `C`, et un changement d'état de `EXCLUSIVE` à `SHARED` dans le protocole **HMESI**.
- données partagées de type de *Multi Read Single Write* : les données partagées entre les *threads* sont modifiées. Les protocoles **DHCCP** et **RWT** envoient la requête

d'écriture directement au cache L2 par le réseau direct ; le protocole **HMESI** envoie la requête `GetM` afin de changer l'état de la ligne de `SHARED` à `EXCLUSIVE`. C'est pour ce type d'applications que les différences entre les trois types de protocoles apparaissent le plus clairement.

Nous avons porté six *benchmarks* sur le ***Giet-VM*** :

- **Histogram** venant de *Benchmark Phoenix-2[24]*. Cette application génère l'histogramme des pixels dans les canaux rouge, vert et bleu pour une image bitmap. Cette image est découpée et distribuée dans chaque cluster afin que chaque *thread* traite localement une partie de l'image, il n'y a donc pas de partage de données entre les *threads*. Cette application appartient donc au premier type.

- **FFT** venant de *benchmark Splash-2[25]*. Cette application réalise la transformation de Fourier rapide, et comporte six étapes [26]. Elle est optimisée pour minimiser la communication *Inter-threads*. Les écritures sont locales, mais chaque *thread* doit aller lire ses données dans tous les clusters et les lectures sont donc distantes. L'application **FFT** appartient donc au deuxième type.

- **LU** venant de *benchmark Splash-2*. Cette application factorise une matrice creuse en une matrice triangulaire inférieure et une matrice triangulaire supérieure. La matrice creuse *n x n* est divisée en *N x N* tableaux de *B x B* blocs (n = N $\times$ B). Chaque *thread* traite un ensemble de blocs, qui sont stockés et modifiés localement. Cette application appartient également au second type [27].

- **Kmeans** venant de *benchmark Phoenix-2*. Le partitionnement en k-moyennes (ou *k-means* en anglais) est une méthode de partitionnement de données et un problème d'optimisation combinatoire. Étant donnés des points et un entier k, le problème est de diviser les points en k sous-ensembles, souvent appelés clusters, de façon à minimiser une certaine fonction. On considère la distance d'un point à la moyenne des points de son cluster ; la fonction à minimiser est la somme de ces distances [28]. Cette application est aussi du second type.

- **Convol** est une application parallèle développée au LIP6 par Alain Greiner. Elle réalise un filtrage d'images médicales 1024*1024 pixels, au moyen d'un noyau de convolution fourni par Philips, de dimension 35*201. L'image est distribuée dans chaque cluster. Comme le filtre est séparable, il peut être réalisé en deux phases : un filtrage horizontal sur les lignes, suivi par un filtrage vertical sur les colonnes. Le découpage en *threads* est réalisé de telle sorte que la quasi-totalité des lectures de données partagées est locale, alors que la quasi-totalité des écritures de données partagées est distante. L'application Convol appartient donc au troisième type.

- **Radix** venant de *benchmark Splash-2*. Le kernel du tri radix est basé sur la méthode décrite dans [29]. L'algorithme est itératif, effectuant une itération pour chaque chiffre de radix r des clés. Dans chaque itération, un processeur génère un histogramme local. Entre deux itérations, les processeurs doivent communiquer entre eux pour échanger des clés, ce qui nécessite des écritures distantes vers tous les autres processeurs [30]. En conséquence, l'application Radix appartient au troisième type.

benchmark	data entrée
Histogram	25 MB image (3408 x 2556)
FFT	2^{18}
Radix	262,144 keys (defaut)
LU	512 x 512 éléments
Kmeans	10,000 points
Convol	1024 x 1024 image

FIGURE 7.1 – Paramètres des *benchmarks* pour évaluer les protocoles de cohérence

Pour favoriser la localité des accès et minimiser les contentions, nous avons distribué les données dans tous les clusters en utilisant la fonction d'allocation distante du ***Giet-VM***. À titre d'exemple, le code original du *benchmark* FFT7.1 est présenté ci-dessous. On constate que les données partagées sont stockées dans des tableaux alloués par la fonction *malloc*. Par conséquent, ces données sont toutes placées dans le même cluster, dans un unique segment de mémoire physique. Ce placement crée une forte contention pour l'accès au cache L2 du cluster contenant ces données.

Code 7.1 – Code original du benchmark FFT

```
double * x;      /* x is the original time-domain data     */
double * trans;  /* trans is used as scratch space         */
double * umain;  /* umain is roots of unity for 1D FFTs    */
double * umain2; /* umain2 is entire roots of unity matrix */

x      = (double *) malloc(2 * (N + rootN * pad_length) * sizeof(double) + PAGE_SIZE);

trans  = (double *) malloc(2 * (N + rootN * pad_length) * sizeof(double) + PAGE_SIZE);

umain  = (double *) malloc(2 * rootN * sizeof(double));

umain2 = (double *) malloc(2 * (N + rootN * pad_length) * sizeof(double) + PAGE_SIZE);
```

La fonction ***remote_malloc(unsigned int size, unsigned int x, unsigned int y)*** du ***Giet-VM*** permet d'allouer un segment de mémoire de *size* octets dans le cluster de coordonnées (x,y). Nous avons donc découpé chaque tableau en autant de sous-tableaux que le nombre de clusters dans l'architecture, de façon a distribuer ces données dans tous les clusters.

Le code 7.2 ci-dessous montre la modification d'allocation de mémoire pour le *benchmark* **FFT**.

Code 7.2 – Code modifié du benchmark FFT

```
double * x[NB_CLUSTERS];       /* x is the original time-domain data     */
double * trans[NB_CLUSTERS];   /* trans is used as scratch space         */
double * umain[NB_CLUSTERS];   /* umain is roots of unity for 1D FFTs    */
double * umain2[NB_CLUSTERS];  /* umain2 is entire roots of unity matrix */

for (i = 0; i < NB_CLUSTERS; i++)
{
    y_size = i % Y_SIZE;
    x_size = i / Y_SIZE;

    x[i]      = (double *) remote_malloc(2 * (N/NB_CLUSTERS + (rootN/NB_CLUSTERS)
                                       * pad_length) * sizeof(double), x_size, y_size);

    trans[i]  = (double *) remote_malloc(2 * (N/NB_CLUSTERS + (rootN/NB_CLUSTERS)
                                       * pad_length) * sizeof(double), x_size, y_size);

    umain[i]  = (double *) remote_malloc(2 * ((rootN/NB_CLUSTERS) * pad_length)
```

```
                                        * sizeof(double), x_size, y_size);

    umain2[i] = (double *) remote_malloc(2 * (N/NB_CLUSTERS + (rootN/NB_CLUSTERS)
                                        * pad_length) * sizeof(double), x_size, y_size);
}
```

7.4 Architecture générique

Les trois protocoles de cohérence des caches sont implémentés dans les composants cache L1 et cache L2 du prototype virtuel, précis au cycle de l'architecture **TSAR**, qui utilise la plateforme de prototypage *SoCLib*[31]. La figure 7.2 montre une plateforme contenant 16 clusters. Chaque cluster contient 4 cœurs avec leurs cache L1, un cache L2 distribué, et deux périphériques système : un contrôleur **DMA** exclusivement utilisé par le système d'exploitation, et un contrôleur d'interruptions[32].

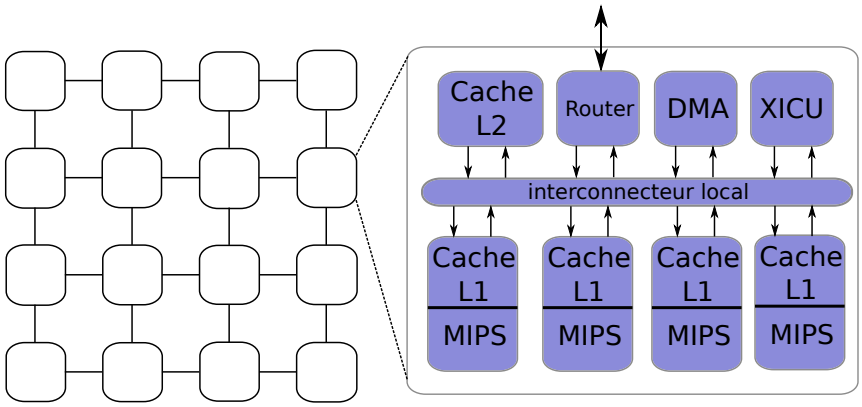


FIGURE 7.2 – Synthèse générale de plateforme **TSAR** pour évaluer le protocole RWT

Mesh Size	jusqu'à 16 x 8
L1 Cache Sets (I & D)	64
L1 Cache Ways (I & D)	4
L1 Cache Words (I & D)	16
L2 Cache Sets	256
L2 Cache Ways	16
L2 Cache Words	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8

FIGURE 7.3 – Paramètres de plateforme

Le prototype virtuel est générique, puisqu'on peut faire varier les dimensions X_SIZE et Y_SIZE de la grille, et le nombre de coeurs par cluster. Pour analyser la scalabilité des protocoles de cohérence, les 6 *benchmarks* ont été exécutés sur des architectures possédant 1, 2, 4, 8, 16, 32, 64, 128, 256 et 512 cœurs séparément pour les protocoles **DHCCP**, **RWT** et **HMESI**. Les caractéristiques de chaque composant sont dans le tableau 7.3.

7.5 Compteurs d'instrumentation

Pour évaluer la qualité des protocoles proposés en terme de performance et consommation énergétique, nous avons introduit dans le modèle du cache L2 des compteurs d'instrumentation fournissant :

- le temps d'exécution en nombre de cycles.
- le nombre de flits correspondant aux transactions de lecture, locales ou distantes.
- le nombre de flits correspondant aux transactions d'écriture, locales ou distantes.
- le nombre de requêtes `GetM` dans le protocole **HMESI**, locales ou distantes.
- le nombre de flits pour chacun des quatre types de transactions de cohérence.

Le coût énergétique est calculé de la façon suivante :

- pour toutes les transactions entre L1 et L2 dans le même cluster, le coût est égal au nombre de flits puisqu'il y a un seul accès au *crossbar* local : Coût local = Nflits x 1
- pour toutes les requêtes entre clusters distincts, le coût inclut deux accès au *crossbar* (un accès au *crossbar* du cluster source, et un autre accès au *crossbar* du cluster destination) plus la traversée d'un nombre de routeurs dépendant de la distance de Manhattan entre les clusters source et destination : Coût distant = Nflits x (Nhops + 2)

7.6 Analyse des Résultats

7.6.1 Résultats Histogram

Cette application n'a pratiquement pas de données partagées entre les *threads*. La figure 7.4 montre le *speedup* pour les trois protocoles jusqu'à 512 coeurs. Les *speedups* pour les 3 protocoles sont très proches du *speedup* idéal (temps d'exécution divisé par N, pour N coeurs).

La consommation est présentée dans la figure 7.5. Le protocole protocole **DHCCP** se singularise par un fort trafic d'écritures. Ce coût est réduit à zéro pour les protocoles **RWT** et **HMESI**, qui ont des comportements identiques.

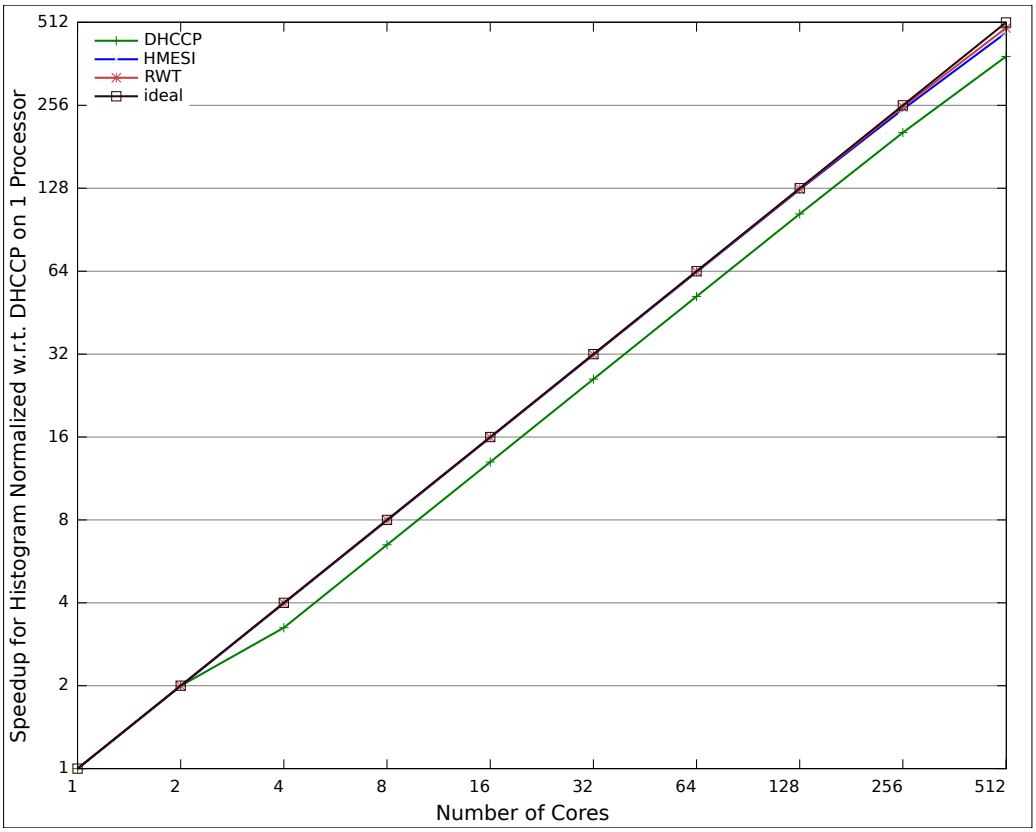


FIGURE 7.4 – Speedup pour Histogram

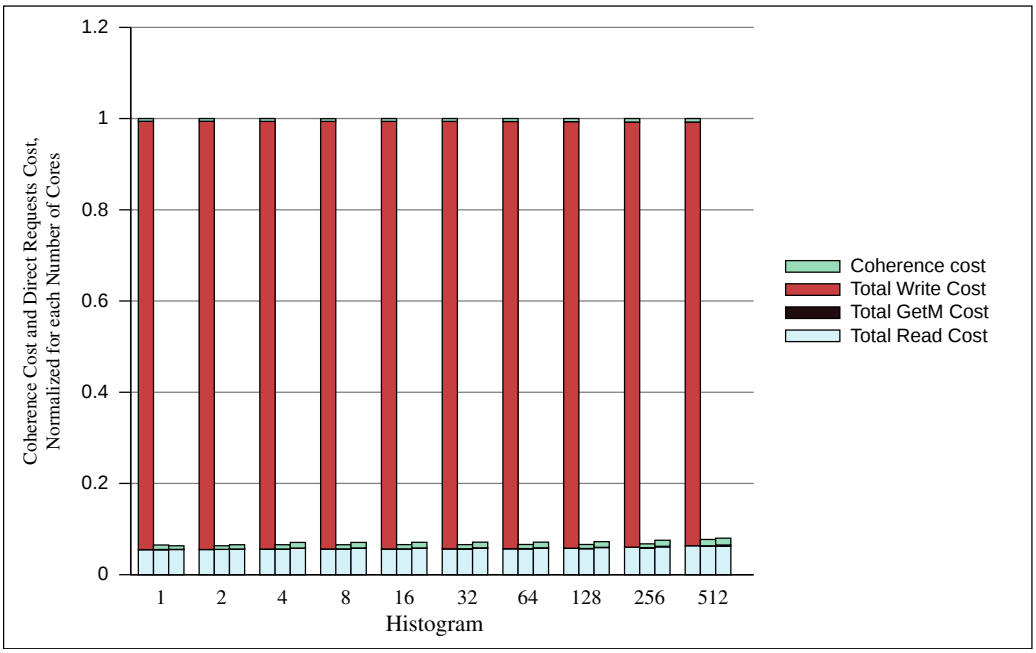


FIGURE 7.5 – Coûts de lecture, écriture, et cohérence de **Histogram** pour les trois protocoles, **DHCCP** (colonne à gauche), **RWT** (colonne à milieu) et **HMESI** (colonne à droite) normalisé par le coût total de **DHCCP**

7.6.2 Résultat FFT, LU et Kmeans

Dans les applications **FFT**, **LU** et **Kmeans**, il n'y a pas d'écriture sur les données partagées. La figure 7.6 montre les *speedups* pour ces trois *benchmarks*. On peut constater que les trois protocoles ont pratiquement les mêmes *speedup*. L'application **FFT** montre une bonne scalabilité. **LU** n'est pas scalable au-delà de 32 cœurs. **Kmeans** n'est pas scalable au-delà de 64 cœurs. Mais nous n'avons pas de différences de performance significatives entre les trois protocoles.

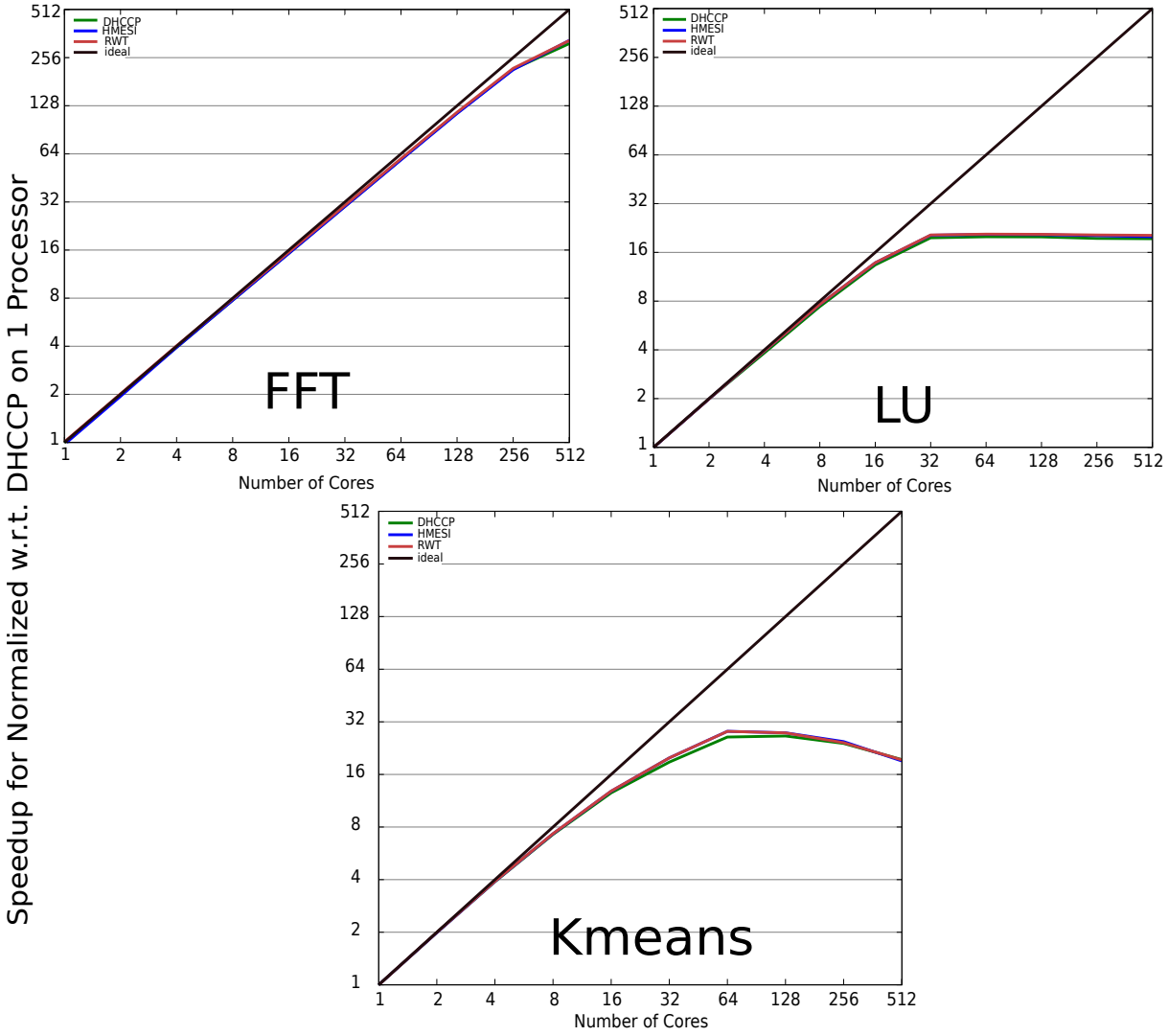


FIGURE 7.6 – Speedup pour FFT, LU et Kmeans

Pour ce qui concerne la consommation, on constate ici encore que les protocoles **RWT** et **HMESI** ont des consommations très voisines, bien inférieures à la consommation du protocole **DHCCP**, car presque toutes les écritures sont effectuées sur des données privées, ce qui réduit énormément le trafic lié aux écritures dans **RWT** et **HMESI**.

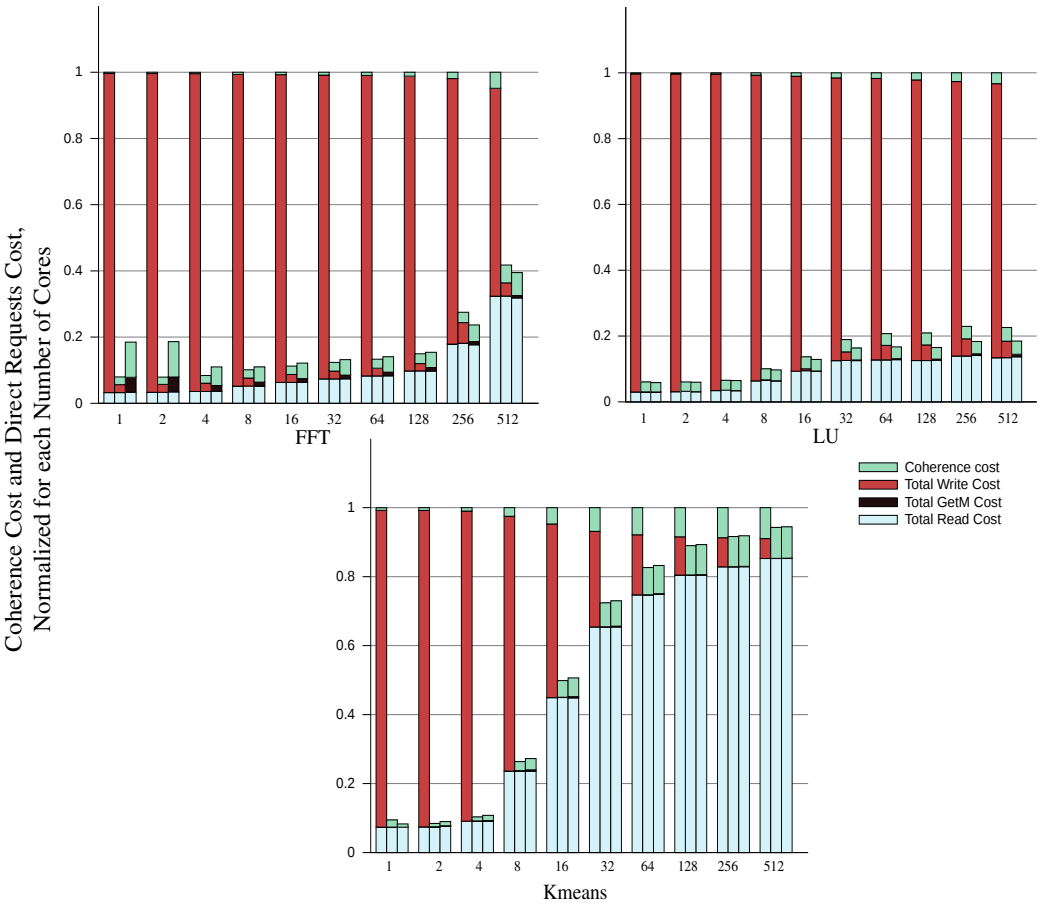


FIGURE 7.7 – Coûts de lecture, écriture, et cohérence de **FFT**, **LU** et **Kmeans** entre les trois protocoles, **DHCCP** (colonne à gauche), **RWT** (colonne à milieu) et **HMESI** (colonne à droite) normalisés par le coût total de **DHCCP**

7.6.3 Résultat Convol et Radix

Dans ces deux applications, on a un nombre important d'écritures sur des données partagées, et on s'attend donc à des différences significatives entre les 3 protocoles.

Les *speedups* sur les applications **Convol** et **Radix** sont montrés dans la figure 7.8. Parmi les trois protocoles, le protocole **HMESI** est le moins scalable, puisqu'il obtient le pire *speedup*. Par ailleurs, les performances du protocole **RWT** sont comparables, voire légèrement meilleures que celles du protocole **DHCCP**.

Les résultats de consommation sont présentés dans la figure 7.9. Le protocole **RWT** supprime la consommation liée aux écritures dans les lignes privées, mais il reste la consommation liée aux écritures dans les lignes partagées. Néanmoins, dans tous les cas, la consommation de **RWT** est très nettement inférieure à celle de **DHCCP**. Pour le protocole **HMESI**, la consommation liée aux écritures sur les lignes partagées se retrouve dans la consommation liée aux requêtes `GetM`. Mais surtout, la consommation liée au trafic de cohérence augmente fortement avec le nombre de coeurs, alors que cette consommation liée au trafic de cohérence reste marginale dans **RWT**, quel que soit le nombre de coeurs. Dans tous les cas, la consommation totale de **RWT** est toujours au moins 4 fois plus faible que celle de **HMESI**, et au moins 2 fois plus faible que celle de

DHCCP.

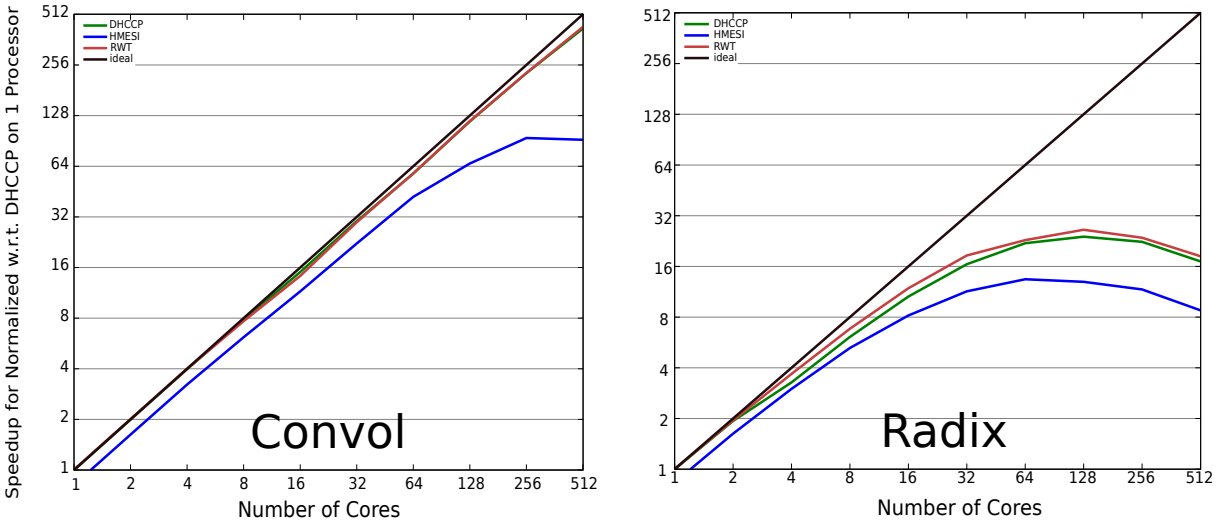


FIGURE 7.8 – Speedup pour Convol et Radix

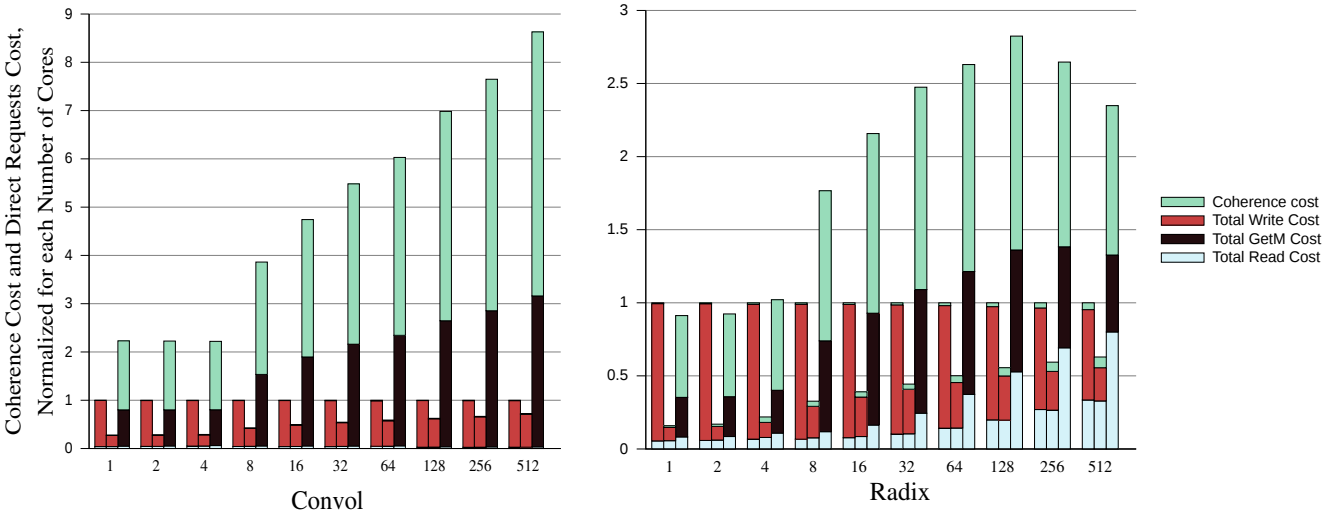


FIGURE 7.9 – Coûts de lecture, écriture, et cohérence de **Convol** et **Radix** pour les trois protocoles, **DHCCP** (colonne à gauche), **RWT** (colonne à milieu) et **HMESI** (colonne à droite)

7.7 Conclusion

Au vu des résultats ci-dessus, nous pouvons dire que, par rapport au protocole **DHCCP** initial, le protocole **RWT** réduit au moins d'un facteur 2 le trafic entre les caches L1 et les caches L2, et réduit donc dans la même proportion la consommation énergétique des 5 réseaux de l'architecture **TSAR**. Ce résultat est obtenu sans aucune dégradation des performances ni de la scalabilité. Pour ce qui concerne le protocole classique **HMESI**, bien qu'il supprime toutes les écritures dans le réseau direct, il est moins scalable que **RWT** et **DHCCP**, et il consomme beaucoup plus. Ces deux

caractéristiques négatives de **HMESI** ont une même cause, qui est l'augmentation importante du trafic de cohérence quand le nombre de coeurs augmente.

En conclusion, le protocole **RWT**, est une amélioration importante du protocole **DHCCP** pour ce qui concerne la consommation énergétique, alors que **DHCCP** était déjà meilleur que **HMESI** en termes de scalabilité, comme en termes de consommation.

Chapitre 8

Résultats Expérimentaux concernant la cohérence des TLbs et le Micro-Cache

8.1 Résultats Expérimentaux concernant la cohérence des TLbs

Les résultats expérimentaux présentés dans cette section évaluent les gains apportés par le mécanisme **CC-TLB** pour la cohérence des **TLBs** par rapport à la solution actuellement implantée dans **TSAR** (appelée méthode de Yang dans ce chapitre). Ce mécanisme **CC-TLB** a été présenté au chapitre 5.

Ce mécanisme a été introduit dans le contrôleur de cache L1 du prototype virtuel de l'architecture **TSAR** de référence, qui implémente le protocole **DHCCP**.

8.1.1 Plate-forme expérimentale

Le but du mécanisme **CC-TLB** est d'une part, de réduire le nombre de *Miss* **TLB** dans le cache L1 (afin d'augmenter la performance), et d'autre part de réduire le nombre de *Miss* dans cache L1 causés par les *Miss* **TLB** (afin de réduire la consommation énergétique).

Nous avons choisi de ne pas utiliser le même système d'exploitation que pour la comparaison des protocoles du chapitre 7. En effet, le coût des *Miss* **TLB** est très faible quand on utilise le ***Giet-VM***, car les tables des pages sont répliquées dans tous les clusters, ce qui supprime la contention et diminue la latence. Par conséquent, le taux de *Miss* **TLB** dans les applications déployées au-dessus du ***Giet-vm*** n'impacte que très faiblement les performances des applications. Nous avons donc décidé d'utiliser un système d'exploitation standard (qui ne réplique pas les tables de pages dans tous les clusters), pour cette évaluation. Nous avons choisi le système d'exploitation UNIX NetBSD, déjà porté sur l'architecture **TSAR**. Avec NetBSD, comme avec Linux, il faut systématiquement envoyer la requête de lecture au cache L2 du cluster(0, 0) afin de trouver la traduction manquante en cas de *Miss* **TLB**.

Les résultats présentés ci-dessous ont été obtenus en exécutant 5 applications sur la plate-forme. Les paramètres pour ces *benchmark* sont présentés dans la table 8.1.

La fonction ***pthread_setaffinity_np*** est ajoutée dans tous les *benchmarks* afin de

contrôler le placement des *threads* sur les différents coeurs.

Les temps de simulation avec NetBSD sont beaucoup plus longs qu'avec le ***Giet-vm***, car le temps de démarrage de NetBSD est près de 10 fois plus long que celui du ***Giet-vm***. Nous avons donc réduit le nombre de cœurs dans chaque cluster, (un coeur au lieu de quatre) et nous nous sommes limités à 16 clusters (mesh de 4 x 4), comme décrit figure 8.2). Les paramètres de plate-forme sont définis dans le tableau 8.3

Benchmark	data entrée
FFT	2^{10}
Radix	262,144 keys (defaut)
LU	512 x 512 éléments
Ocean	258 x 258 éléments
Convol	1024 x 1024 image

FIGURE 8.1 – Paramètres des *benchmarks* pour évaluer le **CC-TLB**

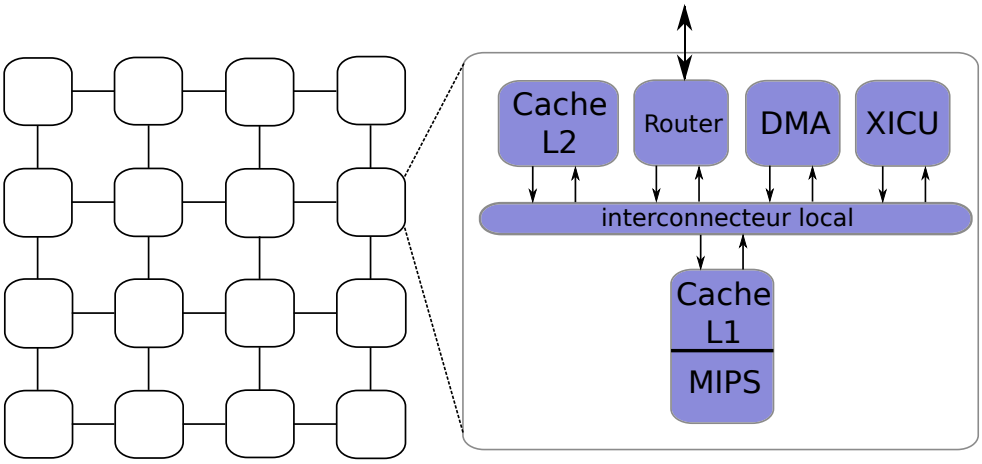


FIGURE 8.2 – Synthèse générale de plate-forme TSAR pour évaluer le **CC-TLB**

Mesh Size	jusqu'à 16 x 8
L1 Cache Sets (I & D)	64
L1 Cache Ways (I & D)	4
L1 Cache Words (I & D)	16
L2 Cache Sets	256
L2 Cache Ways	16
L2 Cache Words	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8
CC-TLB Sets (I & D)	8
CC-TLB Ways (I & D)	8

FIGURE 8.3 – Paramètres de plate-forme avec **CC-TLB**

On a introduit dans le modèle SystemC du contrôleur de cache L1, les compteurs d'instrumentation nécessaires pour mesurer :

- le nombre de *Miss* **TLB** total ;
- le nombre de *Miss* cache causé par un *Miss* **TLB** ;
- le nombre de *Scan* **TLB** causé par un évincement spontané ;
- le nombre de *Scan* **TLB** causé par une requête d'écriture ;
- le nombre de *Scan* **TLB** causé par une requête de cohérence ;
- le nombre de *Reset* **TLB** causé par un évincement spontané ;
- le nombre de *Reset* **TLB** causé par une requête d'écriture ;
- le nombre de *Reset* **TLB** causé par une requête de cohérence ;

8.1.2 Analyse des Résultats

Chaque *benchmark* a été exécuté 10 fois et nous présentons la valeur moyenne des compteurs d'instrumentations. Les résultats présentés dans les figures sont en valeurs relatives par rapport à la solution existante (appelée solution Yang).

La figure 8.4, montre les temps d'exécution pour chaque *benchmark*. Le mécanisme **CC-TLB** réduit les temps d'exécution de 5% à 20% suivant les applications.

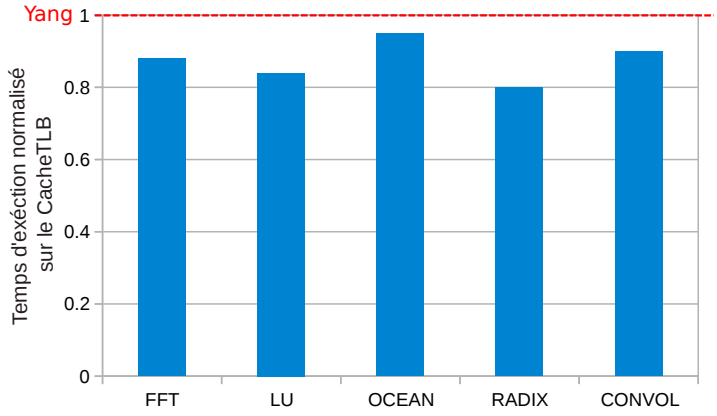


FIGURE 8.4 – Temps d'exécution

La figure 8.5 montre que le nombre total de *Miss* **TLB** est fortement réduit. Sur les *benchmarks* **LU** et **Radix**, la réduction peut éteindre jusqu'à 70%. L'amélioration des performances est donc bien causée par la réduction du nombre de *Miss* **TLB**.

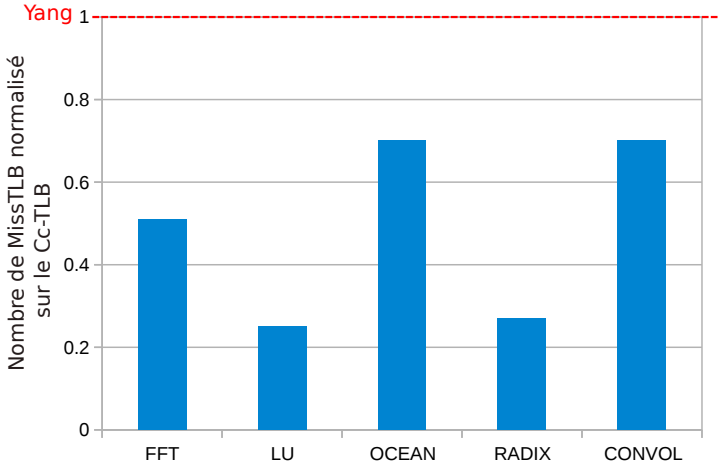


FIGURE 8.5 – Nombre de *Miss* **TLB**

Dans les figures 8.6 et 8.7 nous voyons que le nombre d'événements *Scan-TLB* et *Reset-TLB* est très fortement réduit par rapport à la solution Yang, puisque la grande majorité de ces événements (environ 90% 95% des *Scan-TLB* et 50% des *Scan-TLB*) étaient causés par les évincements spontanés dans le cache L1.

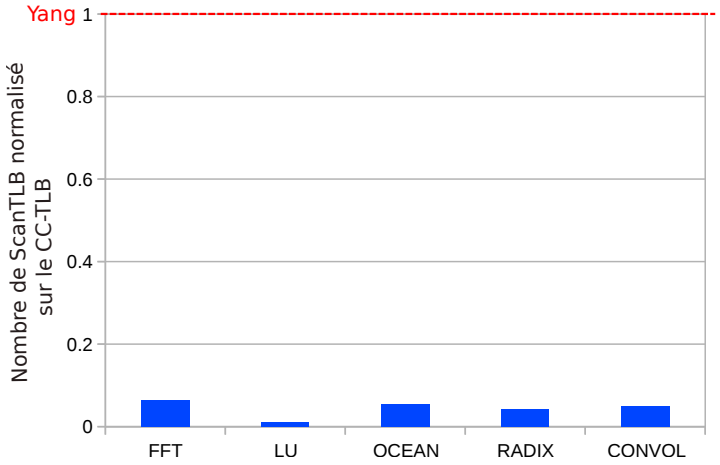


FIGURE 8.6 – Nombre de *Scan-TLB*

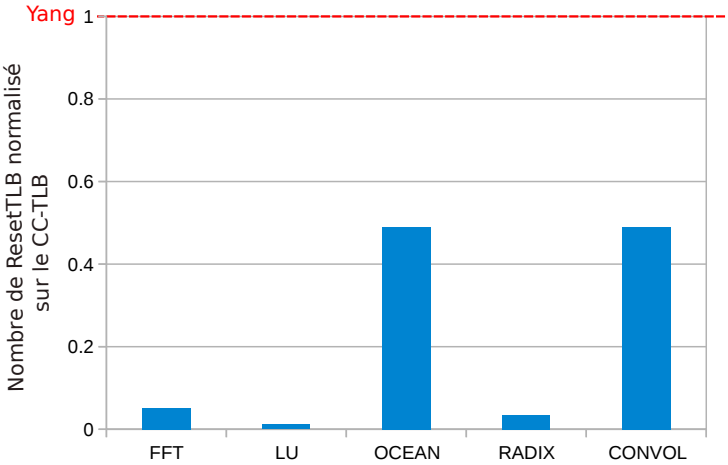


FIGURE 8.7 – Nombre de *Reset-TLB*

Pour la consommation énergétique, nous analysons (dans la figure 8.8) le nombre de *Miss* Cache causé par des *Miss* **TLB** qui ne peuvent être traités avec les informations présentes dans le cache L1 des données, et qui nécessitent donc une transaction sur le réseau. Le **CC-TLB** réduit d'environ 60% le nombre de ces événements. Ce résultat est doublement positif, (i) il y a moins de trafic sur le réseau, ce qui réduit la consommation énergétique ; (ii) on minimise la contention sur le cache L2 contenant les tables des pages.

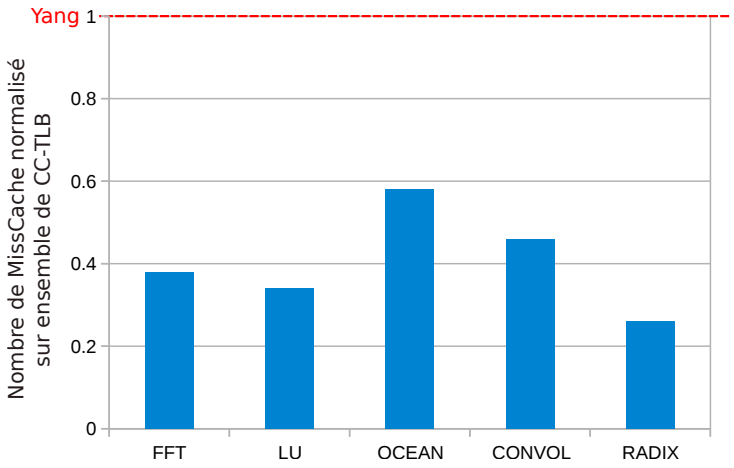


FIGURE 8.8 – Nombre de *Miss* Cache causés par un *Miss* **TLB**

8.1.3 Conclusion

La table **CC-TLB** supprime tous les `Scan-TLB` et `Scan-TLB` correspondant à l'évincement spontané d'une ligne de cache de type ***in_tlb***. Ces évincements spontanés sont des événements fréquents, puisque pratiquement chaque MISS entraîne un évincement.

En conséquence le nombre de *Miss* **TLB** est très fortement réduit pour toutes les applications : entre 30% et 70% suivant les applications. Pour un système d'exploitation standard, du type UNIX ou Linux, où les tables de pages utilisées par les **TLBs** ne sont pas répliquées, cette réduction du taux de *Miss* entraîne des gains de performance compris entre 10% et 20%, ainsi qu'une réduction significative du nombre de *Miss* sur le cache de données, qui ne peut que réduire la consommation énergétique.

Comme cela a été décrit dans le chapitre 5, ces améliorations sont obtenues sans augmentation, mais au contraire avec une légère diminution du coût matériel.

8.2 Résultats Expérimentaux concernant le Micro-Cache

Dans ce section, nous présentons les résultats des expérimentations qui ont été réalisées concernant le mécanisme **Micro-Cache**, qui vise à réduire la consommation énergétique du cache L1 des instructions.

8.2.1 Plateforme expérimentale

Les blocs de **RAM** utilisés pour le prototype VLSI de **TSAR** ayant déjà été optimisés pour réduire la consommation, ils ne permettent pas de lire plus d'une instruction par cycle. C'est d'ailleurs cette optimisation qui explique que la partie *données* du cache instructions consomme légèrement moins que la partie *répertoire*. C'est donc la version simplifiée du **Micro-Cache** qui a été implémenté dans le modèle VHDL synthétisable du contrôleur de cache L1. Plus précisément, nous avons introduit le **Micro-Cache** dans la version de référence du cache L1, qui implémente le protocole **DHCCP** (et non **RWT** ou **HMESI**).

Compte tenu des temps de simulation beaucoup plus longs pour le modèle VHDL synthétisable que pour le prototype virtuel SystemC, nous avons choisi d'exécuter une application beaucoup plus simple sur l'architecture TSAR mono-processeur. Les paramètres de plateforme sont montrées dans le tableau 8.9. L'application retenue est le *benchmark* Dhrystone, qui n'a pas besoin de système d'exploitation, et qui a été utilisé pour l'analyse de consommation présentée au chapitre 6. Nous comptons simplement le nombre d'accès effectifs au cache instructions, pour évaluer l'efficacité du **Micro-Cache**.

L1 Cache Sets (I & D)	64
L1 Cache Ways (I & D)	4
L1 Cache Words (I & D)	16
L2 Cache Sets	256
L2 Cache Ways	16
L2 Cache Words	16
TLB Sets (I & D)	8
TLB Ways (I & D)	8
CC-TLB Sets (I & D)	8
CC-TLB Ways (I & D)	8

FIGURE 8.9 – Paramètres de plate-forme avec **Micro-Cache**

8.2.2 Analyse des résultats

Le résultat de l'exécution du *benchmark* Dhrystone montre que le nombre d'accès au répertoire du cache instructions est divisé par 2 grâce à l'utilisation du **Micro-Cache**. Ce résultat est un peu décevant, mais il permet néanmoins de réduire la consommation de 9.2%, puisque la consommation du répertoire du cache instructions compte pour 18.4% de la consommation totale du cluster.

Le coût matériel est particulièrement faible dans le cas de la version simplifiée du **Micro-Cache**, puisqu'il ne nécessite qu'une dizaine de bascules supplémentaires, et non le stockage d'une ligne de cache complète.

8.2.3 Conclusion

Le **Micro-Cache** - même dans sa version simplifiée permet de réduire de 9% la consommation énergétique d'un cluster, sans aucune dégradation des performances, puisque les chronogrammes ne sont pas modifiés, pour un coût matériel pratiquement négligeable.

Chapitre 9

Conclusion

Dans cette thèse, j'ai proposé et évalué trois optimisations concernant le système mémoire de l'architecture *Manycore* **TSAR**, visant toutes à réduire la consommation énergétique sans dégrader les performances, et sans remettre en cause la scalabilité du protocole de cohérence des caches.

La stratégie d'écriture immédiate utilisée dans le protocole **DHCCP** de **TSAR**, est scalable jusqu'à 1024 coeurs, mais génère énormément de trafic d'écritures entre les caches L1 et les caches L2. Afin de réduire ce trafic, le protocole **RWT** utilise une méthode hybride pour traiter les écritures dans le cache L1 : Pour les lignes non partagées, le contrôleur de cache L1 utilise la stratégie d'écriture différée, de façon à modifier les lignes localement. Pour les lignes partagées, le contrôleur de cache L1 utilise la stratégie d'écriture immédiate pour éviter l'état de propriété exclusive sur ces lignes partagées. Les expérimentations ont été réalisées sur un prototype virtuel précis au cycle, comportant entre 1 et 512 coeurs. Cette plate-forme utilise un système d'exploitation (***Giet-VM***) adapté aux plate-formes *manycore* et permettant d'éliminer la plupart des goulots d'étranglement introduits par l'OS, ce qui est indispensable pour analyser la scalabilité des protocoles. Les six applications parallèles analysées montrent que, par rapport au protocole **DHCCP**, le protocole **RWT** élimine pratiquement toutes les transactions d'écriture inutiles, permettant une réduction d'au moins un facteur 2 du trafic total sur le réseau (et de la consommation associée), sans aucune dégradation de performance. Nous avons également comparé le protocole **RWT** avec un protocole plus classique, de type **MESI**, utilisant exclusivement la stratégie d'écriture différée. Ce protocole a été implémenté dans le prototype virtuel de l'architecture TSAR, et les résultats montrent que ce protocole **HMESI** est beaucoup moins scalable que **DHCCP** ou **RWT**, et que le trafic généré sur le réseau, et la consommation associée), sont au moins quatre fois plus importants. Ces deux caractéristiques négatives de **HMESI** ont une même cause, qui est l'augmentation importante du trafic de cohérence quand le nombre de coeurs augmente.

Nous considérons donc que le protocole **RWT** est une amélioration significative par rapport au protocole **DHCCP** pour ce qui concerne la consommation énergétique, alors que **DHCCP** était déjà meilleur que **MESI**, en termes de scalabilité, comme en termes de consommation.

Pour ce qui concerne la cohérence des **TLBs**, la solution actuellement implémentée dans **TSAR** impose une contrainte d'inclusivité des **TLBs** dans le cache L1 des données. Cette contrainte entraîne une augmentation parasite du nombre de *Miss* **TLB**, qui

entraîne à son tour une augmentation du nombre de *Miss* sur le cache L1 des données, et finalement un trafic inutile sur le réseau. J'ai proposé un mécanisme, appelé **CC-TLB**, qui permet de relâcher la contrainte d'inclusivité des **TLBs** dans le cache L1 des données. La table **CC-TLB** supprime tous les événements `Scan-TLB` et `Reset-TLB` qui se produisent en cas d'évincement spontané d'une ligne de cache de type *IN-TLB*. En conséquence le nombre de *Miss* **TLB** est très fortement réduit pour toutes les applications exécutées : entre 30% et 70% suivant les applications. Pour un système d'exploitation standard, du type UNIX ou Linux, où les tables de pages utilisées par les **TLBs** ne sont pas répliquées, cette réduction du taux de *Miss* entraîne des gains de performance compris entre 10% et 20%, ainsi qu'une réduction significative du nombre de MISS sur le cache de données, ce qui ne peut que réduire la consommation énergétique. Par rapport au mécanisme actuel, ces améliorations sont obtenues sans augmentation, mais au contraire avec une légère diminution du coût matériel.

Nous considérons ici encore qu'il s'agit d'une amélioration significative du protocole **DHCCP**.

Pour un cluster comportant 4 coeurs, la consommation des caches L1 instructions représente 34% de la consommation totale de l'architecture **TSAR**. Pour réduire cette consommation, nous avons proposé d'introduire un nouveau niveau dans la hiérarchie de cache, appelé ***Micro-Cache***. Nous avons proposé deux versions de ce mécanisme, suivant que le composant RAM utilisé permet (ou non) de lire une ligne de cache de 64 octets en un seul cycle. Dans la première version, le mécanisme proposé permet de diviser par un facteur 4 la consommation des caches d'instruction. Dans la seconde version, seule la consommation du répertoire du cache instruction peut être divisée par 4. Dans les deux versions, il n'y a pas de pénalité sur la performance, puisque les chronogrammes ne sont pas modifiés. Le composant mémoire utilisé pour la réalisation VLSI de l'architecture ne permettant de lire qu'une instruction par cycle, c'est cette seconde version qui a été implémentée et évaluée dans le modèle VHDL synthétisable du cache L1, conduisant à une réduction de 9.2% de la consommation totale d'un cluster.

Les trois optimisations proposées dans ce travail sont arrivées trop tard pour être intégrées dans le premier prototype VLSI de l'architecture **TSAR**, développé par le CEA LETI, puisque les modèles VHDL synthétisables fournis par le Lip6 pour ce prototype VLSI ont été gelés fin 2014. L'introduction de ces optimisations dans les modèles VHDL synthétisables reste donc à faire, mais comme elles ont été introduites et validées dans le prototype SystemC précis au cycle, ce travail à venir ne devrait pas révéler de mauvaises surprises.

Bibliographie

[1] Ahmed Jerraya and Wayne Wolf. *Multiprocessor systems-on-chips*. Elsevier, 2004.

[2] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 681–685. IEEE, 2004.

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

[4] John L Hennessy and David A Patterson. *Computer architecture : a quantitative approach*. Elsevier, 2011.

[5] Sarita V Adve, Vikram S Adve, Mark D Hill, and Mary K Vernon. *Comparison of hardware and software cache coherence schemes*, volume 19. ACM, 1991.

[6] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3) :1–212, 2011.

[7] Norman P Jouppi. Cache write policies and performance. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 191–201. ACM, 1993.

[8] Matthias A Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W Felten, and Jonathan Sandberg. *Virtual memory mapped network interface for the SHRIMP multicomputer*, volume 22. IEEE Computer Society Press, 1994.

[9] Patricia J Teller. Translation-lookaside buffer consistency. *Computer*, 23(6) :26–36, 1990.

[10] Tsar projet. https ://www-soc.lip6.fr/trac/tsar.

[11] Ghassan ALMALESS. *Conception d'un système d'exploitation pour une architecture many-cores à mémoire partagée cohérente de type cc-NUMA*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2014.

[12] Eric Guthmuller. *Architecture adaptative de mémoire cache exploitant les techniques d'empilement tridimensionnel dans le contexte des multiprocesseurs intégrés sur puce*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2013.

[13] Yang GAO. *Contrôleur de cache générique pour une architecture manycores massivement parallèle à mémoire partagée cohérente*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2011.

[14] Ozgur Celebican, Tajana Simunic Rosing, and Vincent J Mooney III. Energy estimation of peripheral devices in embedded systems. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 430–435. ACM, 2004.

[15] Bhavya K Daya, Chia-Hsin Owen Chen, Sivaraman Subramanian, Woo-Cheol Kwon, Sunghyun Park, Tushar Krishna, Jim Holt, Anantha P Chandrakasan, and

Li-Shiuan Peh. Scorpio : a 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 25–36. IEEE, 2014.

[16] tilera. https ://www.tilera.com.

[17] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News*, 12(3) :348–354, 1984.

[18] Amd64 architecture. AMD64 Architecture Programmer's Manual Vol 2 'System Programming'.

[19] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE micro*, 30(2) :16–29, 2010.

[20] Sabela Ramos Garea and Torsten Hoefler. Modelling communications in cache coherent systems. *Technical Report*, 2013.

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

[22] Tsar projet. https ://www-soc.lip6.fr/trac/tsar/wiki/InterconnexionNetworks.

[23] Giet vm. https ://www.almos.fr/trac/giet-vm.

[24] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. Ieee, 2007.

[25] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs : Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[26] David H Bailey. Ffts in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989.

[27] Steven Cameron Woo, Jaswinder Pal Singh, and John L Hennessy. *The performance advantages of integrating block data transfer in cache-coherent multiprocessors*, volume 29. ACM, 1994.

[28] K-means clustering. https ://en.wikipedia.org/wiki/K-means_clustering.

[29] Guy E Blelloch, Charles E Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM, 1991.

[30] Steven Cameron Woo, Jaswinder Pal Singh, and John L Hennessy. *The performance advantages of integrating message passing in cache-coherent multiprocessors*. Computer Systems Laboratory, Stanford University, 1995.

[31] Soclib projet. https ://www.soclib.lip6.fr/home.html.

[32] icu component. https ://www.soclib.fr/trac/dev/wiki/Component/VciIcu.

Annexes

Annexe 1 : La description des automates sur le protocole **RWT**

Le protocole **RWT** est implémenté à base du protocole **DHCCP**. Deux composant, le cache L1 et le cache L2, sont modifiés.

Cache L1

La modification est légère dans les automates du cache L1, l'état de ligne de cache est mis à jour lorsqu'il reçoit la réponse de lecture venant de cache L2. Le cache L1 envoie la requête `Cleanup-data` au cache L2 si la ligne invalidée est *Dirty*.

Cache L2

Dans le cache L2, trois automates est modifiée afin d'ajouter un nouvel état de ligne de cache : l'automate **`READ`**, l'automate **`WRITE`** et l'automate **`Cleanup`**.

Procédure de l'automate **`READ`** :

SI la ligne de cache est présente dans le cache L2 :
  SI le compteur de copies de la ligne de cache est nul :
    Le compteur de copies est incrémenté.
    La réponse au `Read` est envoyée en accompagnant l'état de la ligne.
  SINON :
    SI la ligne est en état `NC` :
      SI la ligne peut être enregistrée dans la ***table IVT*** :
        Cette ligne de cache est enregistrée dans la ***table IVT*** afin que le cache L2 envoie une requête `Multicast Invalidate` au cache L1.
        L'automate **`READ`** traitera la commande suivante.
      SINON :
        Le traitement de la requête est retardé.
    SINON (la ligne est en dans l'état `C`) :
      SI la ligne est en mode ***compteur***.
        Le compteur est incrémenté.
        La réponse au `Read` est envoyée en accompagnant l'état `C`.
      SINON (mode ***liste chaînée***) :
        SI le ***HEAP*** contenant les listes chaînées est plein :
          La ligne de cache est placée en mode compteur et la liste chaînées est libérée.
          Le compteur est incrémenté.
          La réponse au `Read` est envoyée en accompagnant l'état `C`.
        SINON (le ***HEAP*** n'est pas plein) :
          Le compteur est incrémenté.
          L'identité du demandeur (SRCID) correspondant à cette ligne est enregistré dans la liste chaînée.
          La réponse au `Read` est envoyée en accompagnant l'état `C`.
SINON (ligne non présente dans le L2) :
  SI la lecture sur cette ligne vers la mémoire externe est déjà enregistrée dans la ***table TRT***.
    Le traitement de la requête est retardé.
  SINON :
    SI la ***table TRT*** n'est pas pleine :

La lecture vers la mémoire externe est enregistrée.
SINON (***table TRT*** pleine) :
Le traitement de la requête est retardé.

Procédure de l'automate **`WRITE`** :

SI la ligne de cache est présente dans le cache L2 :
SI le compteur de copies est non nul :
SI la ligne de cache est en état `NC` :
SI la ligne peut être enregistrée dans la ***table IVT*** :
Cette ligne de cache et enregistrée dans la ***table IVT*** afin que le cache L2 envoie une requête `Multicast Invalidate` au cache L1.
Le traitement de la requête est retardé.
SINON (***table IVT*** pleine) :
Le traitement de la requête est retardé.
SINON (la ligne est dans l'état `C`) :
SI la ligne de cache est en mode ***compteur*** :
SI la ***table TRT*** n'est pas pleine et la ***table IVT*** n'est pas pleine :
La ligne de cache est mise à jour.
La ligne de cache est invalidée.
Une transaction d'écriture vers la mémoire externe est enregistrée dans la ***table TRT***.
Une requête `Broadcast Invalidate` est enregistrée dans la ***table IVT***.
SINON (une des deux tables est pleine) :
Le traitement de la requête est retardé.
SINON (la ligne est en mode ***liste chaînée***) :
La ligne de cache est mise à jour.
La ligne de cache est marquée *Dirty*.
SI la ligne est partagée par d'autres caches que celui qui écrit :
SI la ***table UPT*** n'est pas pleine :
Une transaction de `Multicast Update` est enregistrée dans la ***table UPT*** afin de mettre la jour le cache L1.
SINON (***table UPT*** pleine) :
On réessaye d'enregistrer la requête de `Multicast Update` après un délai.
SINON (pas d'autres copies) :
La réponse à l'écriture est envoyée.
SINON (nombre de copies nul) :
La ligne de cache est mise à jour.
La ligne de cache est marquée *Dirty*.
La réponse à l'écriture est envoyée.
SINON (la ligne n'est pas dans le L2) :
SI la ***table TRT*** n'est pas pleine :
Une requête de lecture vers la mémoire externe est enregistrée dans la ***table TRT***.
SINON (***table TRT*** pleine) :
La traitement de la requête est retardé.

Procédure de l'automate **`CLEANUP`** :

SI la ligne de cache est présente dans le cache L2 :
SI la ligne est en état `NC` :
SI la requête est `Cleanup-data` :
La ligne de cache est mise à jour.
L'état de cette ligne de cache est changée en `C`.
SI cette ligne de cache est enregistrée dans la ***table IVT*** :
SI le changement d'état fait suite à une requête de lecture :
Le changement d'état est terminé.
Le compteur de copies est mis à 1.
Le SRCID de demandeur est enregistré.
L'enregistrement de l'invalidation est effacé.
La réponse à cette lecture est envoyée avec l'état `C`.
La réponse `Clack` est envoyée.
SI le changement d'état fait suite à une requête d'écriture :
Le compteur de copies est mis à 0.
Le changement d'état est terminé.
L'enregistrement de l'invalidation est effacé.
La réponse `Clack` est envoyée.
SINON (la ligne n'a pas d'entrée dans la ***table IVT***) :
Le compteur de copies est mis à 0.
La réponse `Clack` est envoyée.
SINON (la ligne est dans l'état `C`) :
SI la ligne est en mode ***compteur*** :
Le compteur de copies est décrémenté.
La réponse `Clack` est envoyée.
SINON (mode ***liste chaînée***) :
Le compteur de copies est décrémenté.
L'entrée dans la liste chaînée correspondant au SRCID pour le même type de ligne (instruction ou donnée) est remise dans la liste des éléments libres.
La réponse `Clack` est envoyée.
SINON (la ligne n'est pas présente dans le L2) :
SI la ligne de cache est enregistrée dans la ***table IVT*** :
Le compteur de réponse à l'invalidation est décrémenté.
SI le compteur de réponse devient nul :
L'enregistrement de l'invalidation est effacé.
SI l'invalidation fait suite à une écriture :
La réponse à cette écriture est envoyée.
La réponse `Clack` est envoyée.
Si le type de requête est `Cleanup-data` (la ligne évincée dans le cache L2 est en état `NC`) :
La valeur de l'entrée de la ***table TRT*** correspondant cette ligne est mise à jour.
La ligne dans la ***table TRT*** est envoyée à la mémoire principale.
SINON (le compteur de réponse ne devient pas nul) :
La réponse `Clack` est envoyée.
SINON (la ligne n'a pas d'entrée la la ***table IVT***) :
La réponse `Clack` est envoyée.

Annexe 2 : Description des automates du protocole **HMESI**

Le protocole **HMESI** est implémenté sur la base du protocole **RWT**. Deux composants, le cache L1 et le cache L2, sont modifiés. Un état supplémentaire est introduit dans le composant cache L1, et les automates du cache L1 ne sont que légèrement modifiés par rapport au protocole **RWT**. La plupart des modifications sont effectuées dans les 5 automates du cache L2.

Procédure de L'automate **READ** :

SI la ligne de cache est présente dans le cache L2 :
  SI la requête est un *Read* :
    SI le compteur de la ligne est nul :
      Le compteur est incrémenté.
      La réponse au *Read* est envoyée avec l'état de la ligne actuelle (SHARED ou EXCLUSIVE).
    SI la ligne de cache est dans l'état LOCKED :
      Le traitement de requête est retardé.
    SINON SI la ligne est dans l'état EXCLUSIVE :
      SI cette ligne peut être enregistrée dans la ***table IVT*** :
        Cette ligne de cache est enregistrée dans la ***table IVT*** afin que le cache L2 envoie une requête *CC-UPDT* au cache L1.
        La ligne passe dans l'état LOCKED.
        L'automate **READ** traitera la commande suivante.
      SINON :
        Le traitement de la requête est retardé.
    SINON SI la ligne est en mode SHARED :
      SI la ligne est en mode ***compteur*** :
        Le compteur est incrémenté.
        La réponse au *Read* est envoyée avec l'état SHARED.
      SINON (mode ***liste chaînée***) :
        SI le ***HEAP*** contenant les listes chaînées est plein :
          La ligne de cache est placée en mode ***compteur*** et la liste chaînée est libérée.
          Le compteur est incrémenté.
          La réponse au *Read* est envoyée avec l'état SHARED.
        SINON :
          Le compteur est incrémenté.
          L'identité du demandeur (SRCID) correspondant à cette ligne est enregistré dans la liste chaînée.
          La réponse au *Read* est envoyée avec l'état SHARED.
  SINON (requête *GetM*) :
    SI le compteur de la ligne est nul :
      Le compteur est incrémenté.
      La réponse au *GetM* est envoyée.
    SINON :
      SI cette ligne peut être enregistrée dans la ***table IVT*** :

Cette ligne de cache et la requête au *GetM* sont enregistrés
dans la ***table IVT*** afin que le cache L2 envoie une requête *Multicast Invalidate* ou *Broadcast Invalidate* au cache L1.
La ligne passe dans l'état LOCKED.
SI la ligne est en mode ***liste chaînée*** :
Cette liste est libérée.
L'automate **READ** traitera la commande suivante.
SINON (ligne non présente) :
SI la lecture sur cette ligne vers la mémoire externe est déjà enregistrée
dans la ***table TRT*** :
Le traitement de la requête est retardé.
SINON :
SI la ***table IVT*** n'est pas pleine :
La lecture vers la mémoire externe est enregistrée.
SINON :
Le traitement de la requête est retardé.

Procédure de L'automate **WRITE** :

SI la ligne de cache est présente dans le cache L2 :
SI le compteur de la ligne est nul :
La ligne de cache est mise à jour.
La ligne de cache est marquée *Dirty*.
La réponse d'écriture est envoyée.
SINON :
SI la ligne de cache est en état LOCKED :
Le traitement de la requête est retardé.
SINON :
SI la ligne de cache peut être enregistrée dans la ***Table IVT*** : La ligne de cache est enregistrée dans la ***Table IVT***.
SI la ligne de cache est dans l'état SHARED :
La ligne est mise à jour.
Sinon (état EXCLUSIVE) :
La valeur à écrire est stockée à côté afin de mettre à jour la ligne lorsque le cache L2 reçoit la requête *Cleanup*.
La ligne passe dans l'état LOCKED.
SI la ligne de cache est en mode ***liste chaînée*** :
Toute la chaîne est libérée.
Traitement de la requête suivante.
SINON :
SI la ***table TRT*** n'est pas pleine :
Une requête de lecture vers la mémoire externe est enregistrée dans la ***table TRT***.
SINON :
La traitement de la requête est retardé.

Procédure de L'automate **CLEANUP** :

SI la ligne de cache est présente dans le cache L2 :
SI la ligne de cache est dans l'état LOCKED :
La ***table IVT*** est accédée pour obtenir la requête en cours sur cette ligne de cache.
SI à cette ligne correspond une requête *Read* :
SI la requête *Multi ack-miss* correspondant à cette ligne n'est pas traitée par l'automate **MULTI_ACK** :
SI le type de requête est *Cleanup-data* :
La valeur de cette ligne est mise à jour.
Le compteur de cette ligne est mis à 0.
La réponse *Clack* est envoyée.
SINON :
SI le type de requête est *Cleanup-data* :
La valeur de cette ligne est mise à jour.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
Cette ligne passe dans l'état SHARED.
Le SRCID de la requête *Read* est enregistré.
Le compteur de la ligne de cache est mis à 1.
La réponse au *Read* est envoyée.
La réponse *Clack* est envoyée.
SINON SI à cette ligne de cache correspond une requête *GetM* :
Le compteur dans la ***table IVT*** est décrémenté.
SI le compteur d'invalidation devient nul :
SI le type de requête est *Cleanup-data* :
La valeur de cette ligne est mise à jour.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
Cette ligne passe dans l'état EXCLUSIVE.
Le SRCID de la requête au *Read* est enregistré.
Le compteur de la ligne de cache passe à 1.
La réponse au *GetM* est envoyée.
La réponse *Clack* est envoyée.
SINON :
La réponse *Clack* est envoyée.
SINON SI à cette ligne de cache correspond une requête d'écriture :
Le compteur dans la ***table IVT*** est décrémenté.
SI le compteur d'invalidation devient nul :
SI le type de requête est *Cleanup-data* :
La valeur à jour dans la requête *Cleanup-data* est sauvegardée.
La valeur à jour dans la requête d'écriture est sauvegardée.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
Le compteur de la ligne de cache est égal 0.
La ligne est en état EXCLUSIVE.
La réponse d'écriture est envoyée.
La réponse *Clack* est envoyée.
SINON :

La réponse `Clack` est envoyée.
SINON SI la ligne de cache est dans l'état `EXCLUSIVE` :
SI le type de requête est `Cleanup-data` :
La valeur de cette ligne est mise à jour.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
Le compteur de la ligne de cache est mis à 0.
La réponse `Clack` est envoyée.
SINON (état `SHARED`) :
SI la ligne est en mode ***compteur*** :
Le compteur de copies est décrémenté.
La réponse `Clack` est envoyée.
SINON :
Le compteur de copies est décrémenté.
L'entrée dans la liste chaînée correspondant au SRCID pour le même type de ligne (instruction ou donnée) est remise dans la liste des éléments libres.
La réponse `Clack` est envoyée.
SINON (ligne non présente dans le L2) :
SI la ligne de cache est enregistrée dans la ***table IVT*** :
Le compteur de réponse à l'invalidation est décrémenté.
SI le compteur de réponse devient nul :
L'enregistrement de l'invalidation est effacé.
La réponse `Clack` est envoyée.
SI le type de requête est `Cleanup-data` (la ligne évincée dans le cache L2 est dans l'état `NC`) :
L'entrée de la ***table TRT*** correspondant à cette ligne est mise à jour.
La ligne ligne dans la ***table TRT*** est envoyée à la mémoire principale.
SINON :
La réponse `Clack` est envoyée.
SINON (ligne pas dans la ***table IVT***) :
La réponse `Clack` est envoyée.

Procédure de L'automate **`MULTI_ACK`** :

La ***table IVT*** est accédée.
SI le type de requête est `Multi ack-miss` :
SI la requête `Cleanup` a été traitée par l'automate **`CLEANUP`**.
La ligne de cache passe dans l'état `SHARED`.
Le compteur de la ligne est mis à 1.
Le SRCID du demandeur est enregistré.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
La requête au `Read` est envoyée.
SINON :
L'entrée de la ***table IVT*** correspondant à cette ligne est mise à jour afin que l'automate **`CLEANUP`** réponde à la requête `Read`.
SINON :
La ligne de cache passe dans l'état `SHARED`.
Le compteur de la ligne passe à 2.

Le SRCID du demandeur est enregistré dans la liste chaînée.
L'enregistrement de la ligne dans la ***table IVT*** est effacé.
La réponse au `Read` est envoyée.

Publication

Liu, H., Devigne, C., Garcia, L., Meunier, Q., Wajsburt, F., & Greiner, A. (2015, July). RWT : Suppressing Write-Through Cost When Coherence is Not Needed. In VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on (pp. 434-439). IEEE